RocqStar: Leveraging Similarity-driven Retrieval and Agentic Systems for Rocq generation

Andrei Kozyrev *
JetBrains Research
Berlin, Germany

Nikita Khramov *
JetBrains Research
Berlin, Germany

Gleb Solovev
JetBrains Research
Berlin, Germany

Anton Podkopaev
JetBrains Research
Amsterdam, the Netherlands
Constructor University
Bremen, Germany

ABSTRACT

Interactive Theorem Proving was repeatedly shown to be fruitful when combined with Generative Artificial Intelligence. This paper assesses multiple approaches to Rocq generation and illuminates potential avenues for improvement. We identify retrieval-based premise selection as a central component of effective Rocq proof generation and propose a novel approach based on a self-attentive embedder model. The evaluation of the designed approach shows up to 28% relative increase of the generator's performance. We tackle the problem of writing Rocq proofs using a multi-stage agentic system, tailored for formal verification, and demonstrate its high effectiveness. We conduct an ablation study and demonstrate that incorporating multi-agent debate during the planning stage increases the proof success rate by 20% overall and nearly doubles it for complex theorems, while the reflection mechanism further enhances stability and consistency.

KEYWORDS

Rocq; Coq; theorem proving; proof assistant; premise selection; agentic system; multi-agentic debate

ACM Reference Format:
Andrei Kozyrev , Nikita Khramov , Gleb Solovev , and Anton Podkopaev . 2026. RocqStar: Leveraging Similarity-driven Retrieval and Agentic Systems for Rocq generation. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 9 pages. https://doi.org/10.65109/ZQXK7747

1 INTRODUCTION

In recent years, the advent of Generative Artificial Intelligence (AI) has accelerated the process of developing new software. However, there are studies [23] showing that users who use AI assistants tend to introduce more bugs and vulnerabilities into their code, compared to those who write code on their own. Formal software verification could help mitigate the issue of bugs and security flaws, as it ensures that the software operates correctly and reliably in compliance with the given specification. Under the assumption of a well-formed specification, formal verification provides strong guaranties and an acceptance criterion for the generated code. Interactive Theorem Prover (ITP) is a software tool that assists the user with the development of formal specifications and proofs. To date, there exist several ITPs, such as Rocq (formerly Coq) [1], Lean [6], Agda [14], Isabelle [21], and others. Rocq is a mature ITP, which has experienced more than 30 years of continuous development and improvement. Rocq has an extensive track record of high-impact projects. For example, Rocq was used to verify the correctness of the CompCert C compiler [17], the only C compiler, in which an extensive study found no bugs [36].

Verifying software has always been a rigorous process requiring significant time and human effort. To mitigate this, a number of approaches have been developed to automate theorem proving in Rocq. These approaches build upon the underlying interactive proof paradigm of Rocq, where proofs are constructed incrementally using so-called *tactics*. Tactics serve as elementary building blocks that allow the user to manipulate the *proof state* — a data structure containing the current goal and its context. Each applied tactic transforms the proof state, reducing the original task into simpler subgoals that can be solved recursively. Most solutions implement tactic-prediction approaches and employ beam search or a similar algorithm to navigate the search space. Tactician [3] is a kNN-based approach, which does similarity-based retrieval of tactics used in similar states. CoqGym [34] and Proverbot9001 [27] use Recurrent Neural Networks (RNNs), Graph2Tac [26] proposed a novel graph-based neural tactic prediction. Thakur et al. [28] and Kozyrev et al. [16] instead build generation pipelines around general-purpose, cloud-hosted LLMs, so that no heavy computations occur on the user's machine. CoqPilot [16], along with that, contributes a benchmarking framework and allows seamless integration of standalone tools into the workflow of Rocq's user.

Many approaches call attention to the importance of premise selection, *i.e.,* retrieving useful context information to advance generation. Yang et al. [35] introduced LeanDojo, a retrieval-augmented prover in Lean that significantly improves over non-retrieval baselines. Thompson et al. [29] present the Rango tool and report state-of-the-art performance on the CoqStoq benchmark, automatically synthesizing complete proofs for 32% of the theorems. The work highlights how strongly the well-formed context contributes to the success of Rango. Moreover, they show that *proof retrieval* is the most performant mechanism for premise selection. The proof retriever selects relevant previously completed proofs from the current project and provides them as references to the model. According to the evaluation, Rango proved 47% more theorems than

*Authors contributed equally to this work.

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus. © 2026 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). https://doi.org/10.65109/ZQXK7747

the variant without a proof retriever. However, their retrieval mechanism assumes that two textually similar statements have proofs relevant to each other. In this work, we demonstrate that this assumption oversimplifies the relationship between statements and proofs and introduce a novel embedding model for Rocq statements. It is trained to predict the similarity between their proofs and achieves up to a 28% relative improvement on the evaluation set.

Another promising direction in generative theorem proving that we have identified is Agentic Systems. Research by Kozyrev et al. [16] shows that current Rocq generation methods mostly struggle with complex reasoning tasks. Approaches that perform proof search on top of a tactic generator slow down dramatically as theorem complexity grows, since searching for longer proofs becomes exponentially harder due to the explosion of the underlying search space [31]. Other neural methods, which apply LLMs, suffer from the same problem due to the inability of the model to handle complex reasoning tasks [12]. Agentic systems are known to address these problems; however, to our knowledge, there were close to no attempts to build an autonomous agentic system for an ITP. We build an extensive Model Context Protocol (MCP) server for Rocq and implement an autonomous Agentic System over it, utilizing various problem-specific solutions, such as multi-agent debate. We conduct an evaluation and show that our agentic system strongly outperforms all other previously benchmarked solutions in the CoqPilot's work, raising the ratio of successfully proven theorems from 51% to 60%.

