

Adaptive Proof Refinement with LLM-Guided Strategy Selection

Minghai Lu

Purdue University
West Lafayette, USA
lu1074@purdue.edu

Songlin Jia

Purdue University
West Lafayette, USA
jia137@purdue.edu

Zhe Zhou*

Purdue University
West Lafayette, USA
zhou956@purdue.edu

Benjamin Delaware

Purdue University
West Lafayette, USA
bendy@purdue.edu

Danning Xie[†]

Meta
Menlo Park, USA
dxie@meta.com

Tianyi Zhang

Purdue University
West Lafayette, USA
tianyi@purdue.edu

Abstract

Formal verification via theorem proving enables rigorous proofs of software correctness, but it is difficult to scale due to the significant manual effort and expertise required. While Large Language Models (LLMs) have shown potential in automating proof generation, they frequently produce incorrect proofs on the first attempt that require iterative refinement to fix. However, most existing approaches employ fixed refinement strategies and cannot dynamically choose an effective strategy, which limits their performance. To overcome this limitation, we introduce ADAPT, a novel proof refinement framework that leverages an LLM-guided decision-maker to dynamically select a suitable refinement strategy according to the state of the proof assistant and available context of an incorrect proof. We evaluate ADAPT on two benchmark suites against five existing methods and find that it significantly outperforms the best baseline on both by proving 11.26% and 18.58% more theorems, respectively. Furthermore, we demonstrate ADAPT's generalizability by evaluating it across six different LLMs. We also conduct ablation studies to measure the contribution of each component and compare the trade-offs of alternative decision-maker designs.

CCS Concepts

• **Software and its engineering** → **Formal software verification**.

Keywords

Software and its engineering → *Software verification; Formal software verification.*

ACM Reference Format:

Minghai Lu, Zhe Zhou, Danning Xie, Songlin Jia, Benjamin Delaware, and Tianyi Zhang. 2026. Adaptive Proof Refinement with LLM-Guided Strategy Selection. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3832783.3834382>

*Corresponding author.

[†]Work done while at Purdue University.



This work is licensed under a Creative Commons Attribution 4.0 International License. ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
<https://doi.org/10.1145/3832783.3834382>

1 Introduction

Interactive theorem provers (ITPs) such as Rocq [46] and Lean [37] are powerful tools for formally reasoning about complex software systems. These tools have been used to mechanically verify deep correctness properties of safety-critical applications, including compilers [29], distributed systems [50], and OS kernels [20, 26]. These successes have come at the cost of significant human effort and expertise, however, requiring developers to interactively guide the proof assistant step-by-step. The proof of correctness for the CompCert verified C compiler, for example, consists of 100,000 lines of proof scripts [29] that took 6 person-years to develop.

Motivated by these costs, recent years have seen increasing interest in using Large Language Models (LLMs) to automate the development of proofs in ITPs [13, 19, 33, 47, 49, 53]. Many existing approaches adopt an iterative refinement paradigm—they use an LLM to generate an initial proof attempt and then incrementally repair that proof [10, 19, 33, 34, 49, 51]. Yet they either rely on a single refinement strategy to repair every proof error [10, 19, 33, 49] or use a set of hard-coded rules to select a repair strategy based solely on error messages [34, 51]. This stands in contrast to how human experts fix broken proofs: rather than relying on a single error message, developers comprehensively analyze the entire context of a failed proof attempt to decide between different repair strategies. For example, they may introduce new helper lemmas when existing ones are insufficient, draw inspiration from similar proofs and reuse relevant lemmas, or iteratively repair proofs based on ITP feedback. This sort of strategic decision-making is particularly important in the ITP setting, where a wrong choice may send a proof down an unproductive path, compounding the mistake by significantly increasing proof effort or eventually leading to outright failure.

To address this limitation in prior work, we propose ADAPT, an adaptive proof refinement framework that emulates the decision-making process of human experts. ADAPT employs an LLM-guided decision-maker to analyze the current state of the proof assistant and available context, then dynamically selects an appropriate strategy. As illustrated in Figure 1, when a proof attempt fails, the decision-maker employs one of three strategies: (1) proposing new helper lemmas, (2) searching for relevant lemmas and proofs to guide the refinement, or (3) regenerating the proof. This iterative process allows ADAPT to flexibly adapt its approach at each step, enabling it to effectively prove complex theorems.

Our evaluation of ADAPT against three state-of-the-art methods, COBBLESTONE [24], PALM [34] and RANGO [47], and two alternative

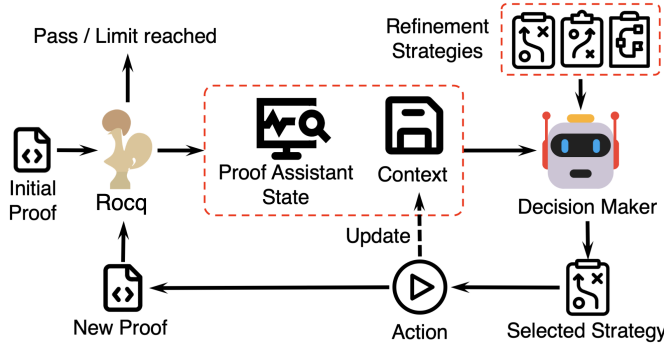


Figure 1: Overview of ADAPT.

refinement designs demonstrates its effectiveness. On the CoqStoQ benchmark [47], ADAPT proves 41.33% of 1,171 theorems, outperforming the strongest baseline by 11.26%. This lead increases to 18.58% on CoqDEV, a new, more challenging benchmark comprised of 1,720 theorems that we have developed. Further experiments confirm the generalizability of ADAPT across six LLMs and, through ablation studies, validate the contribution of each refinement strategy and the LLM-guided decision-maker.

In summary, we make the following contributions:

- (1) We propose ADAPT, an adaptive proof refinement framework that dynamically selects from different refinement strategies based on the proof state and available context.
- (2) We construct CoqDEV, a new, more challenging benchmark mined from real-world Rocq commit histories that model the incremental development process.
- (3) We conduct a comprehensive evaluation of ADAPT on CoqDEV and the most recent CoqStoQ benchmark, demonstrating that ADAPT achieves significant improvements over state-of-the-art techniques.

2 Background and Motivation

To motivate our ideas, Figure 2 presents a self-contained example of a proof development in Rocq [46]—a widely used theorem prover for software verification and the primary ITP considered in this work. We start with a brief overview of the proof process in Rocq and then introduce the problem setting of adaptive proof refinement.

Definitions and Theorems. As is common in Rocq, the example in Figure 2 begins with the **definition** of data structures, functions, and inductive relations. Lines 1–4 define Peano natural numbers (`nat`) with a base constructor (`0`) and an inductive successor constructor (`S`), where `S n` returns the number that is one more than `n`. The file then defines a boolean function `leb` that recursively checks if one number is less than or equal to another, and an inductive relation `le` that provides a formal definition of the same concept.

After the definitions, **theorems** are stated and proven. They start with the keyword `Theorem` or `Lemma`, which are functionally identical but used by convention to distinguish key results from helper proofs. The main theorem, `leb_complete` (line 32), states that if `leb n m` is true, then `n` is less than or equal to `m`. To prove this, two helper lemmas are required: `0_le_n`, which proves that 0

```

1 (* Definitions *)
2 Inductive nat : Set := (* Datatype of natural numbers *)
3   | 0 : nat
4   | S : nat -> nat.
5 Notation "n + 1" := (S n).
6
7 Fixpoint leb (n m : nat) : bool := (* Comparison function *)
8   match n, m with
9   | 0, _ => true
10  | _, 0 => false
11  | n + 1, m + 1 => leb n m
12  end.
13
14 Inductive le : nat -> nat -> Prop := (* Comparison relation *)
15   | le_n (n : nat) : le n n
16   | le_S (n m : nat) (H : le n m) :
17     le n (S m).
18 Notation "n <= m" := (le n m).
19
20 (* Theorems *)
21 Lemma 0_le_n : forall n, 0 <= n.
22 Proof.
23   intros n. induction n.
24   - apply le_n.
25   - apply (le_S 0 n IHn).
26 Qed.
27
28 Lemma n_le_m_Sn_le_Sm :
29   forall n m, n <= m -> (n + 1) <= (m + 1).
30 Proof. (* Proof omitted *) Qed.
31
32 Theorem leb_complete : forall n m,
33   leb n m = true -> n <= m.
34 Proof.
35   intros n.
36   induction n. (* Induction on n *)
37   - intros. (* Base case *)
38     apply 0_le_n.
39   - intros. (* Inductive case *)
40     destruct m. (* Case analysis on m *)
41     + discriminate.
42     + apply n_le_m_Sn_le_Sm. (* Use lemma *)
43       simpl in H. apply IHn in H. apply H.
44 Qed.

