LLMSTEP: LLM proofstep suggestions in Lean

Sean Welleck
University of Washington
Carnegie Mellon University

Rahul Saha
rsaha@alumni.princeton.edu

Abstract

We present LLMSTEP, a tool for integrating a language model into the Lean proof assistant. LLMSTEP is a Lean 4 tactic that sends a user's proof state to a server hosting a language model. The language model generates suggestions, which are checked in Lean and displayed to a user in their development environment. We provide a baseline language model, along with code for fine-tuning and evaluation to support further development. We provide server implementations that run on CPU, a CUDA GPU, or a Google Colab notebook, as a step towards fast, effective language model suggestions for any user.

1 Introduction

Interactive proof assistants such as Lean [de Moura et al., 2015], Isabelle [Wenzel et al., 2008], and Coq [Team, 2019] enable the verification of mathematics and software using specialized programming languages [Avigad, 2023, Ringer et al., 2019]. The emerging area of neural theorem proving integrates neural language models with interactive proof assistants [First et al., 2023, Polu and Sutskever, 2020, Polu et al., 2022, Yang et al., 2023, Welleck, 2023]. Doing so can be mutually beneficial: proof assistants provide correctness guarantees on language model outputs, while language models may help make proof assistants easier to use. A key part of proof development is determining which step to take next at each state of a proof (i.e., which *tactic* to use). Therefore, a tool that suggests useful next steps within a user's development environment could significantly ease proof development.

We present LLMSTEP, a tool for suggesting proof steps (i.e., tactics) with a language model in the Lean proof assistant (Figure 1). LLMSTEP is a Lean 4 tactic that sends a user's proof state to a server hosting a language model. The language model generates suggestions, which are checked in Lean and displayed to a user in their development environment. LLMSTEP is agnostic to the choice of language model, learning framework, and evaluation framework. We provide a baseline language model and example code for fine-tuning and evaluation. The baseline language model is fine-tuned for a standard tactic-prediction task [Han et al., 2022], and outperforms recent open-source tactic-prediction models. Finally, LLMSTEP supports several runtimes, with servers that run on CPU, a GPU, or in a Google Colab notebook, as a step towards fast, powerful language model suggestions for any user.[1]

2 Related Work

Automatically generating proof steps with language models is an active area of research (e.g., Polu and Sutskever [2020], Han et al. [2022], Yang et al. [2023], Azerbayev et al. [2023], Welleck [2023]). Closest to our work is the `gpt-f` tactic from Han et al. [2022], which generates suggestions in Lean 3 by calling a (now disabled) Open-AI API. LLMSTEP is inspired by the idea of language-model based suggestions, but differs in several ways: (1) LLMSTEP is built with open-source components, and can run on a user's own device. (2) LLMSTEP supports prefixed and checked suggestions, detailed below. (3) LLMSTEP is in Lean 4, requiring an implementation using Lean 4 metaprogramming. (4)

[1] `https://github.com/wellecks/llmstep`

37th Conference on Neural Information Processing Systems (NeurIPS 2023) Workshop on MATH-AI.