1.1 Contributions

The main contributions of this paper are the following.

RocqStar proof retriever We propose a novel approach for premise selection in Rocq. Rocq suffers from the data-scarcity problem that is common to most ITPs. Aggregating the largest publicly available repositories, one could expect to collect roughly 300 million tokens of Rocq, and about the same for Lean. In contrast, open-source Python corpora easily exceed 100 billion tokens. To tackle this issue we contribute a convenient standalone tool *BigRocq* to extract additional data from Rocq code, utilizing the nature of Rocq's system and the intermediate states of the proof. BigRocq bridges the gap between Automated Generation and Rocq's ecosystem. Using BigRocq, we mine a dataset of 76,524 statements with corresponding proofs from 4 big projects and train a self-attentive embedder model, which learns to predict how close the proofs of given statements will be. In addition, we provide a pipeline to reproduce such embeddings for an arbitrary project, which offers even better results. We integrate the solution as a new retrieval approach for selecting context theorems in CoqPilot and evaluate it using CoqPilot's benchmarking infrastructure. Compared to the baseline text similarity-based ranker, we show an improvement of 28% on the evaluation set. The BigRocq tool, the training dataset, and the code for training the embedder model are available at https://github.com/JetBrains-Research/rocqstar-rag. The embedder model's checkpoint is available at https://huggingface.co/JetBrains-Research/rocq-language-theorem-embeddings.

RocqStar agentic system Addressing the lack of research on applying agentic systems to ITPs, we build an autonomous system for generating Rocq proofs. A custom MCP server built on top of `coq-lsp` [7] handles the interaction with Rocq; its source code is available at https://github.com/JetBrains-Research/rocqstar-agentic-system. Our approach follows a structured process consisting of *planning*, *execution*, and *reflection.* An ablation study shows that while naive planning has limited impact, effective planning based on the Multi-Agent Debate (MAD) framework plays a crucial role. Specifically, it yields a 20% relative improvement in the overall proof success rate and nearly doubles the success rate on complex theorems with longer reference proofs (33% vs. 17%). Additionally, we demonstrate the benefits of the reflection mechanism, which improves the overall proof success rate from 48% to 66% and more than quadruples success on complex theorems; see § 4.3 for details. The evaluation results show that the RocqStar system solves up to 60% of the theorems from the CoqPilot dataset. It is implemented using Koog[1], an open-source JetBrains framework that offers a type-safe Kotlin DSL for building AI agents with structured workflows and tool interaction. The source code of the agent is available at https://github.com/JetBrains-Research/rocqstar-agentic-system.

The remainder of the paper is organized as follows. § 2 describes our Similarity-Driven Retrieval mechanism. § 3 introduces the agentic system. § 4 presents an evaluation of the retrieval component (§ 4.1), the agent (§ 4.2), and an ablation study of the agentic system (§ 4.3). We describe the related work in § 5 and conclude in § 6.

2 SIMILARITY-DRIVEN RETRIEVAL

A known problem in Retrieval Augmented Generation (RAG), applied to the domain of Interactive Theorem Proving (ITP), is *premise selection* [10, 30]. Premise selection is the task of retrieving relevant facts from a given knowledge base to provide the model with the necessary context to advance the proof. However, Huang et al. [9] and Xu et al. [32] highlight that this context must be carefully curated, as the inclusion of irrelevant information degrades model performance.

We distinguish two ways of doing premise selection in Rocq. *Hint selection* — given a context C and a tactic with an unknown positional argument, e.g. `apply _`, the task is to yield potential candidates for the argument. *Proof selection*, in turn, given theorem statement S, focuses on choosing other statements with their respective proofs, so that their presence in the context of the generation request would help the model with the generation of the proof for statement S. Since the approach of applying general purpose models to proof generation is relatively new, most of the works [2, 13, 29, 35] on premise selection in Rocq and other ITPs focused on hint selection. However, Thompson et al. [29] and Kozyrev et al. [16] show that even a baseline proof selection significantly boosts the model's capabilities and is stronger than hint selection. The baseline proof selection presented in both works [16, 29], given the target statement s_* and a database of already proven theorems $[s_0, p_0], \ldots, [s_n, p_n]$ (where s_i denotes a theorem statement and p_i its respective proof), chooses theorems, statements of which have the maximum similarity to the target one. Similarity is defined by

[1] Koog https://docs.koog.ai

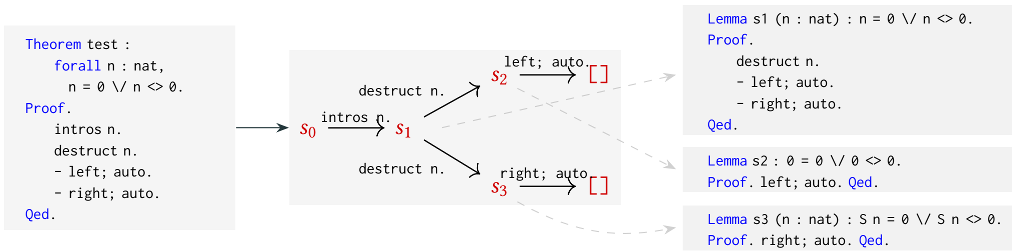

Figure 1: Processing theorems into trees; s_i denotes a state

the BM-25 information retrieval technique [25] or the Jaccard similarity index. That results in packing the generator's context with theorems, based on how similar their statements are syntactically.