```

Figure 2: An example Rocq development that includes a datatype definition (`nat`), a recursive function (`leb`), and an inductive relation (`le`), two lemmas and a theorem.

is less than or equal to any natural number, and `n_le_m_Sn_le_Sm`, which shows that if `n <= m`, then `n + 1 <= m + 1`.

Proof State and Context. Given a theorem statement, Rocq enters an interactive proof mode and displays the current **proof state**. This state includes the **goal** (i.e., the obligation that the user needs to prove) and the **local context** (i.e., any local variables and hypotheses used in that goal). The user applies **tactics** to transform this state by, e.g., simplifying the goal or introducing new hypotheses. Rocq then updates the state or reports an error when tactic application is invalid. The proof of `leb_complete` uses the induction tactic on `n` (line 36) and case analysis (`destruct`) on `m` (line 40). Figure 3a shows the proof state before the tactic on line 38 is applied; the context is above the double line and the goal is below.

Proof Failure. Rocq reports an error for invalid tactic applications. For instance, under the proof state in Figure 3b derived from line 42 of Figure 2, the *erroneous tactic* `apply n_le_m_Sn_le_Sm in H` leads to a proof failure. It attempts to perform forward reasoning by applying the lemma to the hypothesis `H`, but fails with the *error message*: “Unable to apply lemma of type “forall n m : nat, n <= m -> n + 1 <= m + 1” on hypothesis of type “leb (n + 1) (m + 1) = true””.

```

m: nat
H: leb 0 m = true
=====
0 <= m

```

(a) Proof state before line 38 in Figure 2.

```

n, m: nat
IHn: forall m: nat, leb n m = true -> n <= m
m: nat
H: leb (n + 1) (m + 1) = true
=====
n + 1 <= m + 1

```

(b) Proof state before line 42 in Figure 2.

```

n: nat
IHn: forall m: nat, n <= m -> leb n m = true
m: nat
H: n + 1 <= m + 1
=====
leb (n + 1) (m + 1) = true

```

(c) Proof state before line 52 in Figure 4a.

Figure 3: Intermediate proof states. (a) shows the goal for the base case; (b) shows the inductive step requiring a helper lemma; (c) shows a state during the proof of `leb_correct`.

```

45 Theorem leb_correct :
46   forall n m, n <= m -> leb n m = true.
47 Proof.
48   intros n. induction n.          (* Induction on n *)
49   - intros. simpl. reflexivity.    (* Base case *)
50   - intros. destruct m.           (* Inductive case *)
51     + inversion H.
52     + apply Sn_le_Sm_n_le_m in H. (* Use lemma *)
53     simpl. apply IHn in H. apply H.
54 Qed.

```

(a) A Rocq theorem stating the correctness of the `leb` function with respect to the `le` relation, and its proof script.

```

55 Lemma Sn_le_Sm_n_le_m :
56   forall n m, (n + 1) <= (m + 1) -> n <= m.
57 Proof. (* Proof omitted *) Qed.

```

(b) A helper lemma used in the proof of `leb_correct` above.

Figure 4: Example of proof construction by searching and reusing existing proofs.

Project Management and Refinement Strategies. To manage complexity, formal proofs are often organized into a **proof project** with multiple files and imported libraries, similar to a software project. When writing a proof, developers use not only the local context but also a **global context** of definitions and theorems from other files and imported libraries. This leads to several refinement strategies. One common strategy is to search the current project for helpful lemmas. For example, to make progress from the state in Figure 3b, the proven lemma `n_le_m_Sn_le_Sm` is applied to the goal, simplifying it to `n <= m`. When no suitable existing lemma is available, a developer may need to invent a new one, a strategy known as **lemma discovery** [9, 23, 28, 44].

Similar to how developers search and reuse existing code to write new code [41], existing proofs can also serve as a source of inspiration in proof construction. For example, suppose a developer, after proving everything in Figure 2, now wants to prove `leb_correct` (Figure 4a). The statement of `leb_correct` is structurally similar

to `leb_complete`, suggesting a similar proof pattern (induction on `n`, case analysis on `m`). Of course, naively copying the proof of `leb_complete` and reusing it does not work, since the helper lemmas used in `leb_complete` do not directly apply to `leb_correct`. An additional refinement strategy is required: the developer can draw inspiration from them to propose and prove a new, necessary lemma: `Sn_le_Sm_n_le_m` (Figure 4b).

This example highlights the importance of *adaptive proof refinement that integrates multiple proof refinement strategies* according to the available context (i.e., local and global context) as well as the current state of the proof assistant (e.g., partial proof, erroneous tactic, error message, and proof state). A tool that only retrieves existing lemmas would fail because the necessary lemma `Sn_le_Sm_n_le_m` does not exist in the `leb_correct` example. Conversely, an approach focusing only on lemma discovery would have to generate both the new lemmas and proofs from scratch, which is difficult and costly without the guidance of existing lemmas and proofs. The core challenge, therefore, is to create a system that can dynamically assess the available context and current state of the proof assistant to select the most promising strategy, like a human expert.

3 Approach

Given a theorem statement to prove, ADAPT first generates an initial proof and checks it using Rocq. If the proof fails, ADAPT captures the *state of the proof assistant*, including the erroneous tactic, the proof state at the point of failure, the error message from Rocq, and the partial proof preceding the error. It then initializes a working context by retrieving essential information from the global context, including relevant lemmas and similar proofs in the repository.

ADAPT then iteratively refines the failed proof attempt. In each iteration, the decision-maker analyzes the state of the proof assistant and working context to select and apply a strategy, including lemma discovery, context enrichment, and regeneration. If the attempt fails, ADAPT proceeds to the next iteration using the updated state of the proof assistant and working context. This process continues until a successful proof is found or an iteration limit is reached. The rest of this section elaborates on each of these components.

3.1 Initial Proof Generation

ADAPT’s iterative process begins with an initial proof attempt. ADAPT is agnostic to the underlying proof generator, and can freely use, e.g., LLM-based [19, 34, 47, 49], deep learning-based [17, 18, 43, 52], or symbolic approaches [12]. In this work, we use the retrieval-augmented generation method proposed by Lu et al. [34] to generate the initial proof, since it is a recent method with the source code publicly available. Specifically, this approach first identifies candidate theorems and lemmas using TF-IDF and KNN, then reranks them with BM25 [40]. Figure 5 shows the prompt template used to generate the initial proof.¹ The prompt includes the target theorem statement to be proven, all definitions appearing in the statement, and the top K relevant theorems and lemmas. When fewer than K candidates are available, all are included. In this work, we set K to 10 to avoid exceeding the context window limit.

¹Coq was recently renamed to Rocq. Our experiments were conducted before the name change; our prompt templates retain the name “Coq”, which was used in our experiments.

INSTRUCTIONS	
You will be given a theorem statement written in Coq, with related definitions, theorems, and lemmas. Your job is to write a proof. You should first analyze how to prove it based on the definitions, and which theorems and lemmas can be used. Leverage as many theorems and lemmas as possible to facilitate your proof.	
INPUT	
- Theorem Statement:	<theorem>
- Definitions:	<definitions>
- Theorems and Lemmas:	<theorems and lemmas>

Figure 5: Prompt template for initial proof generation.

3.2 Context Initialization

If the initial proof attempt fails, ADAPT collects several pieces of information to help guide its iterative refinement loop. It first captures the state of the proof assistant, which includes the following:

- **Erroneous tactic:** the specific tactic that caused the error.
- **Error message:** Rocq’s explanation of why the tactic failed.
- **Partial proof:** the proof script before the erroneous tactic.
- **Stuck proof state:** the proof state resulting from the execution of the partial proof, which includes the remaining goals to prove and any local variables and hypotheses that are available, as defined in Section 2.

The combination of the erroneous tactic, its error message, and the stuck proof state reveals the immediate cause of failure, while the stuck proof state and the partial proof indicate the overall progress of the current proof. For instance, a simple syntax error in a nearly completed proof could be fixed by regeneration, whereas a failure near the beginning could signal a more fundamental issue with the overall proof strategy. This comprehensive information reflects the feedback a human developer receives and enables the decision-maker to examine different aspects of the failure and select a proper refinement strategy.