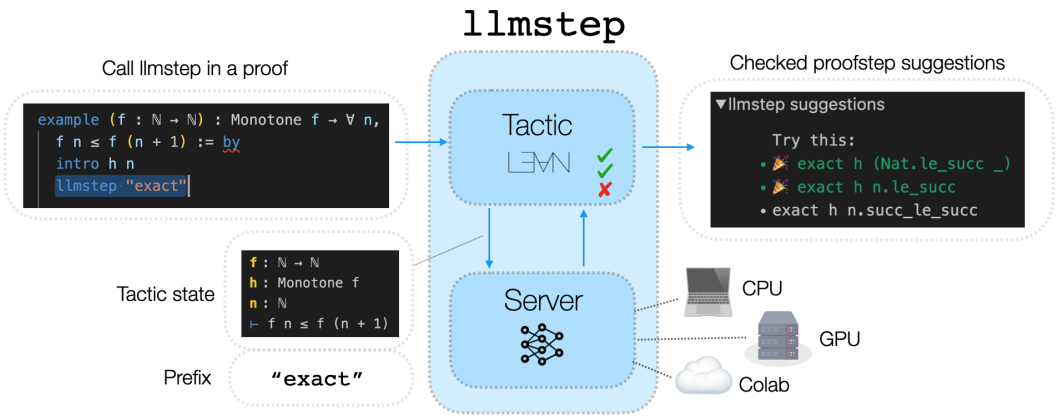

Figure 1: LLMSTEP is a tool for receiving proofstep suggestions from a language model. A user calls LLMSTEP in a proof (*left*), which sends the current state of the proof and a prefix to a server. A language model generates proofstep suggestions which are checked in Lean and shown to the user (*right*). LLMSTEP supports a variety of compute environments, with servers that run on CPU, CUDA GPU, or Google Colab. LLMSTEP is implemented as a Lean 4 metaprogram with Python servers.

LLMSTEP provides code for fine-tuning and evaluating future models. Recently, after the release of LLMSTEP, LeanInfer [Song et al., 2023] provides the ability to run supported language models on CPU through Lean's Foreign Function Interface (FFI). Similar to LLMSTEP it provides tactic suggestions, and uses LLMSTEP's utilities to check, format, and display its suggestions. LLMSTEP additionally supports prefixed suggestions, offers fast GPU inference, and is agnostic to the language model implementation. Proofster [Agrawal et al., 2023] is a related tool offering machine-learning based proof synthesis in Coq via a web interface, while LLMSTEP offers Lean 4 language-model tactic suggestions in the development environment.

3 Approach

LLMSTEP is called by writing `llmstep <prefix>` within a proof, which returns suggestions that start with `<prefix>` (for instance, `llmstep ""` or `llmstep "exact"`). LLMSTEP uses Lean to check whether each suggestion is valid and/or completes the proof. The suggestions are displayed in the Lean 4 VS Code Infoview, which is a standard interface used in proof development. A user can click a suggestion, which places it in the proof. The proof is either complete, or it transitions to the next state and the user continues writing the proof. We detail LLMSTEP's implementation below.

3.1 Implementation

LLMSTEP consists of three parts: (1) a Lean *tactic*, (2) a *language model*, (3) a *server*.

Lean tactic. Writing a proof can be seen as a sequential process $(x_1, y_1), (x_2, y_2), \ldots$ of *states* x_t and *tactics* y_t. A state contains what is left to prove (the *goal*), and available information (the *hypotheses*). A *tactic* transitions the proof to a new state. If the state contains no remaining goals, the proof is complete. Concretely, a user applies tactics by writing Lean code, Lean keeps track of the state, and the development environment shows the state and the written code.

LLMSTEP is itself a tactic. LLMSTEP takes a prefix as an argument, i.e., a sequence of tokens that will start the suggested tactics. For instance, `"intr"` would lead to suggested tactics that start with `"intr"`, such as `intro h x`. LLMSTEP sends the current state x_t and the prefix to a server, LLMSTEP receives suggestions in response, and LLMSTEP checks each suggestion using Lean. Namely, a checked suggestion is *valid* if applying the tactic leads to a state with no errors and at least one goal. A tactic is *complete* if it leads to a state with no errors and no goals. Otherwise the tactic is *invalid*. LLMSTEP displays complete, valid, and invalid tactic suggestions using different colors.

Language model. LLMSTEP uses a language model to predict the next tactic given the current tactic state. While the language model can be arbitrary, one approach is to use a model that has been fine-tuned on (state, next-tactic) examples. For instance, the default language model in LLMSTEP is fine-tuned on sequences of the form:

```
[GOAL]tactic-state[PROOFSTEP]next-tactic<|endoftext|>
```

This format corresponds to the proofstep objective described in Han et al. [2022].

By default, LLMSTEP uses a `Pythia 2.8b` language model [Biderman et al., 2023] fine-tuned on (state, next-tactic) examples extracted from Lean Mathlib [mathlib, 2020] via the LeanDojo Benchmark 4 dataset [Yang et al., 2023]. The `Pythia 2.8b` model is publicly available on Huggingface (link). LLMSTEP is agnostic to the language model implementation, and includes direct support for ReProver [Yang et al., 2023] and other Huggingface models.