We propose a retrieval mechanism that improves the performance of the generator compared to the described baseline. During the generation of the target proof for the statement S we generally assume that the model benefits more from seeing similar proofs to the one that it needs to generate, rather than from seeing similar statements with proofs dissimilar from the target one. Evaluation of our retriever in § 4.1 supports this supposition. One might assume that if statements s_* and s_i are similar, their respective proofs p_* and p_i are similar as well:

$$\text{similarity}(s_*, s_i) \implies \text{similarity}(p_*, p_i) \tag{1}$$

However, we show that this implication often **does not** hold. The heuristic of retrieving similar statements produces decent baseline results, but fails in complex cases, leaving room for improvement. We design our retrieval method to guide context selection based on the similarity of the proofs and show its practicality.

Let us define the proof similarity D_L as the Levenshtein edit distance computed over lists of tactics. Insertions and deletions of tactics have a unit cost, as in the standard Levenshtein formulation. The substitution cost between two tactics is proportional to the Levenshtein distance between their string representations. The resulting distance is normalized by the maximum proof length.

$$p_i = \left[tac_{i_0}, \ldots, tac_{i_m}\right], \quad l_i = |s_i|, \quad D_L(p_i, p_j) = \frac{\text{Lev}(p_i, p_j)}{\max(l_i, l_j)}$$

We conduct the following experiment to examine whether the relation in Equation (1) holds in practice. Considering 1,855,701 pairs of theorems from the IMM project[2], corresponding to all unordered pairs among 1,927 theorems, we compute correlations between statement similarities and respective proof similarities. In summary, BM25-based statement similarity shows a weak negative relationship with the Levenshtein-based proof distance (Pearson $r = -0.154$, Spearman $\rho = -0.171$). The code to reproduce these experiments could be found in the *RocqStar-retriever* repository[3].

To assess the issue of ineffective proof selection, we try to find a function $f(s_i, s_j)$ that correlates with the defined proof distance stronger than statement similarity does. In this work, we introduce a neural method that learns vector embeddings for Rocq theorem statements, training them so that the distance between any two vectors mirrors the similarity between the respective theorem's proofs.

[2]IMM https://github.com/weakmemory/imm
[3]RocqStar retriever: https://github.com/JetBrains-Research/rocqstar-rag/tree/main/experiments

2.1 Dataset mining

Along with other ITPs, Rocq struggles with data scarcity. To address this issue, we mine additional data from the Rocq code. We utilize Rocq system's functionality, preprocess theorems, and transform sequential proof structures into trees. Fig. 1 illustrates this transformation process. Since every node in such a tree is a valid state, we can automatically construct a proof for it by recursively following its subtree edges. Extracting all intermediate statements together with their proofs allows us to expand any given Rocq theorem dataset by a factor roughly proportional to the average proof length. In practice, the observed expansion is also affected by limitations of our extraction procedure. In our case, this resulted in an approximately fourfold increase in dataset size. Kozyrev et al. [15, Appendix A] provides the format of the dataset and further details.

We call the proposed tool *BigRocq* and make it publicly available as a standalone component of our system. The idea of mining additional training data from the intermediate states of the ITP is not new; Kogkalidis et al. [13] conducted analogous research for the Agda [14] language. Similar research for Rocq also takes place [26, 34]; however, some of those works are highly dependent on the deprecated ways of communication with Rocq's compiler [34] and do not support up-to-date versions of Rocq. In contrast, others implement similar ideas as a part of the training pipeline and do not allow for seamless reuse. Using BigRocq, we mine a total of 76,524 statements, collected from 344 files from 4 big Rocq projects. These projects are CompCert, IMM, Promising2Imm, and XMM. CompCert contributes large and heterogeneous proofs, while the remaining projects focus on weak-memory reasoning, which exhibits diverse proof styles across multiple abstraction levels. To avoid data leakage, we exclude theorems belonging to the evaluation datasets from training and only mine subgoals when augmenting a project.

2.2 Modeling

In our work, we formulate the problem as a self-supervised contrastive representation learning task and train a self-attentive embedder model [20]. Given a dataset $\mathcal{T} = (s_i, p_i)$, where s_i is a Rocq statement, p_i is its corresponding proof, and $\mathcal{S}$ denotes the set of all statements appearing in $\mathcal{T}$, together with a similarity function $f(\text{proof}_i, \text{proof}_j)$ defined between two proofs, we aim to learn a function

$$r : \mathcal{S} \times \mathcal{S} \to \mathbb{R},$$

which takes two statements $s_i, s_j \in \mathcal{S}$ as inputs and outputs a score approximating the similarity of their respective proofs p_i, p_j. In other words, $r(s_i, s_j) \approx f(p_i, p_j)$. This formulation allows the ranker r to assign scores to candidate statements relative to a target statement, thereby guiding retrieval towards those whose proofs are most likely to be useful.

In § 4, we evaluate the performance of the proposed model in the following task. Given a target statement s_* and a set of proven theorems $\mathcal{T}$, we aim to select k premises from $\mathcal{T}$ to be used as context for generating a proof of s_*. We take the k most relevant theorems, according to the ranker r.

$$\mathrm{Top}_k(r, s_*) = \underset{(s_i, p_i) \in \mathcal{T}}{\arg\mathrm{top}_k}\, r(s_i, s_*)$$

We say that statement s_* is solved with the use of the ranker r if the generator g produces a valid proof for s_* given the premises selected by r.

$$\mathrm{Solve}(s_*, r, g) = \begin{cases} 1, & g\big(\mathrm{Top}_k(r, s_*),\ s_*\big) \text{ is a valid proof,} \\ 0, & \text{otherwise.} \end{cases}$$

Finally, the quality of the ranker is estimated by the number of theorems in the evaluation set that can be solved using r in combination with a given generator g.