In addition to the state of the proof assistant, ADAPT’s refinement loop maintains a working context that guides its decision-maker and refinement strategies. ADAPT initializes this context with the statements of theorems and lemmas retrieved during the initial proof generation, and the proof script of a similar theorem from the global context. Due to the context window limit of the LLM, we only include one proof script as an example. This proof provides a potential proof structure to guide subsequent proof attempts. Following prior work [47], we select a similar theorem using BM25 [40], which compares the textual similarity between the statement of the target theorem and the statements of other theorems in the global context. We select the theorem with the highest similarity score and include its proof, without setting a minimum similarity threshold. This initialized context is then provided to the decision-maker to select the refinement strategy.

3.3 Decision-Maker

ADAPT leverages the reasoning capabilities of LLMs to decide which refinement strategy to take based on both the current project and state of the proof assistant described in Section 3.2. Figure 6 shows the prompt template for decision making. Specifically, ADAPT provides the LLM with the target theorem statement and the definitions that appear in it, the state of the proof assistant, and the working context. The prompt then instructs the LLM to analyze the failure

INSTRUCTIONS	
We are proving a theorem in Coq but get stuck in a proof state, you need to select a refinement strategy to fix it. You will be provided with the theorem, and the diagnostic information, including the stuck proof state, the erroneous tactic that causes the error, the error message from Coq and the partial proof preceding the erroneous tactic. Additional supporting context, including related definitions and lemmas, and a similar theorem with its proof are also provided.	
INPUT	
- Theorem Statement:	<theorem>
- Definitions:	<definitions>
- Stuck Proof State:	<proof state>
- Erroneous Tactic:	<erroneous tactic>
- Error Message:	<error message>
- Partial Proof:	<partial proof>
- Theorems and Lemmas:	<theorems and lemmas>
- Similar Proof:	<similar proof>
OUTPUT	
Analyze the provided information, then choose one of the following strategies:	
1. Strategy: Lemma Discovery	
Description: Propose new lemmas or refine existing ones to help the proof.	
Output Format: 'Lemma Discovery; [name_1, name_2 ...]' . This strategy always proposes new lemmas. Optionally, provide the names of existing lemmas or theorems to refine. Provide an empty list '[]' to only propose new lemmas.	
2. Strategy: Context Enrichment	
Description: Retrieve more relevant definitions, lemmas, theorems, and proofs from the project and imported libraries.	
Output Format: 'Context Enrichment; [keyword_1, keyword_2, ...]' where you list keywords to search for, such as names of definitions or lemmas. Lemmas with the listed keywords in the statements will be retrieved.	
3. Strategy: Regeneration	
Description: Directly regenerate the proof if the current context seems sufficient or the previous proof is unpromising.	
Output Format: 'Regeneration' .	

Figure 6: Prompt template for decision-making.

reasons and assess the sufficiency of the current context for solving the proof goal. Based on the analysis, the LLM is asked to select one of three predefined strategies. We do not claim that these three strategies are complete, as other useful refinement strategies may exist. Our goal is to demonstrate the effectiveness of adaptively selecting from different refinement strategies, rather than to propose all possible strategies.

- (1) **Lemma discovery:** When a proof goal is too complex to be solved directly with the available context, human developers often propose new helper lemmas to decompose the proof into manageable sub-tasks. This process, known as *lemma discovery*, has been proven to be effective when leveraging LLMs to propose new lemmas [9, 49]. Inspired by this, we integrate the lemma discovery strategy to generate lemmas to decompose complex proof goals. As shown in Figure 6, when selecting this strategy, the decision-maker is also prompted to propose a set of existing theorems and lemmas that can be potentially adapted to generate a new lemma.
- (2) **Context enrichment:** This strategy is designed to address failures caused by an insufficient working context by searching for additional definitions, lemmas, theorems, and proofs.

The decision-maker is also prompted to provide a set of keywords to guide the following context enrichment process, such as the names of definitions or lemmas present in the stuck proof state.

- (3) **Regeneration:** This strategy is employed to either repair simple errors, such as a syntax mistake, or to try a new proof approach when the previous proof contains significant errors, such as applying induction on wrong variables or failing to generalize the induction hypothesis correctly.

Alternative decision-maker designs offer different trade-offs in terms of performance and cost. Section 4.7 explores decision-makers based on heuristic rules and machine learning (ML) models. We describe each refinement strategy in the following sections.

3.4 Lemma Discovery Strategy

Lemma discovery takes as input the theorem statement, the stuck proof state, and the working context. The stuck proof state can reveal the specific difficulty encountered by the proof assistant, which helps ADAPT generate lemmas tailored to the specific obstacle. For example, the proof state in Figure 3c suggests that a lemma establishing $(n + 1) \leq (m + 1) \rightarrow n \leq m$ could be helpful, as it bridges the hypotheses H and IHn . Besides proposing new lemmas directly, ADAPT also refines existing lemmas selected by the decision-maker (described in Section 3.3) to maximize the chance of finding useful lemmas, since existing lemmas may inspire the generation of similar lemmas tailored for the stuck proof state. ADAPT applies both mechanisms rather than choosing between the two. We elaborate on these two mechanisms below.

Directly Propose New Lemmas. Proposing new lemmas requires an agent not only to formulate a statement that is both syntactically valid and helpful, but also to generate a correct proof of the statement. To address this, ADAPT employs a two-stage process that decouples the generation of lemma statements from the generation of their proofs.

In the first stage, an LLM is instructed to generate statements of new lemmas. As shown in the prompt in Figure 7, it is provided with existing theorems and lemmas in the working context and stuck proof state, so that it can generate semantically relevant lemma statements, while avoiding redundant lemmas that already exist. These generated statements are then checked by Rocq for syntactic and semantic correctness. Any invalid statements are discarded.

In the second stage, ADAPT attempts to generate a proof for each valid lemma statement, using the same procedure as the initial proof generation described in Section 3.1. The statement of a successfully proven lemma is added to the working context.

Adapt Existing Lemmas. Some existing theorems and lemmas, while not directly applicable, can be adapted to work in the current proof. For example, the existing lemma $n_le_m_Sn_le_Sm$ (Figure 2, line 28) cannot be applied to the proof state in Figure 3c. However, a human developer might observe that its converse could be helpful and propose the new lemma $Sn_le_Sm_n_le_m$ shown in Figure 4b.

To refine a theorem or lemma selected by the decision-maker, ADAPT provides an LLM with its statement and proof. The prompt, as shown in Figure 8, instructs the LLM to analyze how the lemma can be adapted to solve the proof goal. It then directs the model to generate both the refined statement and proof, encouraging it

INSTRUCTIONS

We are proving a theorem in Coq but get stuck in a proof state, your task is to propose some new lemma statements to help solve it. You will be given the theorem statement and proof state, together with related definitions, existing theorem and lemmas. Your lemmas must state new, non-trivial properties. For each proposed lemma statement, explain why it is correct and useful.

INPUT

- Theorem Statement: <theorem>
- Proof State: <proof state>
- Definitions: <definitions>
- Existing Theorems and Lemmas: <theorems and lemmas>

Figure 7: Prompt template for proposing new lemmas.

to reuse the original proof to simplify the proof generation task. Finally, ADAPT validates each refined lemma by executing its statement and proof in Rocq. Only those lemmas with both a well-formed statement and a correct proof are added to the working context.

After updating the working context, ADAPT prompts an LLM to regenerate a proof. The prompt provides the original theorem statement, the updated working context, and the state of the proof assistant from the prior proof failure. It also instructs the LLM to leverage the new lemmas in its proof attempt.

INSTRUCTIONS

We are proving a theorem in Coq but get stuck in a proof state, your task is to refine an existing lemma to help solve it. You will be given the theorem statement to prove, the stuck proof state, related definitions and the specific lemma to refine, including its proof. A set of existing theorems and lemmas are also provided. First analyze how to adapt this given lemma to help solve the proof goal, then produce the refined lemma statement together with its proof. Try to reuse the original proof of the given lemma where possible.

INPUT

- Theorem Statement: <theorem>
- Proof State: <proof state>
- Definitions: <definitions>
- Existing Theorems and Lemmas: <theorems and lemmas>
- Lemma to Refine: <lemma and proof>

Figure 8: Prompt template for adapting existing lemmas.