Server. LLMSTEP uses a server to handle requests from the Lean tactic and host the language model. The server queries the language model and relays responses back to the Lean tactic. The server is the key computational bottleneck in LLMSTEP, as it hosts a (possibly large) language model. LLMSTEP supports a variety of compute constraints, with servers that run on CPU, a CUDA GPU, or a Google Colab notebook with GPU, as well as a server with fast inference via vLLM [Kwon et al., 2023].

Usage. First, the user starts a server based on their hardware. Currently, servers can run on CPU, CUDA GPU, or Google Colab notebooks. Second, the user imports LLMSTEP as a Lean 4 package. Third, the user calls LLMSTEP by writing `llmstep <prefix>`, which returns suggestions that start with the prefix passed to LLMSTEP. The suggestions are displayed in the Infoview.

4 Evaluation

First, we benchmark the default language model's utility for providing suggestions via proof search– i.e., attempting to fully prove theorems using the language model and a search algorithm.

Proof search. Proof search requires a search algorithm and a method for interacting with Lean. We use best-first search, and provide a self-contained implementation with LeanDojo interaction.

Best-first search is parameterized by the maximum number of generated tactics, defined as the number of attempts $\times$ expansion size per iteration $\times$ maximum iterations, subject to a timeout. We use a 10 minute timeout as in Yang et al. [2023], and use beam search with expansion size 32 based on memory constraints. We compare the Pythia model to ReProver [Yang et al., 2023] without retrieval. To equal the 64 expansions used in Yang et al. [2023], we also report results with a second attempt of 32 samples (i.e., 2×32). We report mathlib4-test from Yang et al. [2023] and miniF2F-test from Thakur et al. [2023], since miniF2F without retrieval was not available in Yang et al. [2023].

Model	Search	mathlib4-valid	mathlib4-test	miniF2F-valid	miniF2F-test
ReProver	1×64	–	48.6%	–	22.1% (54/244)
Pythia 2.8b	1×32	48.8%	47.6%	26.2%	27.9% (68/244)
Pythia 2.8b	2×32	51.2%	50.1%	26.2%	27.9% (68/244)

Table 1: Proofsearch on the Lean Dojo random evaluation splits and miniF2F.

We validate the Pythia model in Table 1, finding that it can exceed the number of closed theorems by ReProver on Lean Dojo Benchmark 4 and miniF2F. Note that LLMSTEP supports suggestions from either model. The ReProver model is particularly useful on CPU, as we discuss next.

Runtime. We tested the runtime of LLMSTEP with different compute environments and hardware. The experiments were done on a set of 17 examples, each containing a tactic state and a prefix. For each example, we measured the time for the server to return N suggestions, using LLMSTEP with the `Pythia 2.8b` and ReProver (`leandojo-lean4-tacgen-byt5-small`) language models.

As shown in Table 2, LLMSTEP with GPU-based inference can yield suggestions in 1 second or less, with vLLM inference approaching real-time (0.11s). As shown in Figure 2, vLLM remains fast as the number of suggestions increases. Note that vLLM does not support the model architecture used in

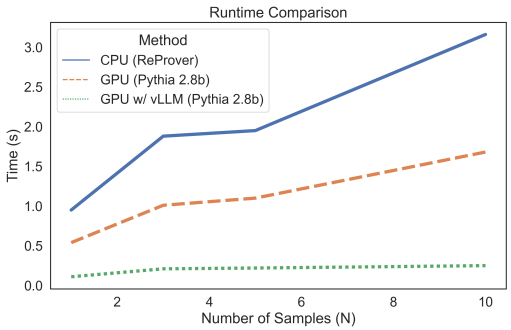

Figure 2: Runtime comparison of different compute environments and models supported by LLMSTEP.

Compute	Model	Hardware	N	Time
CPU	Pythia 2.8b	MB Pro 2.6 GHz 6-Core Intel Core i7	1	8.39s
CPU	ReProver	MB Pro 2.6 GHz 6-Core Intel Core i7	1	0.95s
CPU	Pythia 2.8b	MB Pro 2.6 GHz 6-Core Intel Core i7	10	49.10s
CPU	ReProver	MB Pro 2.6 GHz 6-Core Intel Core i7	10	3.16s
Colab GPU	Pythia 2.8b	NVIDIA T4 GPU	1	1.01s
Colab GPU	Pythia 2.8b	NVIDIA T4 GPU	10	1.68s
GPU w/ vLLM	Pythia 2.8b	NVIDIA GTX 1080Ti	1	0.11s
GPU w/ vLLM	Pythia 2.8b	NVIDIA GTX 1080Ti	10	0.25s

Table 2: Runtime experiments done with different compute resources and hardware, using the `Pythia 2.8b` and Reprover models in LLMSTEP. N refers to the number of generated suggestions. The experiments report average runtime across 17 examples.