One of the difficulties we encountered during training is a U-shaped distribution of proof distances over randomly sampled theorem pairs. Specifically, when plotting the frequency of theorem pairs against their proof distance, most pairs cluster either at very small distances, corresponding to short, highly similar proofs, or at very large distances, corresponding to largely unrelated proofs. As a result, pairs with intermediate distances are underrepresented, creating a gap that hinders effective training. To mitigate this, we define a modified proof distance that combines the previously introduced in § 2 distance measures with an additional similarity term and injected noise for robustness:

$$\mathrm{proof\_distance}(p_i, p_j) = \alpha\, D_L(p_i, p_j) + (1 - \alpha)\, D_J(p_i, p_j) + \gamma$$

$$D_J(p_i, p_j) = 1 - \frac{|p_i \cap p_j|}{|p_i \cup p_j|}$$

The coefficient $\alpha = 0.7$ was chosen heuristically based on the distribution plot and yielded the best performance in experiments. The noise γ is taken from $\mathcal{U}(-1e-3, +1e-3)$.

As we have already explained in § 2, statement similarity is a poor choice of r, as it shows a low correlation with the target function proof_distance. However, it still provides a strong baseline: in practice, similar theorems occasionally have similar proofs. Accordingly, our approach builds upon statement encoders, fine-tuning them to better align with the underlying proof structure. We fine-tune Microsoft's 108-million-parameter encoder CodeBert [8], originally pretrained on a combined corpus of programming and natural language texts. We also experimented with `gte-modernbert-base`[4] as the base model, but it did not yield notable improvements. As shown in § 4.1, the encoder without post-training performs on par with the Jaccard-similarity baseline, indicating that the unadapted model relies primarily on surface-level syntactic similarity rather than proof-related semantics. Our goal, therefore, is to adapt the encoder to capture this deeper semantic relation.

To achieve this, we train the model using the InfoNCE [22] loss. In our setting, the distribution of proof distances is imbalanced even after normalization. InfoNCE naturally handles this case by contrasting a limited number of positives against a set of negatives, ensuring that informative gradients are maintained. In particular, given a statement s, during dataset preprocessing we compute distances to other samples. We then mark a pair as positive if the distance between their proofs is less than a threshold τ_{pos}, and mark it as negative if it is greater than τ_{neg}. Given the hyperparameter k_{neg} and sets of positive and negative pairs P_s^+ and P_s^-, we compute a per-statement loss term $\mathcal{L}_s$ as follows:

$$\mathcal{L}_s = -\log \frac{\exp\big(\cos(z_s, z_p)/T\big)}{\exp\big(\cos(z_s, z_p)/T\big) + \sum_{j=1}^{k_{\mathrm{neg}}} \exp\big(\cos(z_s, z_{n_j})/T\big)}$$

where cos is a cosine similarity between ℓ_2-normalized embeddings of statements, and $p \in P_s^+$, $n_j \in P_s^-$. On average, we observed smoother convergence for higher values of k_{neg}, which is consistent with findings by Chen et al. [5]. However, due to hardware limitations, we selected $k_{\mathrm{neg}} = 100$ as a practical trade-off between convergence stability and computational cost.

Despite the adjustment of proof_distance$(\cdot)$, during training we experienced the problem of the model converging too quickly on *"easy"* negatives—pairs, whose proofs (and typically their statements) are already far apart in the raw distance space. To keep informative gradients flowing, we add *hard negative* pairs; with some probability we treat a pair of statements as negative if $\tau_{\mathrm{hardneg}} \leqslant \mathrm{sim}(\mathrm{proof}_a, \mathrm{proof}_b) \leqslant \tau_{\mathrm{neg}}$. Introduction of negative samples helped to stabilize the training process; we have observed a less steep training curve and better generalization overall. Kozyrev et al. [15, Appendix B] lists other training hyperparameters.

3 AGENTIC SYSTEM

Agent-based approaches are broadly used in code generation and repair tasks [4, 33, 37]. Despite a large number of autonomous and semi-autonomous coding agents, they are not widely used in formal proofs generation and are not tailored to the Rocq specifics. To address this, we have implemented a RocqStar agentic system.

To allow interaction between the agent and Rocq's system, we develop a REST API server that provides a set of tools that are useful during the execution. We apply our domain knowledge and construct these tools to bring an agent-driven proving process as close as possible to a human-driven one. Examples of allowed function calls include checking validity of proofs, retrieving the valid prefix of given proof, gathering additional information about available entities in the context, and interacting with the context via performing commands like `Print ?a` to identify the type of an argument or `Search ?exp` to search for defined terms by a pattern. Kozyrev et al. [15, Appendix C] describes the toolset in detail. The interaction with Rocq's system is carried out through its language server, `coq-lsp` [7]. To conform with a commonly used Model Context Protocol (MCP) and allow seamless agent interaction with the environment through tools, we implement an MCP server that