3.5 Context Enrichment Strategy

When this strategy is selected, ADAPT searches for additional definitions, lemmas, theorems, and proofs from the global context to enrich its working context. Specifically, the decision-maker identifies a set of keywords based on the state of the proof assistant and context, as described in Section 3.3. ADAPT then performs a targeted search by matching these keywords against the statements of available theorems and lemmas. For example, recognizing that the proof state in Figure 3c involves the relation le and function leb , the LLM may suggest these terms as keywords to find more relevant lemmas when the existing ones are deemed insufficient. Similarly, if ADAPT has retrieved the proof of $leb_complete$, it may identify the lemmas $n_le_m_Sn_le_Sm$ and 0_le_n used in its proof, and generate their names as keywords if they are not in the working context. Subsequently, ADAPT employs the same proof generation procedure used after lemma discovery described in Section 3.4, but supplies it with the enriched working context and instructs the LLM to leverage the newly retrieved context.

INSTRUCTIONS	
I'm proving a theorem in Coq but got stuck in a proof state. You will be given the theorem statement to prove and the diagnostic information, including the stuck proof state, the erroneous tactic that causes the error, the error message from Coq and the partial proof preceding the erroneous tactic. Additional context, including related definitions, theorems and lemmas, and a similar theorem with its proof are also provided. First analyze how to prove the theorem based on the provided context, then write a correct proof. You can repair the proof if the error is simple, or generate a new proof if the prior proof is infeasible.	
INPUT	
- Theorem Statement:	<theorem>
- Stuck Proof State:	<proof state>
- Erroneous Tactic:	<erroneous tactic>
- Error Message:	<error message>
- Partial Proof:	<partial proof>
- Definitions:	<definitions>
- Theorems and Lemmas:	<theorems and lemmas>
- Similar Proof:	<similar proof>

Figure 9: Prompt template for regeneration.

This design contrasts with prior work [47], which selects theorems and lemmas by assessing the similarity between their statements and the entire proof state. The intuition is that the entire proof state may contain terms that are not critical for making progress and therefore using the entire proof state in a query can retrieve irrelevant theorems and lemmas or miss important ones. For example, in Figure 3c, the proof state contains common types like `nat` and `bool`. Including them in the query would lead the retrieval system to fetch broadly related but unhelpful lemmas about natural numbers or booleans, potentially missing a lemma `n_le_m_Sn_le_Sm` particularly relevant to the inequality in the current goal. Section 4.6 validates this design choice by comparing ADAPT to *ADAPT_full_state*, a variant of ADAPT that utilizes all terms in the stuck proof state for searching instead of LLM-generated keywords. The result shows that using LLM-generated keywords indeed leads to more theorems to be proved (651 vs. 625).

3.6 Regeneration Strategy

The regeneration strategy can either repair simple errors in the previous proof, or explore a new proof approach when the previous one is deemed infeasible. When this strategy is selected, ADAPT provides an LLM with the target theorem statement, the working context, and the state of the proof assistant from the current proof failure. This strategy prompts the LLM to reason about the previous failure and then generate a new proof (Figure 9).

4 Evaluation

We investigate five research questions in our evaluation:

- **RQ1:** How effective is ADAPT compared to state-of-the-art proof generation tools?
- **RQ2:** How sensitive is ADAPT to the underlying LLMs used?
- **RQ3:** How much does each proof refinement strategy contribute to the overall effectiveness of ADAPT?
- **RQ4:** How does our LLM-guided decision-maker compare to other alternative designs?

- **RQ5:** How does ADAPT's iteration limit impact its performance?

4.1 Benchmarks

We perform experiments using two different sets of benchmarks. The first is CoqStoQ, a state-of-the-art benchmark based on 14 real-world Rocq projects on GitHub [47]. We use its cut-off split, created to mitigate data leakage issues, consisting of 1,171 theorems.

We have also built a new set of benchmarks, called CoqDev, that represents a more realistic and challenging setting for automatic proof generation. The motivation is based on our observation that proof generation tasks from existing benchmarks are typically constructed by simply omitting the proof of the target theorem, leaving any helper lemmas and similar proofs available to the LLM. In practice, however, the relevant helper lemmas are introduced while constructing a proof, and similar proofs may be subsequently introduced after the commit of the proof. In other words, this simple masking-in-the-middle evaluation setting provides the LLM with important information that may not exist when the proof was actually written. Therefore, this setup may not fully reflect the capabilities of proof generation tools in real-world settings.

We constructed CoqDev by mining the commit histories of open-source Rocq projects on GitHub. Specifically, we searched for projects using Rocq as their primary language as of February 20, 2025, and ranked them by creation date, prioritizing newer projects to mitigate potential data leakage. We only included projects that can be built automatically with *Makefile* or *_CoqProject*, resulting in a final set of 14 projects. These projects span a range of verification domains, including logic, data structures, programming languages, and mathematics. We collected all commits of each selected project to get 2,225 commits in total. Next, we compare each commit against its parent to identify any newly introduced or modified theorems.

For each theorem, we masked its proof body as well as the other theorems and lemmas introduced in the commit. In this way, we make sure that LLMs cannot access any helper lemmas or theorems introduced during the construction of the proof. As a result, LLMs need to propose these lemmas or theorems as human developers did when writing the proof. In addition, any theorems and lemmas introduced in future commits are also not available to LLMs. Only the previously committed lemmas and theorems are available to the LLMs, mimicking how proofs are actually developed in practice. On average, 60.34 same-commit lemmas and theorems are masked per task. Moreover, 23.2% of the human-written proofs use at least one declaration introduced in the same commit, indicating that the masked context is relevant to proof construction. To prevent a few large projects from dominating the benchmark and ensure diversity, we randomly sampled at most 200 theorems from each project to form the final benchmark. In the end, we collected 1,720 theorems. The average number of tactics in the human-written proofs is 16.1, and the median is 7.0.

4.2 Comparison Baselines

We compare ADAPT against three state-of-the-art LLM-based Rocq proof generation tools. We select COBBLESTONE [24] and RANGO [47] as they achieved state-of-the-art performance on the most recent CoqStoQ benchmark. We also include PALM [34] which employs

hard-coded rules to select a refinement strategy. Since PALM does not query the LLM during refinement, we build two additional baselines, SELF-REFINE and SELF-REFINE+RAG, that refine proofs with iterative LLM queries. Each baseline is described below.

- **COBBLESTONE** [24] adopts a divide-and-conquer strategy. It prompts an LLM to generate whole proofs, preserves the successful sub-proofs, and queries the LLM to resolve remaining unproven subgoals.
- **RANGO** [47] fine-tunes the DeepSeek-Coder 1.3B [21] for step-by-step proof generation. At each proof step, it retrieves helper lemmas and existing proofs based on their similarity to the current proof state.
- **PALM** [34] employs hard-coded rules to select repair heuristics that fix common errors based on error types. If the heuristics fail, it falls back to a backtracking strategy to incrementally regenerate previous proof steps with CoqHammer [16].
- **SELF-REFINE** [35] adopts a simple iterative repair loop. After the initial proof failure, it repeatedly prompts the LLM to generate a new proof based on the state of the proof assistant from the previous failure, without a working context, further lemma discovery or context enrichment.
- **SELF-REFINE+RAG** enhances SELF-REFINE by initializing the working context before entering the iterative repair loop. In each iteration, it instructs the LLM to explicitly analyze the state of the proof assistant and working context, then regenerate a proof. However, it does not perform further adaptive lemma discovery or context enrichment.

4.3 Experimental Setup

All experiments are conducted on a server with an AMD EPYC 7313 CPU, an NVIDIA A5500 GPU, and 128 GB memory, running 64-bit Ubuntu 22.04 LTS.

To enable a fair comparison with COBBLESTONE and PALM, we allow ADAPT, SELF-REFINE, and SELF-REFINE+RAG to use *CoqHammer*. We first invoke *CoqHammer* on the initial proof state, and once more when a tactic fails. In the latter case, if *CoqHammer* fails to solve the goal, the proof attempt is considered unsuccessful. We have not modified RANGO to use *CoqHammer*, as doing so would require significant changes to its source code.

