Kimina Lean Server: A High-Performance Lean Server for Large-Scale Verification

Marco Dos Santos [1,2] **Hugues de Saxcé** [1] **Haiming Wang** [3] **Ran Wang** [1]

Mantas Bakšys [1,2] **Mert Ünsal** [1] **Junqi Liu** [3] **Zhengying Liu** [3] **Jia Li** [1]

[1] Project Numina [2] University of Cambridge [3] Moonshot AI

Abstract

Recent progress in neural theorem proving has been driven by the training of large language models on Lean 4 problems via reinforcement learning, a process that requires fast and scalable verification of proofs.

We introduce the Kimina Lean Server, an open-source project designed as a high-performance verifier for reinforcement learning pipelines. Built on top of the Lean REPL (Read-Eval-Print Loop) maintained by the Lean FRO[1], our server combines server-side parallelism by managing multiple Lean processes in parallel with a Least Recently Used (LRU) caching mechanism that reuses Lean imports across requests. On the client side, a lightweight Python package enables submitting proof batches and receiving Lean feedback, including extracted tactics and tactic states.

Together, these features enable a scalable workflow for large-scale verification and data extraction. In our experiments, the Kimina Lean Server outperforms previous Lean interaction tools, achieving a 1.5 to 2 times speedup in verification time. Moreover, its improved efficiency has enabled its use in the large-scale training of state-of-the-art models such as Kimina-Prover [Wang et al., 2025].

We hope that our open-source project[2] will support the neural theorem proving community and accelerate future progress by enabling efficient large-scale verification and proof data extraction.

1 Introduction

Recent advances in AI for formal mathematics have driven the development of systems that integrate machine learning models with interactive proof assistants. These interactive environments enable using proof outcomes as reward signals for reinforcement learning. With its active community and robust ecosystem, Lean 4 [Moura and Ullrich, 2021] has become the proof assistant of choice for both mathematicians and AI researchers. The successful large-scale reinforcement learning training of theorem proving models [Wang et al., 2025, Ren et al., 2025, Chen et al., 2025] highlights a community need for a highly efficient Lean server which supports large-scale verification and data extraction workflows.

Several projects have tackled the challenge of interfacing Lean 4 with Python [Yang et al., 2023, Aniva et al., 2025, Dressler, 2025, Poiroux et al., 2025, Thakur et al., 2025], each with distinct trade-offs.

[1] The Lean Focused Research Organization (FRO) is a non-profit organization to improve Lean's critical systems and guide it toward long-term self-sustainability, cf. About - The Lean FRO

[2] Our code is available at `https://github.com/project-numina/kimina-lean-server`.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Kimina Lean Server: A High-Performance Lean Server for Large-Scale Verification.

Table 1: Verification time (mm:ss) for the 9,419 Lean proofs from NuminaMath-LEAN (lower is better). Our Kimina Lean Server is fastest across all core counts, outperforming the next best baseline by a factor of 1.5 to 2.

Project	8 cores	16 cores	32 cores	64 cores
leanclient	109:55	56:58	30:16	18:01
LeanInteract	87:35	45:51	24:11	12:56
Kimina Lean Server	**42:40**	**21:48**	**11:33**	**7:56**

While a lot of progress has been made in the past couple of years, existing tools face performance bottlenecks and scalability issues. For instance, most are not designed to support parallelization across CPU cores and incur significant initialization costs for each verification. While tools such as leanclient [Dressler, 2025] and LeanInteract [Poiroux et al., 2025] support parallelization, their performance is limited, and a high-performance solution designed specifically for large-scale reinforcement learning pipelines has been lacking.

The Kimina Lean Server is designed to address this gap. It performs server-side parallelization of verification and extraction tasks, and it employs a Least Recently Used caching strategy that reuses Lean imports across multiple requests, reducing initialization costs and enabling large-scale batch processing. On the client side, we provide a lightweight Python package that makes it easy to submit multiple Lean scripts and receive Lean feedback programmatically. In addition, our extraction pipeline processes Lean's infotree output to partition proofs into non-overlapping tactics, which is particularly useful for tree search models.

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.
- **Lightweight client-side Python package.** A simple, high-level API to submit Lean scripts and receive Lean feedback, designed for integration with RL and data-extraction pipelines.
- **Data extraction for tree search.** Our client package can be used to process Lean's infotree output to provide partitions of proofs with non-overlapping tactics and tactic states, a format tailored for tree search models.