[4]gte-modernbert-base: https://huggingface.co/Alibaba-NLP/gte-modernbert-base

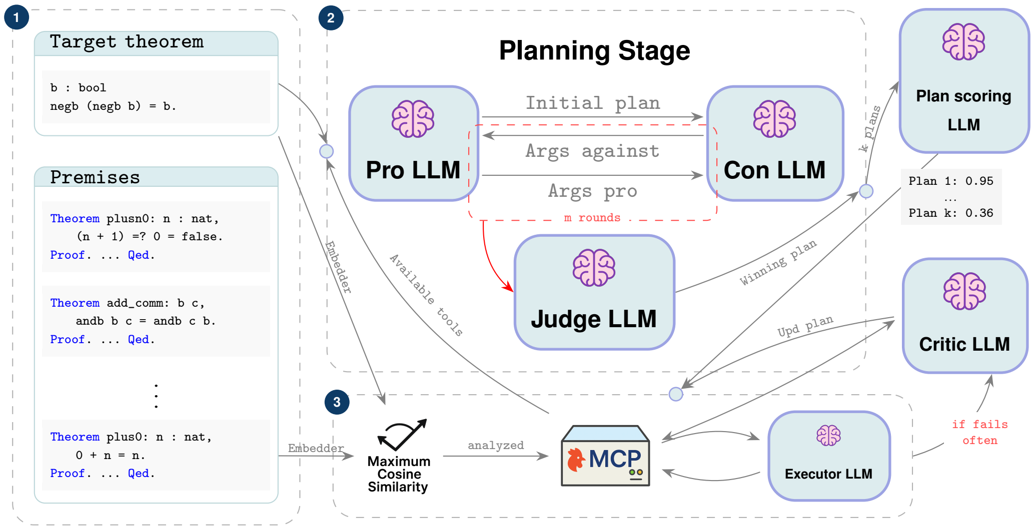

Figure 2: Agentic pipeline with RocqStar retriever

wraps the REST API server. Among the provided tools, the most important is the proof-checking tool. It not only verifies whether a proof is valid but, in case of an error, returns detailed diagnostic information: the error message, its exact location, the valid prefix preceding the error, and the remaining goals after that prefix. This functionality allows the agent to maintain awareness of the current proof state and leverage partial proof progress.

3.1 Agent Logic

The input to the agent is presented as a target theorem without a proof and a file where it was declared, see box 1 of Fig. 2. Agent's pipeline is logically split into two main stages: *planning* and *execution*. In the planning phase, multiple language models rigorously work out the strategy for the further implementation. During execution agents follow the plan aiming to generate the correct proof.

Planning Stage We employ the idea of multi-agent debates [19] to generate a strategy for proving the given theorem. Specifically, two LLMs engage in a discussion: one proposes and defends an initial plan (*pro* LLM), while the other critiques it (*con* LLM); see box 2 in Fig. 2. After several debate rounds, the entire message history is passed to a *judge* LLM, which determines the winner and produces the final plan. By repeating this procedure, we generate k candidate strategies. These are then evaluated by a *plan scoring* LLM that assigns each a numerical score (the higher, the better). Finally, the top-l plans are selected and forwarded to the *Execution Stage*; see box 3 of Fig. 2.

Execution Stage For each of the selected plans, we run an *executor* agent that follows it step by step, invoking tools from the provided tool set — proof checker, context–inspection queries, search commands, and others, as atomic actions. Through these tool calls, the agent interacts with the environment via the MCP server. In addition to this iterative execution, we employ a reflection mechanism that monitors the progress of the proof and adjusts the strategy when necessary. We track how many consecutive erroneous proof attempts occur, and once this number exceeds a predefined threshold (set to five during evaluation), a *critic* model is called to assess the current proof state and identify deviations from the intended strategy. After that, we retrieve theorems along with their proofs, whose top-level goals are similar to the currently remaining goal, according to the cosine similarity between their RocqStar-ranker embeddings. We prompt the LLM to explain which tactic sequences could be helpful to finish our proof. We gather the generated criticism and send it to the *replanner* LLM to refine the current plan along with similar proofs and their analysis. The replanner is a separate language model that revises the plan based on the critic's feedback and the retrieved examples. The whole message history is sent back to the *executor* agent. During the execution of each plan, n tool calls are allowed. If valid proof is not found after n tool calls, we denote the plan as failed. In this case, we ask a *plan failure summarizer* LLM to generate a short explanation of why the strategy execution failed and what happened during it. Then this summarized explanation is sent to the new execution stage with the next selected plan. This procedure is repeated until the correct proof is found or there are no more strategies to execute. For clarity, Figure Fig. 2 presents a simplified view of the system and does not explicitly depict all auxiliary LLM components used in the pipeline.

4 EVALUATION

To evaluate our approach, partially and as a whole, we use the CoqPilot benchmarking framework. We required a dataset with a large number of human-written theorems and proofs. To compare our solution to existing ones, we decided to re-use the dataset by Kozyrev et al. [16]. It is limited to 300 theorems from the IMM project [24], which was suitable for us in terms of computational and financial costs. The theorems are partitioned into three groups, corresponding to the difficulty level. The length (in tactics) of the human-written reference proof of the theorem estimates its difficulty. The sizes of each group are chosen with respect to the initial distribution of proof lengths in the project. Final group sizes and

Group	⩽ 4			5 − 8			9 − 20		
Ranker	Jaccard	ModernBert	**RocqStar**	Jaccard	ModernBert	**RocqStar**	Jaccard	ModernBert	**RocqStar**
GPT-4o	48% ± 5%	44% ± 4%	**51%** ± 5%	18% ± 4%	21% ± 5%	**25%** ± 3%	11% ± 4%	8% ± 4%	**14%** ± 5%
Claude 3.5	58% ± 5%	57% ± 3%	**61%** ± 4%	28% ± 5%	30% ± 3%	**36%** ± 5%	16% ± 5%	16% ± 5%	**21%** ± 5%