ReProver. However, on CPU the ReProver model is much faster due to its small parameter count. Therefore, we suggest using Pythia when a GPU is available, and Reprover on CPU.

Qualitative examples. Figures 1 and 3 show example suggestions made by LLMSTEP.

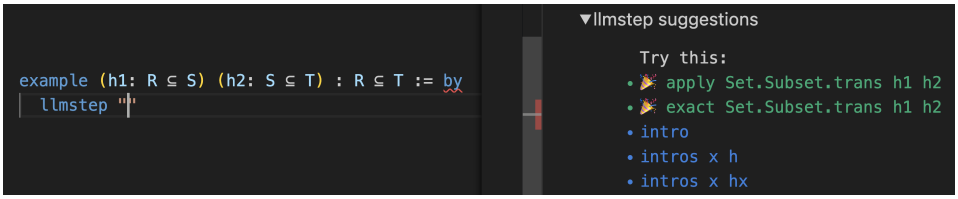

Figure 3: In this example, the user inputs an empty prefix. LLMSTEP returns five suggestions. The first two suggestions each finish the proof, and the last two are valid.

5 Towards contextual suggestions

Real-world proofs often use theorems and definitions that are unique to the proof development. A key limitation of the models described above is that they lack *context* beyond the current proof state. That is, the model $p_\theta(y_t|x_t)$ only receives the current proof state x_t as input. As a result, the model cannot use newly defined theorems, definitions, and other information from the current proof development, unless the proof development was seen during training.

To remove this limitation, we explore LLMSTEP suggestions that use the *preceding file contents* as additional context. To do so, we prompt and generate from a suitable language model two times:

$$y_t \sim p(y_t|z_t), \tag{1}$$

$$y_t \sim p(y_t|x_t; \{(x', y')\}), \tag{2}$$

where z_t is the preceding contents of the source file up to invocation of LLMSTEP, and $\{(x', y')\}$ denotes a set of (state, tactic) examples. We return the union of generated tactics as suggestions. Naturally, there are many other possible ways to incorporate context, which we leave for future work.

We use the LLEMMA [Azerbayev et al., 2023] language model, since it was shown to have few-shot tactic prediction and other related capabilities in Azerbayev et al. [2023]. We present two case studies here, and implement contextual suggestions in the LLMSTEP tool for users to explore further.

The first case study in Figure 4 is a toy example demonstrating the need for contextual suggestions. The theorem requires using properties of the newly defined `my_object`. The contextual LLMSTEP suggestions use the newly defined structure's `cool_property` successfully, completing the proof.

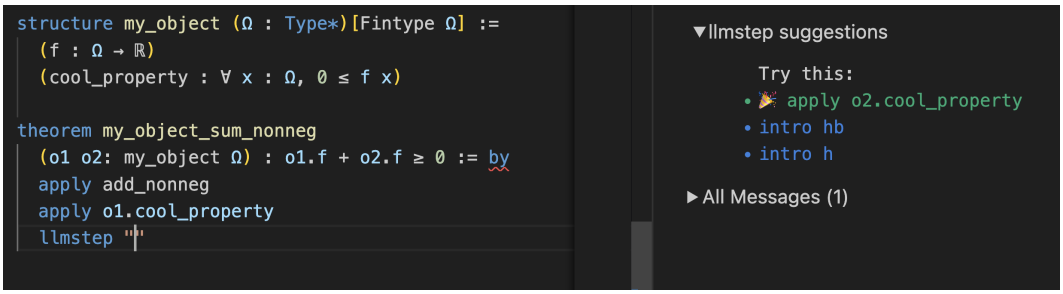

Figure 4: Contextual LLMSTEP suggestions using LLEMMA.