2 Lean Server

2.1 Server Side

The server exposes a Representational State Transfer (REST) API that enables proof verification from any programming language. To achieve high performance, its architecture is built on two core principles: server-side parallelization and import caching.

Parallelization The server maintains a pool of Lean REPLs, each running in its own process to ensure optimal CPU core usage. As requests arrive, the server routes each proof to an idle Lean REPL and returns the response. This parallelization strategy enables the server's performance to scale efficiently with available hardware, as demonstrated in our experiments in Section 3.

Caching Initializing a Lean REPL is computationally expensive, primarily due to the time required to load large libraries such as `Mathlib`. To mitigate this overhead, we introduce a caching mechanism that reuses REPLs with pre-loaded imports.

As shown in Figure 1, each incoming script is split into a header $\boldsymbol{H}$ containing only imports and a body **B** with the remaining code. The server then uses the header as a key in a Least Recently Used (LRU) cache to find a "warmed" worker that has already processed the same imports. If a matching worker is found, the server only needs to verify the body of the script, reusing the ready context. This mechanism leads to a significant performance boost, as quantified in our experiments in Section 3.

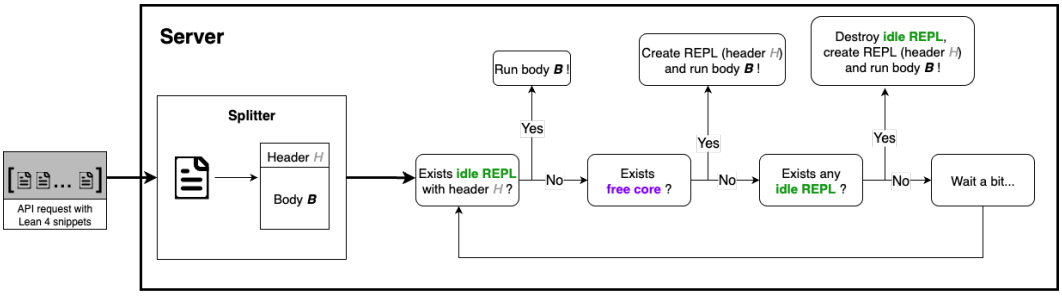

Figure 1: Architecture of the Kimina Lean Server parallelization and caching.

2.2 Client Side

Client-Server Interaction To simplify interaction from Python, we provide a lightweight client as a PyPI package that wraps the server's standard REST API. All interactions are handled by a single call, `check`, which takes a list of Lean scripts and returns a list of results. The server handles all parallel execution and REPL caching behind the scenes.

For each script, the response includes any Lean messages (warnings or errors), the REPL environment identifier, the elapsed time, and an optional infotree. This infotree can be used to extract tactics and tactic states, as described in the next paragraph.

An example of large-scale verification is given in Appendix C.1.

Data Extraction via Infotree Processing To support data extraction and interactive proof search, our Python client package can process Lean's infotree to extract a clean sequence of non-overlapping tactics and their corresponding tactic states.

Our processing pipeline first extracts the position of each tactic, adjusts these positions to eliminate overlaps, extracts the corresponding code snippet for each interval, and performs final adjustments to handle whitespaces, comments, and special tactics. This produces a clean sequence of tactics and states, supporting all Lean tactics, including `have` and `let` tactics, `calc` mode, and `conv` mode, which are not always supported by the LeanREPL's Tactic mode.

An example of proof data extraction is given in Appendix C.2.

3 Experiments

To validate the design and performance of our Kimina Lean Server, we conducted a series of experiments focusing on three key aspects: overall performance compared to existing tools, scalability with increasing CPU cores, and the efficiency gains from our caching mechanism. Finally, we report on its successful application in a large-scale reinforcement learning workflow.

For all benchmarks, experiments were conducted on a Google Cloud Platform (GCP) C4 general-purpose virtual machine, powered by 5th Generation Intel Xeon processors, using Lean v4.15.0. Further details on our experimental setup are provided in Appendix B.

Performance We first evaluated our server's performance against two other available tools that support some level of parallelization: `leanclient` [Dressler, 2025] and `LeanInteract` [Poiroux et al., 2025]. We measured the total time required to process a large dataset of proofs, scaling the number of available CPU cores. For this benchmark, we used a filtered subset of the NuminaMath-LEAN dataset [Wang et al., 2025], which contains 9,419 valid, sorry-free proofs.

