

---

# KIMINA LEAN SERVER: TECHNICAL REPORT

---

TECHNICAL REPORT OF KIMINA LEAN SERVER

Project Numina & Kimi Team

## ABSTRACT

We introduce the Kimina Lean Server, an open-source project that enables fast and scalable interaction with Lean 4 via a unified REST API, designed as a simple verifier for reinforcement learning pipelines. Built on top of the Lean FRO's LeanREPL, it combines server-side parallelization by managing multiple Lean REPL processes in parallel, with an LRU caching strategy that reuses Lean imports across multiple requests. These features help reduce initialization overhead and allow large-scale batch processing of Lean code. The client-side interface allows users to submit batches of proofs and receive Lean feedback, including extracted tactics and tactic states via infotree processing. These features enable a high-performance, scalable workflow for both interaction and extraction of proofs, tactics, and tactic states. We open source our implementation on GitHub<sup>1</sup>.

## 1 Introduction

Recent advances in AI for formal mathematics have spurred the development of systems that integrate machine learning models with interactive proof assistants. These interactive environments, with their automatic reward signals based on proof outcomes, open up new possibilities for training AI models using reinforcement learning. With its active community and robust ecosystem, Lean 4 has become the proof assistant of choice for both mathematicians and AI researchers. This popularity highlights a critical need for an efficient interface between Lean 4 and Python, supporting efficient verification and data extraction workflows.

Several projects have tackled the challenge of interfacing Lean 4 with Python (Yang et al. 2023; Aniva et al. 2024; Dressler 2025; Thakur et al. 2025), each with distinct trade-offs. While a lot of progress has been made in the past couple of years, prior tools remain limited. Most frameworks suffer from lack of scalability and performance bottlenecks, as they are not designed to support parallelization, and incur initialization costs for each session (e.g. reloading the mathlib environment for each new verification).

The Kimina Lean Server is designed to overcome these limitations. It performs server-side parallelization of verification and extraction tasks, along with an LRU caching strategy that reuses Lean imports across multiple requests. This enables large-scale batch processing of Lean code, with reduced initialization overhead. The client-side interface allows submitting multiple proof scripts and receive Lean feedback along with parsed tactics and their corresponding tactic states via infotree processing. This extraction is designed to partition proofs into non-overlapping tactics, which is particularly useful for proof search algorithms using a proof completion model. Our contributions include:

- **Server-side parallelization.** Parallel Lean 4 verification across multiple Lean REPL processes to fully utilize multicore CPUs, enabling efficient batch processing of Lean code.
- **LRU caching of imported modules.** In-memory caching of frequently used modules (e.g., Mathlib) to considerably reduce initialization overhead.
- **Client-side API with infotree processing.** A high-level Python API for submitting proof scripts and receiving Lean feedback, along with partitions of proofs with non-overlapping tactics and tactic states extracted from Lean's infotree output, tailored for proof search with proof completion models.

---

<sup>1</sup><https://github.com/project-numina/kimina-lean-server>

## 2 Server

**Parallelization** The server keeps a pool of pre-started Lean REPL workers and spreads incoming verification jobs across them. Each worker runs in its own process, and the workload can occupy all CPU cores at once, ensuring good utilization of available hardware. When a user sends a list of scripts, the server routes them to idle workers, gathers the results, and returns a response.

# CPUs	Total Verification Time (mm:ss)	Average Iterations Rate (it/s)
8	20:11	0.83
16	09:57	1.67
32	05:54	2.82
60	03:51	4.33

Table 1: Performance scaling of proof verification with different CPU configurations (60-core Intel Xeon CPU @ 3.10GHz) on the first 1000 samples from the Goedel-LM/Lean-workbook-proofs dataset. Increasing the number of CPUs consistently translates into higher average iterations rates.