The second case study in Figure 5 shows LLMSTEP within the real-world formalization of the Cedar policy language from Amazon Web Services (github.com/cedar-policy/cedar-spec/cedar-lean). The theorem and its proof rely on project-specific code (e.g., `Request`, `scope_analysis_is_sound`) that are outside the domain of models trained only on Mathlib. The contextual LLMSTEP suggestions successfully use `scope_analysis_is_sound` to finish the proof.

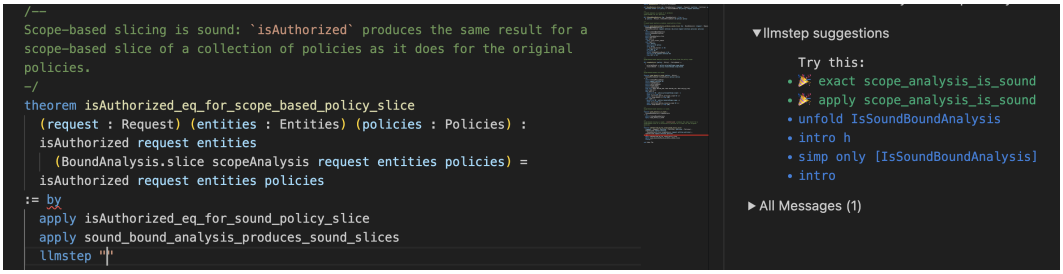

Figure 5: Contextual LLMSTEP suggestions on a formalization of the AWS Cedar policy language.

6 Conclusion

We present LLMSTEP, a tool designed to make it easy for a Lean 4 user to obtain tactic suggestions from a language model. In addition, we provide a fine-tuned language model that achieves strong performance on mathlib and miniF2F. Active areas of work include fast CPU inference, improved models, and tasks beyond tactic prediction. We hope that LLMSTEP's simple, model-agnostic recipe opens up new research avenues on generative tools for formal mathematics and verification.

7 Acknowledgements

We thank Mario Carneiro, Zhangir Azerbayev, and Scott Morrison for valuable guidance and feedback.

References

Arpan Agrawal, Emily First, Zhanna Kaufman, Tom Reichel, Shizhuo Zhang, Timothy Zhou, Alex Sanchez-Stern, Talia Ringer, and Yuriy Brun. Proofster: Automated Formal Verification. In *Proceedings of the Demonstrations Track at the 45th International Conference on Software Engineering*

(ICSE), pages 26–30, Melbourne, Australia, May 2023. doi: 10.1109/ICSE-Companion58688.2023.00018. DOI: 10.1109/ICSE-Companion58688.2023.00018.

Jeremy Avigad. Mathematics and the formal turn. *Bulletin (New Series) of the American Mathematical Society, Received by the editors October 2, 2023.*, 2023. URL https://www.andrew.cmu.edu/user/avigad/Papers/formal_turn.pdf.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. *arXiv preprint arXiv:2310.06786*, 2023.

Stella Rose Biderman, Hailey Schoelkopf, Quentin G. Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, Usvsn Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. Pythia: A suite for analyzing large language models across training and scaling. *ArXiv*, abs/2304.01373, 2023.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer, 2015.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models, 2023.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=rpxJc9j04U.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

mathlib. The lean mathematical library. In *CPP 2020 - Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, co-located with POPL 2020*, 2020. ISBN 9781450370974. doi: 10.1145/3372885.3373824.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning, 2022.

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *Foundations and Trends in Programming Languages*, 2019. ISSN 23251131. doi: 10.1561/2500000045.

Peiyang Song, Kaiyu Yang, and Anima Anandkumar. Leaninfer: Neural network inference in lean 4. https://github.com/lean-dojo/LeanInfer, 2023.

The Coq Development Team. The coq proof assistant, oct 2019. URL https://doi.org/10.5281/zenodo.3476303.

Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. A language-agent approach to formal theorem-proving. *ArXiv*, abs/2310.04353, 2023. URL https://api.semanticscholar.org/CorpusID:263829101.

Sean Welleck. Neural theorem proving tutorial. https://github.com/wellecks/ntptutorial, 2023.

Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. The isabelle framework. In *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings 21*, pages 33–38. Springer, 2008.

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.