In all experiments, we set the temperature to 0 to mitigate randomness. In all experiments except the one in RQ2, we use GPT-4o-2024-08-06 [2] as the underlying LLM for ADAPT and the non-RANGO comparison baselines. RANGO uses a finetuned LLM for step-by-step proof generation, and our experiments use the finetuned DeepSeek-Coder model from its original implementation. We choose not to finetune GPT-4o for RANGO due to its high cost. In RQ2, we further experiment with five additional popular LLMs with varying parameter sizes as the backbone of ADAPT. These models include GPT-5.2 [4], GPT-4o-mini [3], Claude-3.7-Sonnet [1], DeepSeek-V3 [32], and Llama-4-Maverick-17B-128E [5]. Due to the high computational and token costs, we conduct the RQ2 experiment on 500 randomly sampled theorems from CoqDEV.

In RQ3, we conduct an ablation study to measure the contribution of ADAPT’s proof refinement strategies. Specifically, we created three variants of ADAPT: (1) *ADAPT_no_lemma*, which disables the lemma discovery strategy; (2) *ADAPT_no_enrich*, which disables the

Table 1: Theorems proven by each approach.

Approach	# of Theorems Proven		Avg. Cost (\$)
	CoqDEV	CoqSTOQ	
COBBLESTONE	527 (30.64%)	435 (37.15%)	0.154
RANGO	481 (27.97%)	352 (30.06%)	-
PALM	522 (30.35%)	415 (35.44%)	0.012
SELF-REFINE	503 (29.24%)	357 (30.49%)	0.051
SELF-REFINE+RAG	549 (31.92%)	410 (35.01%)	0.058
ADAPT	651 (37.85%)	484 (41.33%)	0.085

context enrichment strategy; and (3) *ADAPT_base*, which retains only the regeneration strategy. When a strategy is disabled, we modify the prompt for the decision-making LLM accordingly to prevent its selection. Furthermore, to validate our keywords-based searching within the context enrichment strategy, we implemented a variant called *ADAPT_full_state* that utilizes all terms in the stuck proof state for searching instead of LLM-generated keywords.

In RQ4, we investigate alternative designs of the decision-maker used by ADAPT, experimenting with deep learning-based, rule-based, random, and sequential approaches. Section 4.7 describes each decision-maker in more detail.

In all experiments except the one in RQ5, we set the refinement iteration limit to 3 for ADAPT and the baselines. RQ5 investigates how the limit on refinement iterations impacts ADAPT’s performance.

4.4 RQ1: Overall Effectiveness of ADAPT

Table 1 presents the number and percentage of theorems proven by each approach on both the existing CoqSTOQ and CoqDEV. Overall, ADAPT significantly outperforms all baselines on both benchmarks. On CoqSTOQ, ADAPT proves 11.26%, 16.63%, and 37.50% more theorems than COBBLESTONE, PALM and RANGO, respectively. On CoqDEV, which simulates a more realistic development scenario with limited context, ADAPT’s advantage is maintained. It outperforms the baselines by 23.53%, 24.71%, and 35.34%, respectively. Moreover, it surpasses the strongest baseline, SELF-REFINE+RAG, by 18.58%. These results demonstrate the effectiveness of ADAPT in integrating and dynamically selecting different refinement strategies in proof generation compared to state-of-the-art methods.

We can also compare the sets of theorems uniquely proved by either ADAPT or one of our baselines. On CoqDEV, ADAPT proves 91 (5.29%) theorems that no baseline proves, while 133 (7.73%) theorems are proved by at least one baseline but not by ADAPT. On CoqSTOQ, the corresponding counts are 54 (4.61%) and 61 (5.21%), respectively. These results show that ADAPT is complementary to the baselines.

To better understand the refinement strategies selected by ADAPT, we analyzed the percentage of tasks where ADAPT selected each strategy in each refinement iteration, as shown in Figure 10. The result shows that the LLM decision-maker in ADAPT selects different strategies throughout the refinement loop, instead of adhering to a single fixed approach. This indicates that each strategy contributes to solving different proof challenges.

Cost Analysis. On average, ADAPT costs \$0.085 USD per theorem, due to the multiple LLM calls required during iterative refinement. SELF-REFINE and SELF-REFINE+RAG cost \$0.051 USD and \$0.058

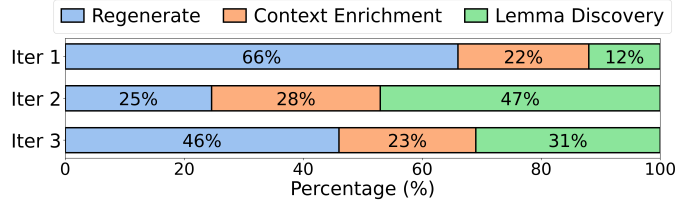


Figure 10: Percentage of strategies selected in each iteration.

USD per theorem, respectively, as they also repeatedly regenerate proofs after failures. COBBLESTONE has the highest cost at \$0.154 USD per theorem. This is expected based on its divide-and-conquer strategy which prompts the LLM to discharge unproven subgoals. PALM only costs \$0.012 USD per theorem, since it only prompts an LLM to generate the initial proof, and relies solely on symbolic methods to repair any proof failures. RANGO incurs the minimal cost, since it runs a fine-tuned local model rather than querying a closed-source API. Overall, these results highlight a trade-off between cost and performance. ADAPT achieves the best performance at a moderate cost.

4.5 RQ2: Sensitivity of ADAPT to LLM Choices

Table 2 presents the number of theorems proven by ADAPT using each underlying LLM. We observe that ADAPT performs the best with GPT-5.2, successfully proving 196 theorems. The default model used in RQ1, GPT-4o, also performs strongly, proving 188 theorems. Its performance remains comparable when using other large models such as Claude-3.7-Sonnet, DeepSeek-V3, and Llama-4-Maverick-17B-128E, proving 185, 179, and 177 theorems, respectively. As expected, ADAPT proves 19.15% fewer theorems when using GPT-4o-mini, which is significantly smaller (estimated to have 8 B parameters) compared to GPT-4o (estimated to have 200 B parameters). This is a reasonable reduction given the limited capability of smaller models. However, ADAPT’s performance with GPT-4o-mini is still comparable to state-of-the-art baselines like PALM, which uses the more powerful GPT-4o model in RQ1.

4.6 RQ3: Contribution of Refinement Strategies

Table 3 presents the performance of these variants compared to the complete ADAPT framework on the CoqDev benchmark. We observe that the full ADAPT proves 10.34% and 4.66% more theorems than *ADAPT_no_lemma* and *ADAPT_no_enrich*, respectively. These results indicate that both lemma discovery and context enrichment contribute to the overall effectiveness of ADAPT, and lemma discovery is particularly critical. Furthermore, *ADAPT_base* performs comparably to SELF-REFINE, which is expected because *ADAPT_base* functions as a standard iterative refinement loop. Finally, replacing generated keywords with the full proof state for searching reduces the success rate by 4.00% and increases token consumption by 7.44%, confirming the efficiency of our keyword-based approach.

4.7 RQ4: Comparing Decision-Maker Designs

While ADAPT leverages the reasoning capabilities of LLMs for adaptive strategy selection based on reflection over the current proof progress and available context, we also investigate alternative decision-making mechanisms to understand the trade-offs.

Table 2: Theorems proven by ADAPT using different LLMs on 500 theorems sampled from CoqDev.

Approach	# of Theorems Proven
GPT-5.2	196 (39.20%)
GPT-4o	188 (37.60%)
GPT-4o-mini	152 (30.40%)
Claude-3.7-Sonnet	185 (37.00%)
DeepSeek-V3	179 (35.80%)
Llama-4-Maverick-17B-128E	177 (35.40%)

Table 3: Effectiveness of refinement strategies on CoqDev.

Variant	# of Theorems Proven
<i>ADAPT_no_lemma</i>	590 (34.30%)
<i>ADAPT_no_enrich</i>	622 (36.16%)
<i>ADAPT_base</i>	519 (30.17%)
<i>ADAPT_full_state</i>	625 (36.34%)
ADAPT	651 (37.85%)

Learning-Based Decision Making. Another mechanism we explored is a learning-based approach, which employs a machine learning (ML) model to predict the strategy a developer would use in a given state of the proof assistant. Training such a model requires a labeled dataset of the state of the proof assistant and the corresponding strategies taken by human developers, which is not available. To address this, we collect a training dataset of 18,628 examples from the human-written proofs in CoqDev, and evaluate the model using ten-fold cross-validation.