**Caching** Initializing a Lean REPL is expensive because it must load large libraries such as `Mathlib`. To avoid paying that cost for every request, the server reuses workers that already contain the right set of imports. Scripts are split into a header and a body, and workers are indexed by their header in an LRU cache. When a new script arrives, the server grabs a warmed worker with the same header (or creates one if needed) and verifies only the body. This reuse of ready contexts, combined with parallel execution, delivers large speed-ups.

When running the experiments below on ‘Goedel-LM/Lean-workbook-proofs’, we pre-started the cached Lean REPL processes with `import Mathlib` and `import Aesop`

Mode	Total Verification Time (mm:ss)	Average Verification Time (s/it)
Cached	05:50	3.65
Non-Cached	08:14	5.14

Table 2: Performance comparison of cached vs. non-cached verification on a MacBook Pro M2 with 32GB RAM and 10 CPUs on the first 100 samples from the Goedel-LM/Lean-workbook-proofs dataset. Caching leads to significantly faster verification times.

## 3 Client

**Client-server interaction** All the client-server interaction is handled by a single Python call: `verify`. The user provides a list of Lean scripts, and receives one result per script. Nothing else is required: the server takes care of parallel execution and REPL caching behind the scenes.

For every submitted script, the response contains:

- Lean messages: warning or errors generated during checking;
- Environment: the identifier of the REPL environment created to perform the check;
- Elapsed time: the time spent on the verification, in seconds;
- Infotree (optional): the full proof tree produced by Lean, which can then be used to extract tactics and tactic states, as described in the next paragraph.

**Infotree Processing** We process Lean’s infotree outputs to extract a clean, non-overlapping sequence of tactics and their corresponding tactic states. This is valuable both to extract data from existing proofs, and to interact with Lean during proof search.

The infotree already contains nodes corresponding to individual tactics with their corresponding tactic states before and after the tactic is applied. However, these individual tactics often overlap (one tactic can be a sub-tactic of another), and the set of tactics does not cover the entire proof (e.g. some tactics do not appear in the infotree, and whitespaces and comments are not included in the infotree).

The goal of the processing is to convert a list of tactics and tactic states extracted from the infotree into a new list of tactics and tactic states that covers the entire proof, with no overlaps. This is particularly useful to parse a proof into a sequence of code snippets that can be considered as checkpoints for a proof search algorithm performing multi-turn theorem proving with a proof completion model. Each checkpoint can be interpreted as a node in a tree search algorithm. This implementation supports all Lean tactics, including `have` and `let` tactics, `calc` mode, or `conv` mode, which are not always supported by the LeanREPL's Tactic mode. It allows to access all intermediate states present in the infotree.

The processing pipeline consists in the following steps:

1. It first extracts the positions corresponding to each node in the infotree and creates intervals from these positions. At this point, the intervals may overlap and they do not cover the entire proof.
2. It then adjusts the intervals so they become disjoint and partition the whole proof, by setting the end of each interval to the start of the next one.
3. Once the intervals are disjoint, it extracts the corresponding Lean 4 code snippet for each newly created interval.
4. It applies final touches: trailing whitespaces and comments are adjusted to be included with their corresponding tactic, special handling is applied to some tactics (`by`, `calc`), and tactics are merged until every opening bracket and parenthesis is matched by its closing counterpart.

Examples of extracted data are given in the demo notebook<sup>2</sup>.

The main limitation of this approach is that infotrees do not contain intermediate states before term-mode sub-proofs. As a result, these intermediate states are not retrieved by the infotree processing, but the term-mode sub-proofs are still extracted. Exploration of term-mode proofs extraction is left for future work.

## 4 Conclusion

The Kimina Lean Server is a fast, scalable tool to interact with Lean through a unified REST API. The deployed FastAPI server relies on parallelization to spread verification across multiple Lean REPL processes, and on an LRU cache to reuse pre-loaded environments. On the client side, a single `verify` call submits batches of scripts and returns Lean 4 feedback. The infotree processing converts the infotree of a proof into a clean, non-overlapping sequence of tactics and tactic states, which can be used both to extract data from existing proofs, and to interact with Lean during proof search. The open-source implementation therefore supports both large scale verification and data extraction workflows needed for modern learning-based theorem proving.