Table 1: Model performance under different ablations across all evaluation sets

Reference proof length	⩽ 4	5–8	9–20	Total
Group size	131	98	71	300
OpenAI GPT-4o	50%	26%	15%	34%
OpenAI o1	66%	31%	8%	41%
Deepseek R1	58%	29%	11%	37%
Claude 3.5 Sonnet	73%	41%	27%	51%
LleMMa 7B	24%	11%	1%	15%
Tactician (`synth`)	45%	23%	10%	29%
Rango	38%	18%	8%	25%
RocqStar Agent	**76%**	**56%**	**38%**	**60%**

Table 2: Different Rocq generation methods via CoqPilot

Reference proof length	⩽ 4	5–8	9–20	Total
Group size	22	16	12	50
Full Agent	**91%**	**56%**	**33%**	**66%**
Agent w/o MAD	86%	44%	17%	56%
Agent w/o Planning	86%	50%	17%	58%
Agent w/o Rocq* retr.	86%	50%	33%	62%
Agent w/o Reflection	73%	44%	8%	48%
Claude 3.5 Sonnet	86%	37%	8%	52%

Table 3: Ablation study of Multi-Agent Debate

length ranges of each group could be found in Table 2. The dataset is further restricted to theorems whose human-written reference proofs contain no more than 20 tactics, following the original setup of the CoqPilot benchmark. From now on, we will refer to the described dataset as the *IMM-300* dataset. For smaller ablation studies we additionally prepared *IMM-50*, a 50-theorem subset of IMM, constructed with the same procedure. No theorems from the dataset were present in the training set of the RocqStar ranker embedding model. Moreover, the training set only contained *partial theorem goals*, *i.e.,* intermediate proof goals arising during proof execution, rather than the original top-level theorem statements. Kozyrev et al. [15, Appendix D] describes the split of both datasets into groups, as well as additional details and limitations. Kozyrev et al. [15, Appendix E] describes the computational and financial resources used for the experiments.

4.1 Retrieval Mechanism

We integrate our retrieval mechanism as a ranker into CoqPilot and evaluate it on the IMM-300 dataset with different models under the hood. To assess its performance, we compare our approach against two baselines: (i) an untrained embedder model (we use `gte-modernbert-base`, with `codebert-base` yielding comparable results), and (ii) a lexical similarity baseline based on the Jaccard index. In the latter, given a target theorem statement s_* and a set of proven theorems $[s_0, p_0], \ldots, [s_n, p_n]$, it ranks the theorems in descending order of $J(s_*, s_i)$, where $J(s_*, s_i)$ is the Jaccard-similarity index. The statement is split into tokens by whitespaces, commas, etc. Jaccard-similarity index is semantically almost the same as the BM-25 metric and produces the same numerical results. For each theorem in the dataset, we take theorems within the same file, sort them using the ranker (Jaccard, ModernBert, or RocqStar, respectively), take the k most relevant ones (k is equal to 7 in our experiments) and send a request to the model to generate the completion. The chosen theorems are being sent as a few-shot prompt. Generation for each theorem is requested 12 times. If the Rocq's system accepts any of the proofs, the theorem is considered solved. The target metric in our evaluation is the ratio of solved theorems. The evaluation results are presented in Table 1. The reported values denote mean success rates, and the ± intervals correspond to the standard deviation across three independent runs.

As can be seen from Table 1, our RocqStar ranker consistently outperforms both the Jaccard baseline and the untrained ModernBert encoder, demonstrating reliable gains across all evaluation groups. Most of the performance increase could be seen in the second group; we interpret these results as follows. For short theorems in the first group, the assumption that similar statements imply similar proofs often holds; therefore, all rankers perform comparably. For complex theorems from the third group, it rarely happens that two theorems have significantly similar proofs, resulting in less advancement space for the model.

4.2 Agentic System

We evaluate our agentic system on the IMM-300 dataset with the goal of solving as many theorems as possible. We use the strongest available Rocq proof–generating models, as measured by pass@12 on the CoqPilot benchmark. In addition, following the common principle of using a critic stronger than the executor [38], we employ a more capable model for the planning and evaluation stages. Specifically, for all parts of the planning stage, we use the Claude 3.5 Sonnet model and perform two rounds of debate between actors. Four plans are generated, and two are chosen for further execution. During execution, 20 tool calls are allowed from the MCP server. Additionally, after five proof-checking calls, the critic model

(Claude 3.7 Sonnet) is invoked and analyzes whether a deviation from the initial plan has occurred. We use Claude 3.5 Sonnet for the execution and re-planning, and Google Gemini Flash 2.0 for other tasks, due to the necessity of a big context. Results of the evaluation are shown in Table 2.

As shown in Table 2, our agentic system outperforms other benchmarked models inside the CoqPilot framework. The strongest model so far was Claude 3.5 Sonnet, which achieves 51% accuracy on the dataset, given 12 retries for each theorem. RocqStar agent achieves 60%, showing vigorous improvement. In terms of financial costs, we estimate a run of an agent on one theorem at 1.3 US dollars, compared to 0.25 US dollars for 12 requests to the pure Claude 3.5 Sonnet in CoqPilot. Along with five language models invocated through the CoqPilot framework, we have compared our solution to other Rocq generation approaches, such as Tactician and Rango. On our IMM-300 dataset both solutions showed a result comparable to CoqPilot with OpenAI GPT-4o as the generator model.