To generate each data point in the training dataset, we first simulate the working context at each step of a human-written proof by using the same retrieval method described in Section 3.2. We then extract the following features to characterize the state of the proof assistant:

- (1) The number of unique definitions appearing in the proof goal.
- (2) The number of unique definitions appearing in the local hypotheses.
- (3) The count of definitions that appear in the goal but not in the hypotheses.
- (4) The count of definitions that appear in the goal but not in existing lemmas in the working context.
- (5) The similarity score of the retrieved similar theorem.
- (6) The number of lemmas used in the similar proof that are not present in the working context.

We select features 1 and 2 to reflect proof state complexity. Features 3–5 measure how many definitions in the goal lack supporting lemmas in the working context. Feature 6 assesses if potentially useful information in a similar theorem has not been incorporated.

Given the extracted features, the ML model predicts which refinement strategy to take at the given proof state, which is a typical classification problem. The ground-truth labels are derived from the corresponding human-written tactic at the proof state: if the tactic applies a new helper lemma introduced in the same commit, it is labeled as *Lemma Discovery*; if it applies a lemma that existed

Table 4: Average accuracy and F1 scores for each model.

Model	Acc.(%)	F1
SVM	41	0.48
Decision Tree	25	0.29
Random Forest	41	0.47
XGBoost	71	0.74
DNN	82	0.78

Table 5: Performance and cost comparison of ADAPT with different decision-makers on CoqDev.

Approach	# of Theorems Proven	# Iters	# of Tokens (Input / Output)
ADAPT _{ml}	595 (34.59%)	1.24	9801 / 3157
ADAPT _{rule}	647 (37.62%)	1.37	24244 / 6109
ADAPT _{random}	605 (35.17%)	1.61	12568 / 3476
ADAPT _{seq}	613 (35.64%)	1.58	11376 / 3296
ADAPT	651 (37.85%)	1.35	16348 / 4383

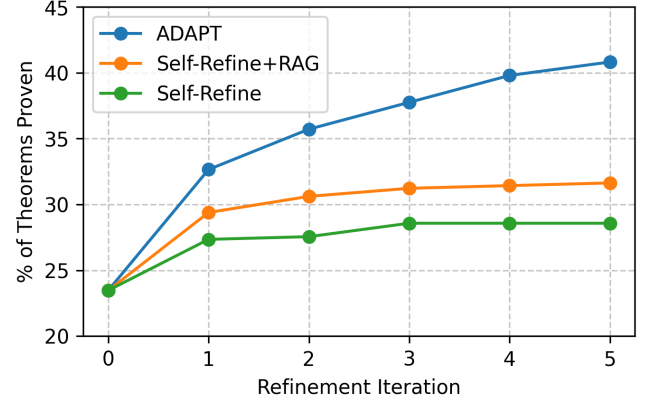
prior to the current commit, it is labeled as *Context Enrichment*; otherwise, it is labeled as *Regeneration*.

We conduct a preliminary experiment using ten-fold cross validation to compare several ML models and select the best one to be compared with other types of decision-makers. We evaluate traditional models, including Support Vector Machine (SVM) [22], Decision Tree [42], Random Forest [11], and XGBoost [14]. In addition, we build a Deep Neural Network (DNN) [31] with three fully-connected layers, using ReLU [38] as the activation function and a dropout rate of 0.2. We train this DNN for 50 epochs using the Adam optimizer [25], with a learning rate of 0.001. The average accuracy and F1 score of each model is shown in Table 4. We ultimately select DNN as it achieves the highest average accuracy and F1 score.

Rule-Based Decision Making. One alternative is a rule-based method, which employs heuristics to select a strategy. First, it checks whether the retrieved similar proof uses lemmas that are missing from the current working context. If so, it prioritizes context enrichment. Next, it analyzes the definitions (e.g., types and functions) in the current proof state. If the proof state involves definitions that are not present in any lemmas in the working context, it selects lemma discovery. The intuition is that formal proofs rely on lemmas to specify the properties of definitions. A definition in the goal without corresponding lemmas indicates an absence of established properties and therefore requires lemma discovery. If neither of these two conditions is met, it defaults to Regeneration.

Random Decision Making. We also implement a baseline decision-maker that randomly selects one of the refinement strategies in each iteration. This is expected to be the weakest baseline, as it does not incorporate any reasoning or contextual information when choosing a strategy.

Sequential Decision Making. We further evaluate ADAPT_{seq}, a deterministic baseline that cycles through strategies in a fixed, cost-ascending order: (1) Regeneration, (2) Context Enrichment, and (3) Lemma Discovery.

**Figure 11: Percentage of theorems proven by refinement iterations.**

Results. Table 5 compares the success rates and average costs of different decision-makers in terms of refinement iterations and token consumption. Overall, the results indicate that the LLM-guided decision-maker achieves the best balance of performance and efficiency. Although the rule-based decision-maker achieves a comparable success rate, it is significantly less efficient, consuming nearly 50% more tokens on average. This inefficiency stems from its simple heuristics, which often unnecessarily select the resource-intensive lemma discovery strategy.

The importance of intelligent strategy selection is further highlighted by the random baseline, which proves 7.07% fewer theorems than the LLM decision-maker and requires, on average, 19.26% more iterations to find a proof. The learning-based decision-maker using DNN performs even worse than the random baseline. We attribute its underperformance to two key challenges: first, our quantitative features may not fully capture the complex semantics of a proof state, and second, the scarcity of training data, especially for lemma discovery, makes the model overly conservative.

4.8 RQ5: Impact of Refinement Iteration Limit

Figure 11 shows the cumulative percentage of theorems successfully proven on CoqDev as the iteration count increases. We observe that while all three approaches benefit from more iterations, the success rates for SELF-REFINE and SELF-REFINE+RAG begin to saturate after 3 attempts. In contrast, ADAPT continues to improve its performance more steadily across all 5 iterations. This suggests that ADAPT’s ability to introduce powerful new lemmas via lemma discovery and context enrichment provides it with more opportunities to make progress on challenging theorems where simpler iterative repair methods have stalled.

4.9 Failure Analysis

To understand the limitations of our approach, we manually analyzed 100 randomly sampled theorems from the CoqDev benchmark that ADAPT failed to prove. Our analysis identified three primary failure modes:

1. Incorrect Strategic Decisions (39%). A primary cause of failure is the decision-maker’s inability to identify the correct strategy.

```
(* Required existing lemma *)
Lemma Ordered_iff_Ordered' t :
  Ordered t <=> Ordered' t.

(* Theorem statement and human-written proof *)
Lemma rotate_right_Ordered_imp v l r :
  Ordered (rotate_right v l r) -> Ordered (Node v l r).
Proof.
  destruct l as [|lv ll lr]; eauto 2.
  rewrite !Ordered_iff_Ordered'.
  ...
```

(a) **Incorrect strategy selection:** ADAPT selects regeneration in all iterations and never retrieves the required existing lemma [7].

```
(* Required lemma statement correctly proposed by Adapt *)
Lemma valt_dec : forall T1 n n1 v1,
  val_type n v1 T1 -> n1 <= n -> val_type n1 v1 T1.

(* Theorem statement and human-written proof *)
Lemma envt_dec : forall n n1 H G,
  env_type n H G -> n1 <= n -> env_type n1 H G.
Proof.
  ...
  eapply valt_dec; eauto.
Qed.
```

(b) **Ineffective strategy execution:** ADAPT correctly proposes the required helper lemma but fails to prove it [6].

```
(* Required existing lemma correctly retrieved by Adapt *)
Lemma const_unfold : forall T (x : T),
  const x = Cons x (const x).

(* Theorem statement *)
Lemma L_as_K : forall n, L n = K n << 0.
Proof.
  (* Human-written proof *)
  destruct n as [| n].
  - apply const_unfold.
  ...

  (* LLM-generated proof *)
  destruct n.
  - rewrite const_unfold.
  ...
```

(c) **Proof generation failure:** the required lemma is available, but ADAPT applies it using an invalid tactic [8].

Figure 12: Representative failure examples from CoqDev.

Specifically, in 25% of cases, the LLM failed to select context enrichment in any refinement iteration, despite the need for existing lemmas. For instance, Figure 12a shows an example where the LLM repeatedly selected regeneration, failing to find the necessary lemma. Similarly, in 14% of cases, it never attempted lemma discovery, though new helper lemmas were required.

2. Ineffective Strategy Execution (32%). Even when the correct strategy is selected, the execution modules may still fail. In 17% of cases, *Context Enrichment* failed to select required lemmas because the LLM generated ineffective search keywords. In 15% of cases, the lemma discovery strategy only proposed trivial lemmas or failed to prove the proposed lemmas. For instance, Figure 12b illustrates an example where the lemma discovery strategy correctly proposed the statement of the required lemma, but failed to prove it.