The results, summarized in Table 1, show that our server consistently outperforms both baselines across all core counts. On average, our server is 1.5 to 2 times faster than the next best alternative, demonstrating its superior efficiency for large-batch proof verification.

Parallelization and Scalability A key design goal of our server is to maximize hardware utilization. To demonstrate its scalability, we benchmarked its performance on the full NuminaMath-LEAN

dataset while varying the number of available CPU cores from 8 to 64. A single Lean REPL process is single-threaded, and its CPU usage does not exceed one core. Our server leverages this by running one persistent REPL process per dedicated core, enabling highly efficient parallelization.

The results in Table 2 show a strong correlation between added cores and reduced verification time. Scaling from 8 to 32 cores reduces the total time from 42:40 to 11:33, a nearly 4x speedup. This demonstrates our server's ability to effectively scale with available computational resources.

Table 2: Performance scaling of proof verification with different CPU configurations on the NuminaMath-LEAN dataset [Wang et al., 2025].

# CPUs	Total Verification Time (mm:ss)	Average Verification Time (s)
8	42:40	0.272
16	21:48	0.139
32	11:33	0.074
64	7:56	0.051

Caching To quantify the impact of our LRU caching mechanism, we compared two modes on the NuminaMath-LEAN dataset: a "non-cached" mode (fresh REPL per proof) and a "cached" mode (reusing pre-initialized REPLs).

The results in Table 3 show that caching reduces the average verification time from 0.099 seconds to 0.051 seconds, a 1.94x speedup. This demonstrates the important performance benefit of avoiding the costly re-importing of large libraries, an advantage particularly effective for datasets and workflows that frequently reuse the same imports.

Table 3: Performance comparison of cached vs. non-cached verification on the 9,419 Lean proofs from NuminaMath-LEAN with 64 CPU cores. Caching leads to significantly faster verification times.

Mode	Total Verification Time (mm:ss)	Average Verification Time (s)
Cached	7:56	0.051
Non-Cached	15:28	0.099

Validation in Large-Scale Training Beyond synthetic benchmarks, our Lean server has been tested as the core verification engine for the reinforcement learning training of the Kimina-Prover family of models [Wang et al., 2025]. The successful training of these models, which have twice achieved state-of-the-art performance on the miniF2F benchmark [Zheng et al., 2022], serves as a practical validation of our system's efficiency and robustness for neural theorem proving workflows.

4 Conclusion

We introduced the Kimina Lean Server, an open-source Lean server designed for high-performance proof verification in large-scale machine learning workflows. Our server combines server-side parallelism to fully utilize multi-core systems with an LRU caching mechanism that reuses costly module imports across requests.

To facilitate interaction, we provide a lightweight Python package. A single `check` function enables users to submit batches of proofs and receive structured feedback from Lean, abstracting away the complexity of the underlying parallel execution and caching. Our server outperforms existing parallel solutions by 1.5 to 2 times, and has served as the core verification engine for training state-of-the-art theorem-proving models, further validating its efficiency and scalability.

With its direct use of the official Lean REPL, our server is compatible with any Lean version supported by the REPL. Furthermore, the architecture is modular: with minor modifications to the process spawning logic, the server could be adapted to use alternative Lean proof checkers while retaining the same REST API.

We hope that our open-source server will benefit the AI for mathematics community by drastically decreasing the effort required to interact with Lean from Python, thereby accelerating future research.

References

Leni Aniva, Chuyue Sun, Brando Miranda, Clark Barrett, and Sanmi Koyejo. Pantograph: A Machine-to-Machine Interaction Interface for Advanced Theorem Proving, High Level Reasoning, and Data Extraction in Lean 4. In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 104–123, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-90643-5. doi: 10.1007/978-3-031-90643-5_6.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. Seed-prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*, 2025.

Oliver Dressler. leanclient: Python client to interact with the lean4 language server, 1 2025. URL `https://github.com/oOo0oOo/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.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, pages 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-79875-8. doi: 10.1007/978-3-030-79876-5_37. URL `https://doi.org/10.1007/978-3-030-79876-5_37`.