4.3 Ablation study

We conduct an ablation study to analyze the contribution of individual components of the agentic pipeline to the overall success rate. In particular, we investigate the effects of removing (1) the *Multi-Agent Debate (MAD)* layer responsible for iterative plan refinement, (2) the *Planning* stage entirely, (3) the *RocqStar retrieval* module, and (4) the *Reflection* mechanism responsible for forced retrieval, criticism, and replanning. All experiments are performed on the IMM-50 dataset, with all other system components kept unchanged. The results are summarized in Table 3.

Planning Considering that software-verification tasks cannot be solved ad hoc, without explicit planning, we measure how removing the MAD layer and reverting to single-pass planning affects the proportion of successfully proved theorems. We run two versions of the agent: one generates plans via MAD, and the other produces a single plan in one LLM call without further refinement. Additionally, we include a configuration with the *Planning* stage entirely disabled, where the executor immediately attempts to construct a proof without any plan. This comparison shows that an agent without planning performs nearly identically, or slightly worse, than one guided by a poor single-pass plan, suggesting that a suboptimal plan does not provide advantage. In contrast, multi-step planning via MAD yields a consistent improvement across all groups, confirming the importance of structured plan refinement. Kozyrev et al. [15, Appendix F] presents an example of MAD repairing a previously unsuccessful plan.

RocqStar retrieval To further assess the contribution of retrieval to the agentic pipeline, we evaluate the agent with its retrieval component replaced by a Jaccard-based baseline. This substitution results in a small but consistent decrease in performance, indicating that the proposed retrieval mechanism provides a meaningful, though moderate, improvement in the agentic setting. While the agentic architecture mitigates some of the limitations of weaker premise selection, effective retrieval remains an important component that contributes to more stable and reliable proof generation.

Reflection Finally, we disable the reflection mechanism that triggers forced retrieval and replanning after failed attempts. Without reflection, the agent loses its ability to recover from early mistakes, leading to a noticeable degradation of performance, especially on longer and more complex proofs. The result shows that reflection complements planning by enabling recovery from failed reasoning trajectories.

5 RELATED WORK

Many Rocq generation methods improve generation using Retrieval Augmentation. Most of those works solve the hint selection problem [2, 29], described in § 2. Those approaches build proofs tactic by tactic, retrieving relevant lemmas or definitions to use in the next step. The problem of retrieving existing proofs that can help advance generation is barely explored in the literature. CoqPilot [16] and Rango [29] address this problem by augmenting the generator's context with previously proven theorems whose statements are similar to the target theorem being proved. Our work proposes a novel premise selection method and demonstrates improvements over the baselines used in prior work [16, 29].

In our multi-agentic system, we distribute responsibility across multiple agents, each responsible for a simpler subtask within the overall proof generation process. This decomposition, which separates high-level reasoning, planning, and execution, is a common design pattern in agentic systems. For example, Li et al. [18] propose a task force split into Thinker, Solver, Critic, and Debug agents, while Liang et al. [19] introduce a multi-agent debate framework that encourages divergent reasoning in complex tasks. We show that explicitly separating planning from execution is essential for the formal verification pipeline. Theorem-proving demands a clear and high-level picture of the proof before executing any code. Running a multi-agent debate at the planning stage ensures rigorous evaluation of different approaches before interacting with Rocq's system. We produce several plans for further execution. In a manner similar to Islam et al. [11], we assign scores to plans and run them in the order of score decrease. To our knowledge, there have been no substantial attempts to build fully agentic systems for ITPs. The closest related effort is the early proof-of-concept by Yang et al. [35], which explores the use of agents for proof generation in Lean. While this work illustrates the potential of agent-based approaches in this domain, it relies on minimal tooling and does not form a fully agentic system with autonomous planning and iterative execution.

As a user interface, we utilize CoqPilot to integrate into the common Rocq's programmer pipeline. CoqPilot is a VSCode[5] plugin, facilitating access to Rocq generation methods for end-users. Currently, the RocqStar retrieval component is available in CoqPilot as a premise-ranking module. Integration of the full RocqStar agentic system as a proof generator is ongoing.

6 CONCLUSION

We have presented a method to enhance retrieval-augmented generation in Rocq via leveraging neural premise selection using a self-attentive embedder model. We evaluated our proposed solution on a dataset of 300 Rocq theorems with two different generator models under the hood and showed a noticeable improvement of up

[5]VSCode: https://code.visualstudio.com

to 28% relative to the baseline. Our result suggests that proof-aware premise selection considerably improves generation quality, particularly for medium-difficulty theorems, where the gap between statement similarity and proof similarity becomes more significant.

Our work pioneers the use of Agentic Systems applied to Formal Verification. We have implemented an advanced pipeline that includes rigorous planning via multi-agent debate, domain-specific tooling, and an adaptive executor–critic loop that iteratively refines proofs based on partial progress. We conclude that our RocqStar agent shows promising results, surpassing strong baselines and highlighting the applicability of agentic systems in the domain of theorem proving. The ablation study further demonstrates that both multi-agent planning and reflection are key to maintaining stable reasoning and achieving consistent improvements in the proof's success rate.

ACKNOWLEDGMENTS

We thank Ekaterina Verbitskaia, Ivan Kabashnyi, Maksim Rozenberg, and Pavel Guliaev for their valuable comments on this work. We also thank the Dynamic Program Analysis Research team at JetBrains (Nikita Dukin, Saga Rut Sunnevudóttir, and Egor Klimov) for their help in building the tool execution distributions described by Kozyrev et al. [15, Appendix G]. Finally, we thank the Koog team at JetBrains for developing Koog, an open-source framework for building AI agents with an idiomatic, type-safe Kotlin DSL.