3. Proof Generation Failures (29%). Despite correct strategic decisions and execution, the LLM may still fail to generate the correct proof script. In 9% of cases, the LLM failed to utilize the theorems or synthesized lemmas, although they were provided in the prompt. The remaining 20% of failures were caused by incorrect tactic applications or syntax errors. For instance, Figure 12c illustrates an example where the correct lemma (`const_unfold`) is provided in the prompt, but the LLM mistakenly tried to apply it to the goal using the `rewrite` tactic instead of the `apply` tactic.

5 Discussion

5.1 Threats to Validity

Internal Validity. The main threat to internal validity arises from the inherent randomness of LLMs. To mitigate this threat, we set the temperature of all evaluated LLMs to 0 for more deterministic outputs. Furthermore, we conducted evaluations across two benchmarks and with six LLMs of different parameter sizes. The consistent results observed across these varied settings help to reduce this threat.

External Validity. The threat to external validity concerns the generalizability of our experimental results. To mitigate this threat, we conducted experiments on two benchmarks that include various real-world projects covering multiple domains. However, while the principles of ADAPT’s approach are applicable to other interactive proof assistants such as Isabelle and Lean, our current implementation and evaluation are confined solely to the Rocq ecosystem. Extending ADAPT to and evaluating it within these other ITPs remains for future work.

Construct Validity. A potential threat to construct validity is that we measure the effectiveness of a decision-maker with the overall success rate, which is an indirect metric. This is because a good strategic choice does not guarantee a successful proof. For example, the decision-maker could select a suitable strategy, such as lemma discovery, but the execution of this strategy may fail, leading to an overall proof failure that does not reflect a poor decision.

5.2 Limitations and Future Work

The framework of ADAPT can be expanded by integrating more refinement strategies, such as proof repair models fine-tuned on ITP feedback, similar to the approach of Baldur [19]. Our decision-maker primarily leverages LLM reasoning, with initial explorations into rule-based and machine learning approaches. Future research could investigate more sophisticated methods. For instance, this could involve creating more nuanced heuristics for rule-based methods or incorporating richer features for the machine learning models. Currently, ADAPT relies solely on LLMs to propose lemma statements, but the quality of these LLM-generated statements can vary; some are trivial, syntactically flawed, or unprovable. Future work could enhance the lemma discovery strategy by developing advanced prompting techniques or integrating symbolic methods.

6 Related Work

Proof Refinement. Recent work in proof refinement has focused on combining a vanilla LLM proof generation model with a standalone repair procedure when errors occur. These approaches typically generate an initial proof attempt with an LLM and then employ various repair strategies that leverage proof error information. For instance, Baldur [19] fine-tunes a second LLM specifically for proof repair, regenerating a new proof from both the incorrect proof and the Rocq error message. SAFE [15] fine-tunes an LLM for both proof generation and refinement. PALM [34] employs hard-coded rules to select bug-fixing heuristics based on error types. If the heuristics fail, it falls back to a backtracking strategy that incrementally reverts proof steps and leverages automated theorem provers such as CoqHammer [16] to regenerate those steps. AutoVerus [51] automates the verification of Rust programs using Verus [30] by refining LLM-generated proofs based on the specific error type. COBLESTONE [24] employs a divide-and-conquer strategy. It samples whole proofs from the LLM and uses additional LLM calls to resolve failed subgoals while preserving successful proof steps. A common limitation of these methods lies in their rule-based decision-making mechanisms. In contrast, ADAPT adopts an LLM-guided decision-maker to dynamically select suitable proof refinement strategies based on the proof failure and related context.

Retrieval-Augmented Proof Generation. Another line of work enhances automated proof generation by integrating retrieval mechanisms for relevant context. A common approach is to retrieve premises like definitions and lemmas. For instance, ReProver [53] fine-tunes a retrieval model to find relevant premises given a proof state. Magnushammer [36] improves the retrieval process by training a reranking model to prioritize the most relevant premises. Furthermore, Rango [47] retrieves both relevant proofs and lemmas via textual similarity to the current proof state, providing richer guidance for the proof search. While these methods focus on enhancing proof generation with retrieval methods, ADAPT integrates multiple strategies and selects from them adaptively. Furthermore, ADAPT leverages pretrained LLMs, whereas these approaches often require significant training or fine-tuning.

Lemma Discovery. Lemma discovery facilitates automated proof generation by proposing helper lemmas aimed at simplifying complex proofs. These techniques range from neuro-symbolic approaches to data-driven methods. For instance, Lemmanaid [9] employs a neuro-symbolic pipeline where a fine-tuned model generates lemma templates with symbolic holes that a symbolic engine then instantiates with concrete variables. Lfind [44] offers a data-driven alternative, framing lemma discovery as a synthesis task that generalizes the proof goal into a sketch and then finds instantiations. These dedicated lemma discovery techniques are complementary to our work. Integrating them into ADAPT's adaptive framework to improve the quality of proposed lemmas remains a promising avenue for future research.

LegoProver [49] uses an LLM to propose and prove new lemmas. The verified lemmas are stored in a library and retrieved through embedding-based search. It also refines existing lemmas to improve their reusability. LemmaNet [55] focuses on lemma discovery for proving verification conditions. It first analyzes the source code

and specifications to synthesize and prove helper lemmas that are relevant to program semantics. It then adapts these lemmas to the verification condition using feedback from the proof assistant. Quarry [54] decomposes a proof by prompting an LLM to propose helper lemmas. It then ranks the proposed lemmas based on a difficulty model, and recursively proves each lemma. While these tools use lemma discovery as their main refinement mechanism, ADAPT selects from multiple refinement strategies dynamically.

Agentic Theorem Proving. COPRA [45] leverages the in-context learning capabilities of LLMs to perform a backtracking search. At each step, it prompts the LLM with the current proof state, previous failures, and retrieved lemmas to generate the next tactic. Failed or non-progressing tactics are recorded and avoided in subsequent attempts. It backtracks to the preceding state after repeated unsuccessful attempts at a proof state. RocqStar [27] employs a multi-agent framework. It first uses multi-agent debate to generate and rank candidate proof plans, and then assigns an executor agent to execute the selected plan. If execution repeatedly fails, a critic model analyzes the current proof state and context, after which a replanner model revises the plan for another execution attempt. AutoRocq [48] represents the current proof as a tree and prompts the LLM to generate one tactic each iteration. Upon failures, it first prompts the LLM to regenerate the tactic, and prompts the LLM to generate a query to search for additional lemmas when an error persists after several regenerations. APOLLO [39] is an agentic framework for repairing Lean proofs. It uses compiler feedback to fix syntax errors, localize failing sub-lemmas, apply automated solvers, and invoke an LLM on the remaining goals. All these systems employ a fixed workflow, while ADAPT employs a decision-maker to dynamically select refinement strategies based on the state of the proof assistant and current working context.

7 Conclusion

In this paper, we propose ADAPT, an adaptive proof refinement framework. ADAPT integrates multiple refinement strategies, including lemma discovery, context enrichment and regeneration. Upon proof failures, ADAPT employs an LLM-based decision-maker to select an appropriate strategy based on the state of the proof assistant and available context. We also introduce CoqDEV, a new benchmark that better simulates more challenging real-world development processes. Our experiments show that ADAPT significantly outperforms state-of-the-art baselines on both CoqDEV and the existing CoqStoq benchmark. Further analysis demonstrates ADAPT's generalizability on six different LLMs, and measures the contribution of its refinement strategies. We also compare the trade-offs of alternative decision-maker designs.

8 Data Availability Statement

A replication artifact is publicly available: <https://doi.org/10.5281/zenodo.21708212>

Acknowledgments

We thank the anonymous reviewers for their valuable suggestions and feedback. This work is supported in part by NSF grants CCF-2340408, CCF-2321680, and ITE-2333736.