Auguste Poiroux, Viktor Kuncak, and Antoine Bosselut. LeanInteract: A Python Interface for Lean 4, 2025. URL `https://github.com/augustepoiroux/LeanInteract`.

ZZ Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, et al. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

Amitayush Thakur, George Tsoukalas, Greg Durrett, and Swarat Chaudhuri. proofwala: Multilingual proof data synthesis and theorem-proving, 2025. URL `https://arxiv.org/abs/2502.04671`.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-Prover Preview: Towards Large Formal Reasoning Models with Reinforcement Learning. 2025. URL `http://arxiv.org/abs/2504.11354`.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Neural Information Processing Systems (NeurIPS)*, 2023.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. miniF2F: a cross-system benchmark for formal Olympiad-level mathematics. In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=9ZPegFuFTFv`.

A 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 was released in 2021 [Moura and Ullrich, 2021] and the community has since migrated the entirety of Mathlib to this new version [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.

The primary low-level interface for programmatic interaction is the official LeanREPL [Lean FRO, 2023], a read-eval-print loop that exposes Lean as an interactive subprocess. This interface allows submitting one tactic at a time and obtaining the new goals, or sending an entire proof script and receiving the final result. LeanREPL does not support multi-line `have` and `let` tactics, `calc` mode, or `conv` mode natively. Pantograph [Aniva et al., 2025] was developed as a more feature-rich alternative, overcoming some of LeanREPL's 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. Both of these tools are inherently single-threaded and are not designed for the high-throughput verification required by large-scale reinforcement learning pipelines.

To address the need for large-scale verification, several tools have focused on enabling parallel processing of Lean code. leanclient [Dressler, 2025] interfaces with Lean via its Language Server Protocol (LSP) to support batch processing of files in parallel. LeanInteract [Poiroux et al., 2025] offers a similar capability by managing multiple REPL instances. Although these tools support parallel processing, their performance remains limited for large-scale training, and they are not designed to extract proof data such as individual tactics and tactic states.

Other projects like ProofWala [Thakur et al., 2025] and LeanDojo [Yang et al., 2023] provide more comprehensive gym-like environments for data extraction and proof search. These tools are not optimized for the verification speed needed in modern RL workflows.

Our work builds on these previous efforts by providing a solution specifically engineered for high-performance verification and data extraction, addressing the scalability needs of training state-of-the-art language models.

B Experiments

B.1 Experimental Setup

All experiments were conducted on a Google Cloud Platform (GCP) C4 general-purpose virtual machine. We selected the C4 instance type, powered by 5th Generation Intel Xeon processors, as it is optimized for CPU-based inference.

The machine was configured with 72 vCPUs, 540 GB of RAM (7.5 GB per vCPU), and a 200 GB Hyperdisk Balanced storage volume with 20,000 provisioned IOPS (Input/Output operations Per Second). To ensure predictable performance for these compute-bound tasks, we disabled Simultaneous Multi-Threading (SMT), which can hinder overall application performance and add unpredictable variance to jobs.

All experiments were run fully locally to eliminate any network latency.

The software environment consisted of a standard Linux distribution with Miniconda, Git, elan, and Lean v4.15.0, which was used for all benchmarks.

B.2 Dataset Processing

Our benchmark dataset is a filtered subset of the NuminaMath-LEAN dataset [Wang et al., 2025]. We performed a three-step processing pipeline to ensure data quality and validity:

1. First, we processed the original dataset to remove duplicate entries based on their unique identifiers (column `uuid`).
2. Next, we filtered for complete proofs (column `ground_truth_type` equal to `complete`).

3. Finally, we performed a validation step and removed a small number of proofs that either contained `sorry` (26 proofs) or failed to compile with our target Lean version (3 proofs).

This process resulted in a final dataset of 9,419 valid, sorry-free proofs.

B.3 Step-by-Step Protocol

To ensure a fair comparison, we followed a specific protocol for our server and the two baselines, `leanclient` and `LeanInteract`. The Python scripts used to run the `leanclient` and `LeanInteract` experiments are available in the supplementary material.

Kimina Lean Server The server was configured by setting environment variables to match the core count of each experiment (e.g., `LEAN_SERVER_MAX_REPLS`=8, 16, 32, 64). We also increased the request timeout (`LEAN_SERVER_MAX_WAIT`) and memory-per-REPL (`LEAN_SERVER_MAX_REPL_MEM`). The benchmark was executed using the provided Python client, which submits all 9,419 proofs in batches to the server API.

leanclient For `leanclient`, we created a new Lake project and configured its `lean-toolchain` and `lakefile.lean` to use the exact same Lean and Mathlib versions (v4.15.0) as our server. The verification was then performed using a Python script that interacted with this Lake project.