REFERENCES

[1] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media. https://doi.org/10.1007/978-3-662-07964-5

[2] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. Tactic learning and proving for the Coq proof assistant. *arXiv preprint arXiv:2003.09140* (2020). https://doi.org/10.29007/wg1q

[3] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. The tactician: A seamless, interactive tactic learner and prover for coq. In *International Conference on Intelligent Computer Mathematics.* Springer, 271–277. https://doi.org/10.1007/978-3-030-53518-6_17

[4] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. arXiv:2403.17134 [cs.SE] https://arxiv.org/abs/2403.17134

[5] Changyou Chen, Jianyi Zhang, Yi Xu, Liqun Chen, Jiali Duan, Yiran Chen, Son Tran, Belinda Zeng, and Trishul Chilimbi. 2022. Why do we need large batch-sizes in contrastive learning? a gradient-bias perspective. *Advances in Neural Information Processing Systems* 35 (2022), 33860–33875.

[6] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. 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.* Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[7] Emilio Jesús Gallego Arias et al. 2022. Visual Studio Code Extension and Language Server Protocol for Coq. https://github.com/ejgallego/coq-lsp

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[9] Yue Huang, Yanbo Wang, Zixiang Xu, Chujie Gao, Siyuan Wu, Jiayi Ye, Xiuying Chen, Pin-Yu Chen, and Xiangliang Zhang. 2025. Breaking Focus: Contextual Distraction Curse in Large Language Models. *ArXiv* abs/2502.01609 (2025). https://api.semanticscholar.org/CorpusID:276107466

[10] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. Deepmath-deep sequence models for premise selection. *Advances in neural information processing systems* 29 (2016).

[11] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403* (2024).

[12] Bowen Jiang, Yangxinyu Xie, Zhuoqun Hao, Xiaomeng Wang, Tanwi Mallick, Weijie J Su, Camillo J Taylor, and Dan Roth. 2024. A peek into token bias: Large language models are not yet genuine reasoners. *arXiv preprint arXiv:2406.11050* (2024).

[13] Konstantinos Kogkalidis, Orestis Melkonian, and Jean-Philippe Bernardy. 2024. Learning structure-aware representations of dependent types. *Advances in Neural Information Processing Systems* 37 (2024), 65095–65118.

[14] Wen Kokke, Jeremy G. Siek, and Philip Wadler. 2020. Programming language foundations in Agda. *Science of Computer Programming* 194 (2020), 102440. https://doi.org/10.1016/j.scico.2020.102440

[15] Andrei Kozyrev, Nikita Khramov, Gleb Solovev, and Anton Podkopaev. 2025. RocqStar: Leveraging Similarity-driven Retrieval and Agentic Systems for Rocq generation. Full paper version with appendices. arXiv:2505.22846 [cs.LG] https://arxiv.org/abs/2505.22846

[16] Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. 2024. CoqPilot, a plugin for LLM-based generation of proofs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 2382–2385. https://doi.org/10.1145/3691620.3695357

[17] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress.*

[18] Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. Codetree: Agent-guided tree search for code generation with large language models. *arXiv preprint arXiv:2411.04329* (2024).

[19] Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2023. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118* (2023).

[20] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. 2017. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130* (2017).

[21] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer. https://doi.org/10.1007/3-540-45949-9_5

[22] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).

[23] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do users write more insecure code with ai assistants?(2022). *arXiv preprint arXiv:2211.03622* (2022).

[24] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–31.

[25] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[26] Jason Rute, Miroslav Olšák, Lasse Blaauwbroek, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, and Vasily Pestun. 2024. Graph2Tac: Learning Hierarchical Representations of Math Concepts in Theorem proving. *arXiv preprint arXiv:2401.02949* (2024). https://doi.org/10.48550/arXiv.2401.02949

[27] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages.* 1–10. https://doi.org/10.1145/3394450.3397466

[28] Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. 2023. A language-agent approach to formal theorem-proving. *arXiv preprint arXiv:2310.04353* (2023). https://doi.org/10.48550/arXiv.2310.04353

[29] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F Ferreira, Sorin Lerner, and Emily First. 2024. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. *arXiv preprint arXiv:2412.14063* (2024).

[30] Josef Urban. 2004. MPTP–motivation, implementation, first experiments. *Journal of Automated Reasoning* 33 (2004), 319–339.

[31] Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Zheng Yuan, Wenwei Zhang, Dahua Lin, and Kai Chen. 2025. InternLM2.5-StepProver: Advancing Automated Theorem Proving via Critic-Guided Search. arXiv:2410.15700 [cs.AI] https://arxiv.org/abs/2410.15700

[32] Xiaohan Xu, Chongyang Tao, Tao Shen, Can Xu, Hongbo Xu, Guodong Long, Jian guang Lou, and Shuai Ma. 2024. Re-Reading Improves Reasoning in Large Language Models. arXiv:2309.06275 [cs.CL] https://arxiv.org/abs/2309.06275

[33] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs.SE] https://arxiv.org/abs/2405.15793

[34] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning*. PMLR, 6984–6994. https://doi.org/10.48550/arXiv.1905.09381

[35] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. 2023. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural*

Information Processing Systems 36 (2023), 21573–21612.
[36] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294. https://doi.org/10.1145/1993498.1993532
[37] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs.SE] https://arxiv.org/abs/2404.05427
[38] Yefan Zhou, Austin Xu, Yilun Zhou, Janvijay Singh, Jiang Gui, and Shafiq Joty. 2025. Variation in Verification: Understanding Verification Dynamics in Large Language Models. arXiv:2509.17995 [cs.CL] https://arxiv.org/abs/2509.17995