References

- [1] 2025. Claude 3.7 Sonnet. <https://www.anthropic.com/news/claude-3-7-sonnet>.
- [2] 2025. GPT-4o. <https://openai.com/index/hello-gpt-4o>.
- [3] 2025. GPT-4o-mini. <https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence>.
- [4] 2025. GPT-5.2. <https://openai.com/index/introducing-gpt-5-2/>.
- [5] 2025. Llama 4 Maverick. <https://ai.meta.com/blog/llama-4-multimodal-intelligence>.
- [6] 2026. Failure due to ineffective strategy execution. https://github.com/TiarkRompf/types-and-proofs/blob/8818bd6e622cb13902f6221cf18e0b21bc255ae6/pub/stlc_rec_indexed_eff.v#L298.
- [7] 2026. Failure due to wrong strategy selection. <https://github.com/Dessertion/coq-bst/blob/87bcd47f57705bbd967a9825f587cf0043188f23/AVL.v#L958>.
- [8] 2026. Proof generation failure. <https://github.com/meithecatte/busycoq/blob/979611edb7364ed99de1d160f69611620c6f7d8c/verify/ShiftOverflowBins.v#L42>.
- [9] Yousef Alhessi, Sólrún Halla Einarssdóttir, George Granberry, Emily First, Moa Johansson, Sorin Lerner, and Nicholas Smallbone. 2025. Lemmanaid: Neuro-Symbolic Lemma Conjecturing. *arXiv preprint arXiv:2504.04942* (2025).
- [10] Chenyang An, Zhibo Chen, Qihao Ye, Emily First, Letian Peng, Jiayun Zhang, Zihan Wang, Sorin Lerner, and Jingbo Shang. 2024. Learn from Failure: Fine-tuning LLMs with Trial-and-Error Data for Intuitionistic Propositional Logic Proving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 776–790. doi:10.18653/v1/2024.acl-long.45
- [11] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (Oct. 2001), 5–32. doi:10.1023/A:1010933404324
- [12] Alan Bundy. 2002. A Critique of Proof Planning. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*. Springer-Verlag, Berlin, Heidelberg, 160–177.
- [13] Jialun Cao, Yaojie Lu, Meiziniu Li, Haoyang Ma, Haokun Li, Mengda He, Cheng Wen, Le Sun, Hongyu Zhang, Shengchao Qin, et al. 2025. From informal to formal-incorporating and evaluating llms on natural language requirements to verifiable formal proofs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 26984–27003.
- [14] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.
- [15] Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misa, Hao Yu, Nan Duan, Peng Cheng, et al. 2025. Automated proof generation for rust code via self-evolution. In *International Conference on Learning Representations*, Vol. 2025. 71891–71918.
- [16] Łukasz Czapka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *J. Autom. Reason.* 61, 1–4 (jun 2018), 423–453. doi:10.1007/s10817-018-9458-4
- [17] Emily First and Yuriy Brun. 2022. Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 749–761. doi:10.1145/3510003.3510138
- [18] Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: semantics-aware proof synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 231 (Nov. 2020), 31 pages. doi:10.1145/3428299
- [19] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (San Francisco, CA, USA) (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1229–1241. doi:10.1145/3611643.3616243
- [20] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 653–669. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu>
- [21] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).
- [22] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. 1998. Support vector machines. *IEEE Intelligent Systems and their applications* 13, 4 (1998), 18–28.
- [23] Moa Johansson. 2019. Lemma discovery for induction: a survey. In *Intelligent Computer Mathematics: 12th International Conference, CICM 2019, Prague, Czech Republic, July 8–12, 2019, Proceedings 12*. Springer, 125–139.
- [24] Saketh Ram Kasibatla, Arpan Agarwal, Yuriy Brun, Sorin Lerner, Talia Ringer, and Emily First. 2026. Cobblestone: A Divide-and-Conquer Approach for Automating Formal Verification. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering (Rio de Janeiro, Brazil) (ICSE '26)*. 13 pages.
- [25] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs.LG]* <https://arxiv.org/abs/1412.6980>
- [26] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.
- [27] Andrei Kozyrev, Nikita Khramov, Gleb Solovov, and Anton Podkopaev. 2026. RocqStar: Leveraging Similarity-driven Retrieval and Agentic Systems for Rocq generation. In *Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems (Paphos, Cyprus) (AAMAS '26)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1256–1264. doi:10.65109/ZQXK7747
- [28] Cole Kurashige, Ruyi Ji, Aditya Giridharan, Mark Barbone, Daniel Noor, Shachar Izhaky, Ranjit Jhala, and Nadia Polikarpova. 2024. CCLemma: e-graph guided lemma discovery for inductive equational proofs. *Proceedings of the ACM on Programming Languages* 8, ICFP (2024), 818–844.
- [29] Daniel Kästner, Ulrich Wünsche, Jörg Barro, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 2018. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS 2018: Embedded Real Time Software and Systems*. SEE. http://xavierleroy.org/publi/erts2018_compcert.pdf
- [30] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (April 2023), 30 pages. doi:10.1145/3586037
- [31] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.
- [32] Aixun Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Cheng-gang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).
- [33] Haoxiong Liu, Jiacheng Sun, Zhenguo Li, and Andrew C Yao. 2025. ProofAug: efficient neural theorem proving via fine-grained proof structure analysis. In *Proceedings of the 42nd International Conference on Machine Learning (Vancouver, Canada) (ICML '25)*. JMLR.org, Article 1583, 19 pages.
- [34] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof Automation with Large Language Models. 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, 1509–1520. doi:10.1145/3691620.3695521
- [35] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. SELF-REFINE: iterative refinement with self-feedback. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (New Orleans, LA, USA) (NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2019, 61 pages.
- [36] Maciej Mikula, Szymon Tworowski, Szymon Antoniak, Bartosz Piotrowski, Qiaochu Jiang, Jin Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. 2024. Magnushammer: A transformer-based approach to premise selection. In *International Conference on Learning Representations*, Vol. 2024. 39326–39350.
- [37] Leonardo de Moura and Sebastian Ullrich. 2021. 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*. Springer-Verlag, Berlin, Heidelberg, 625–635. doi:10.1007/978-3-030-79876-5_37
- [38] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning (Haifa, Israel) (ICML '10)*. Omnipress, Madison, WI, USA, 807–814.
- [39] Azim Osmanov, Farzan Farnia, and Roozbeh Mohit. 2026. Apollo: Automated llm and lean collaboration for advanced formal reasoning. *Advances in Neural Information Processing Systems* 38 (2026), 41599–41633.
- [40] 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.
- [41] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (Bergamo, Italy) (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 191–201. doi:10.1145/2786805.2786855
- [42] S Rasoul Safavian and David Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics* 21, 3 (1991), 660–674.
- [43] Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. 2023. Passport: Improving Automated Formal Verification Using Identifiers. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 12 (June 2023), 30 pages. doi:10.1145/3593374

- [44] Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, and Todd Millstein. 2022. Data-driven lemma synthesis for interactive proofs. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 143 (Oct. 2022), 27 pages. doi:10.1145/3563306
- [45] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. 2024. An In-Context Learning Agent for Formal Theorem-Proving. In *First Conference on Language Modeling*. <https://openreview.net/forum?id=V7HRrxXUhn>
- [46] The Rocq Prover Development Team. 2025. The Rocq Prover, Version 9.1.0. doi:10.5281/zenodo.15149628
- [47] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. 2025. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) (ICSE '25). IEEE Press, 347–359. doi:10.1109/ICSE55347.2025.00161
- [48] Haoxin Tu, Huan Zhao, Yahui Song, Mehtab Zafar, Ruijie Meng, and Abhik Roychoudhury. 2026. Agentic Verification of Software Systems. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE157 (June 2026), 24 pages. doi:10.1145/3808164
- [49] Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, et al. 2024. Lego-prover: Neural theorem proving with growing libraries. In *International Conference on Learning Representations*, Vol. 2024. 31566–31597.
- [50] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) (PLDI '15). Association for Computing Machinery, New York, NY, USA, 357–368. doi:10.1145/2737924.2737958
- [51] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 396 (Oct. 2025), 29 pages. doi:10.1145/3763174
- [52] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning*. PMLR, 6984–6994.
- [53] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: theorem proving with retrieval-augmented language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) (NIPS '23). Curran Associates Inc., Red Hook, NY, USA, Article 944, 40 pages.
- [54] Ning Zhang, Nongyu Di, Zenan Li, Yuan Yao, and Xiaoxing Ma. 2026. Planning to Hammer: Difficulty-Aware Decomposition for Automating Rocq Proofs. *arXiv preprint arXiv:2606.17981* (2026).
- [55] Huan Zhao, Haoxin Tu, Zhengyao Liu, Martin Rinard, and Abhik Roychoudhury. 2026. Lemma Discovery in Agentic Program Verification. *arXiv preprint arXiv:2603.22114* (2026).

Received 2026-03-26; accepted 2026-06-18