LeanInteract For `LeanInteract`, we used a script that followed the library's official recommendation for parallel processing: using one `LeanREPLConfig` instance, and one `AutoLeanServer` instance per process. We did not specify `add_to_session_cache=True` when calling `run` on a command because each proof gets its own `AutoLeanServer`, and having to manage pools of reusable `AutoLeanServer` instances would involve exactly the engineering effort put into our Lean server.

C Example Usage

We illustrate the two main use cases of our Kimina Lean Server: large-scale verification of Lean scripts and structured extraction of proof data.

C.1 Large-scale Verification

This example demonstrates how to use the Kimina Lean Server to verify a large batch of Lean scripts. The simplest method is to use the high-level `run_benchmark` function, which handles data loading and processing automatically. The following verifies the first 1000 samples from the NuminaMath-LEAN dataset [Wang et al., 2025].

```python
from datasets import load_dataset
from kimina_client import KiminaClient

client = KiminaClient()

client.run_benchmark(dataset_name="AI-MO/NuminaMath-LEAN",
                     n=1000,
                     batch_size=8,
                     max_workers=10)
```

For a more controlled and flexible approach, the following example shows how to manually prepare the data and call the core `check` function directly:

```python
from datasets import load_dataset
from kimina_client import KiminaClient
from kimina_client.models import Snippet

# Load the dataset
dataset = load_dataset("AI-MO/NuminaMath-LEAN", split="train")
dataset = dataset.filter(lambda x: x["ground_truth_type"] == "complete")
dataset = dataset.select(range(1000))

# Deduplicate on uuid
seen = set()
def keep_first(ex):
    u = ex["uuid"]
    if u in seen:
        return False
    seen.add(u)
    return True

dataset = dataset.filter(keep_first)

# Send to the Kimina Lean Server for validation
client = KiminaClient()

snippets = [
    Snippet(id=str(dataset[i]["uuid"]), code=dataset[i]["formal_ground_truth"])
    for i in range(len(dataset))
]

resp = client.check(snippets, timeout=120)

# Check the validation results for sample with a specific uuid
sample_uuid = "84f26e70-3dfd-589b-b7d0-7792576f0cc9"
for r in resp.results:
    if r.id == sample_uuid:
        print(f"Sample {sample_uuid} status: {r.analyze().status.value}")
        break
```

C.2 Proof Data Extraction

This example demonstrates the client package's capability to process Lean's infotree output into a structured sequence of proof steps, each containing a tactic and its corresponding tactic states.

As an example, we take the following proof from the NuminaMath-LEAN dataset [Wang et al., 2025].

```
import Mathlib

/- Given that the product \( a \cdot b \cdot c = 1 \), what is the value of the
    following expression?

\frac{a}{a b + a + 1} + \frac{b}{b c + b + 1} + \frac{c}{c a + c + 1}
-/
theorem algebra_4013 {a b c : ℝ} (h : a * b * c = 1) (haux : 1 + a + a * b ≠ 0) :
    a / (a * b + a + 1) + b / (b * c + b + 1) + c / (c * a + c + 1) = 1 := by
  -- need ne_zero condition to perform division
  have : a * b * c ≠ 0 := by rw [h]; norm_num
  have ha : a ≠ 0 := left_ne_zero_of_mul <| left_ne_zero_of_mul this
  have hb : b ≠ 0 := right_ne_zero_of_mul <| left_ne_zero_of_mul this
  --  Multiply the second fraction by \(a\).
  conv => lhs; lhs; rhs; rw [← mul_div_mul_left _ _ ha]
  --  Multiply the third fraction by \(ab\).
  conv => lhs; rhs; rw [← mul_div_mul_left _ _ (mul_ne_zero ha hb)]
  -- Thus, we get:
  --  \[
  --  \frac{a}{ab + a + 1} + \frac{ab}{abc + ab + a} + \frac{abc}{abca + abc + ab}
  --  \]
  rw [show a * (b * c + b + 1) = a*b*c + a*b + a by ring]
  rw [show a*b*(c * a + c + 1) = a*b*c*a + a*b*c + a*b by ring]
  -- **Simplify the expression using \(abc = 1\):**
  rw [h, one_mul]
  ring_nf
  -- **Combine the terms with the same denominator:**
  rw [← add_mul]
  nth_rw 2 [← one_mul (1 + a + a * b)⁻¹]
  rw [← add_mul, show a * b + a + 1 = 1 + a + a * b by ring]
  exact mul_inv_cancel₀ haux
```