## References

- Aniva, Leni et al. “Pantograph: A Machine-to-Machine Interaction Interface for Advanced Theorem Proving, High Level Reasoning, and Data Extraction in Lean 4”. In: *arXiv preprint arXiv:2410.16429* (2024).
- Dressler, Oliver. *leanclient: Python client to interact with the lean4 language server*. Jan. 2025. URL: <https://github.com/o0o0o0o/leanclient>.
- Lean Community. *Mathlib Port Status*. 2023. URL: <https://leanprover-community.github.io/mathlib-port-status/>.
- Lean FRO. *A read-eval-print-loop for Lean 4*. <https://github.com/leanprover-community/repl>. 2023.
- Thakur, Amitayush et al. PROOFWALA: *Multilingual Proof Data Synthesis and Theorem-Proving*. 2025. arXiv: 2502.04671 [cs.AI]. URL: <https://arxiv.org/abs/2502.04671>.
- Yang, Kaiyu et al. “LeanDojo: Theorem Proving with Retrieval-Augmented Language Models”. In: *Neural Information Processing Systems (NeurIPS)*. 2023.

<sup>2</sup><https://github.com/project-numina/kimina-lean-server/blob/main/demo/demo.ipynb>

## A Contributions

### Project Numina

Jia Li  
Marco Dos Santos  
Hugues de Saxcé  
Ran Wang  
Mantas Baksys  
Mert Unsal

### Kimi Team

Haiming Wang  
Junqi Liu  
Zhengying Liu

## B Related Work

Over the past couple of years, a number of projects have tackled the challenge of interfacing Lean 4 with Python, each addressing different needs and exhibiting distinct trade-offs. Lean 4 has been released in 2021 and the Lean community has quickly moved to this newer version, to which every file of Mathlib has been ported by July 2023 (Lean Community 2023). The Lean community has also developed a number of tools to facilitate the interaction with Lean 4 from Python. These tools are designed to support various tasks, including data extraction and interaction with Lean’s proving environment.

LeanREPL (Lean FRO 2023) is an official Lean 4 read–eval–print loop that exposes Lean as an interactive subprocess for automated proving tasks. This interface allows submitting one tactic at a time and obtain the new goals, or send an entire proof script and receive the final result. LeanREPL does not support multi-line `have` and `let` tactics, `calc` mode, or `conv` mode natively. LeanDojo (Yang et al. 2023) is a Python toolkit for machine-learning in theorem proving that supports both Lean 3 and Lean 4. It can be used to extract proof data (tactic states, tactics, premises) from Lean projects and to interact programmatically with Lean’s proving environment. Building on LeanREPL, Pantograph (Aniva et al. 2024) overcomes LSP limitations by treating subgoals independently and supporting `have`, `let`, `conv`, and `calc` modes, as well as extracting tactic states and whole proof scripts with comments. Leanclient (Dressler 2025) is a thin Python wrapper around the Lean language server that enables interaction with a Lean language server instance running in a subprocess. It supports querying and changing Lean 4 files via the LSP, and batch processing of files. `itp-interface` (Thakur et al. 2025) is a generic framework for connecting to interactive theorem provers (currently supporting Lean 4 and Coq) and collecting data for learning-based proof search. It provides Python classes to represent tactic states and tactics applications in a gym-like environment.

## C Example Usage

The demo notebook<sup>3</sup> illustrates the three main uses cases of the Kimina Lean Server:

1. Large-scale verification: sending a large batch of Lean codes to the server and getting verification results.
2. Proof data extraction: extracting all tactics and tactic states from a Lean code.
3. Interaction: sending code, receiving verification and extracted data, updating the code.

---

<sup>3</sup><https://github.com/project-numina/kimina-lean-server/blob/main/client/demo/demo.ipynb>