First, we use the Python client to call the server with the `infotree="tactics"` option to retrieve the raw proof trace. Then, we use helper functions provided by the client library to parse this trace into a clean list of proof steps.

```
dataset = load_dataset("AI-MO/NuminaMath-LEAN", split="train")

sample = dataset[0]

client = KiminaClient()

snippets = [
    Snippet(id=str(dataset[0]["uuid"]), code=dataset[0]["formal_ground_truth"])
]

resp = client.check(snippets, timeout=120, infotree="tactics")

infotree = resp.results[0].response["infotree"]
header, body = split_snippet(sample["formal_ground_truth"])
intervals = extract_data(infotree, body)
```

The resulting `intervals` object is a list where each element represents a distinct proof step with three key fields: `goalsBefore`, `tactic` and `goalsAfter`.

The first five steps for the example proof are shown below:

```
INTERVAL 1
-----
Goals before:
a b c : ℝ
h : a * b * c = 1
haux : 1 + a + a * b ≠ 0
⊢ a / (a * b + a + 1) + b / (b * c + b + 1) + c / (c * a + c + 1) = 1
-----
Tactic:
by
  -- need ne_zero condition to perform division
  have : a * b * c ≠ 0 :=
-----
Goals after:
a b c : ℝ
h : a * b * c = 1
haux : 1 + a + a * b ≠ 0
⊢ a * b * c ≠ 0
--------------------
INTERVAL 2
-----
Goals before:
a b c : ℝ
h : a * b * c = 1
haux : 1 + a + a * b ≠ 0
⊢ a * b * c ≠ 0
-----
Tactic:
 by rw [h];
-----
Goals after:
a b c : ℝ
h : a * b * c = 1
haux : 1 + a + a * b ≠ 0
⊢ 1 ≠ 0
--------------------
INTERVAL 3
-----
Goals before:
a b c : ℝ
h : a * b * c = 1
haux : 1 + a + a * b ≠ 0
⊢ 1 ≠ 0
-----
Tactic:
 norm_num
-----
Goals after:
a b c : ℝ
h : a * b * c = 1
haux : 1 + a + a * b ≠ 0
this : a * b * c ≠ 0
⊢ a / (a * b + a + 1) + b / (b * c + b + 1) + c / (c * a + c + 1) = 1
--------------------
INTERVAL 4
-----
Goals before:
a b c : ℝ
h : a * b * c = 1
haux : 1 + a + a * b ≠ 0
this : a * b * c ≠ 0
⊢ a / (a * b + a + 1) + b / (b * c + b + 1) + c / (c * a + c + 1) = 1
-----
```

```
Tactic:

  have ha : a ≠ 0 := left_ne_zero_of_mul <| left_ne_zero_of_mul this
-----
Goals after:
a b c : ℝ
h : a * b * c = 1
haux : 1 + a + a * b ≠ 0
this : a * b * c ≠ 0
ha : a ≠ 0
⊢ a / (a * b + a + 1) + b / (b * c + b + 1) + c / (c * a + c + 1) = 1
--------------------
INTERVAL 5
-----
Goals before:
a b c : ℝ
h : a * b * c = 1
haux : 1 + a + a * b ≠ 0
this : a * b * c ≠ 0
ha : a ≠ 0
⊢ a / (a * b + a + 1) + b / (b * c + b + 1) + c / (c * a + c + 1) = 1
-----
Tactic:

  have hb : b ≠ 0 := right_ne_zero_of_mul <| left_ne_zero_of_mul this
-----
Goals after:
a b c : ℝ
h : a * b * c = 1
haux : 1 + a + a * b ≠ 0
this : a * b * c ≠ 0
ha : a ≠ 0
hb : b ≠ 0
⊢ a / (a * b + a + 1) + b / (b * c + b + 1) + c / (c * a + c + 1) = 1
```