Towards Automated Formal Verification of Backend Systems with LLMs

Kangping Xu[1], Yifan Luo[1], Yang Yuan[1,2*], Andrew Chi-Chih Yao[1,2*]

[1*]Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China.
[2]Shanghai Qizhi Institute, Shanghai, China.

*Corresponding author(s). E-mail(s): yuanyang@tsinghua.edu.cn; andrewcyao@tsinghua.edu.cn;
Contributing authors: xkp24@mails.tsinghua.edu.cn; luoyf24@mails.tsinghua.edu.cn;

Abstract

Software testing is essential for ensuring system reliability, yet existing automated testing approaches struggle to match the capabilities of human engineers due to key limitations such as test locality, lack of general reliability, and business logic blindness. In this work, we introduce a framework that transforms Scala backend code into formal Lean representations, automatically generating theorems that specify the intended behavior of APIs and database operations. Our pipeline uses LLM-based provers to verify these theorems: when proved, the logic is guaranteed correct, eliminating the need for testing; when the negation is proved, bugs are confirmed; otherwise, human intervention is required. Evaluation on real-world backend systems demonstrates our method can formally verify over 50% of test requirements, potentially halving testing engineers' workload. At an average cost of $2.19 per API, LLM-based verification proves substantially more cost-effective than manual testing and scales easily through parallel execution. Our results indicate a promising direction for AI-powered software testing that can significantly enhance engineering productivity as models continue to advance. Code is available at https://github.com/xukp20/code-formal-verification. Data is available at https://github.com/xukp20/code-formal-verification-data.

Keywords: Formal Verification, Large Language Model, Proof Automation

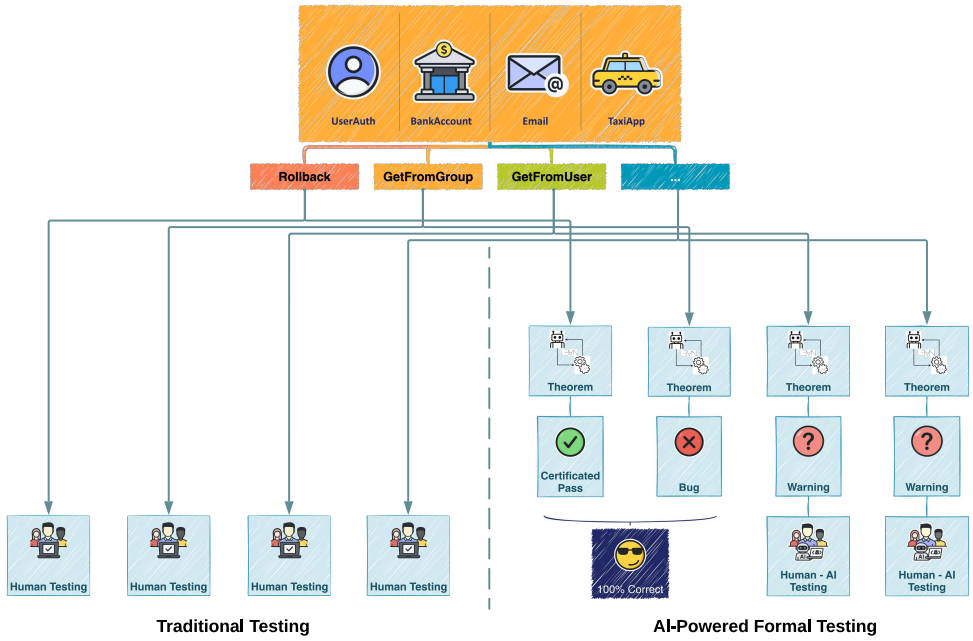

Fig. 1 Comparison between traditional human testing and our framework. Traditional human testing (left) requires extensive labor and relies on manually written test cases. Our framework (right) addresses this limitation by introducing an LLM-based formalizer that automatically translates backend implementations and documentation into Lean formal system, with another LLM-based prover for proving these theorems. Experimental results demonstrate that our approach can formally verify over 50% of API specifications and table properties, which significantly reduces manual testing workload.

1 Introduction

Today, the global software testing industry generates over $55 billion annually [1], reflecting its crucial role in software engineering. In the context of AI-driven development, testing is becoming even more important: as LLMs can now generate massive code rapidly, the role of software engineers is shifting from coding to verifying AI-generated outputs. A natural question arises: can we automate this verification process?

Extensive and intriguing research efforts have been directed toward this goal, from static analyzers and linters to unit testing frameworks [2–7]. The field of program verification has developed numerous automated tools such as ESBMC [8], Z3 [9], and CVC4 [10] for vulnerability detection, while languages like Dafny [11], Verus [12], and Soda [13] integrate verification capabilities directly into programming workflows. Interactive theorem provers like Rocq [14], Isabelle [15], and Lean [16] offer more powerful frameworks with automation tactics such as CoqHammer [17] and TacTok [18],

though they require manual translation of source code into formal languages. These approaches have enabled impressive verification achievements in microprocessors [19], compilers [20], file systems [21], and operating systems [22], though typically requiring substantial expert effort. Recent advancements leverage AI or LLMs to automate parts of the verification pipeline, with techniques like RAG improving auto-formalization accuracy [23, 24] and LLMs showing promise in generating specifications and invariants [25, 26]. However, most current approaches still rely on rule-based components, as seen in FVEL [27] and other systems [28–32]. AI-aided domain-specific applications have emerged for hardware verification [33, 34], smart contracts [35], memory safety [12], and cryptographic protocols [36].

However, to the best of our knowledge, existing approaches still fall short of replicating the work of real-world testing engineers. In particular, they suffer from three main limitations:

1. **Test Locality:** they generally focus on testing individual functions or single files, rather than taking entire systems into consideration.
2. **Lack of General Reliability:** they either require language-dependent and predefined verification patterns, or rely on LLMs for automatic test that reduces the reliability of verification.
3. **Business Logic Blindness:** they often overlook actual business logic, concentrating primarily on aspects like memory management and security vulnerabilities.

Yet, these are precisely the aspects that real-world test engineers care about most. They require methods capable of testing entire systems, ensuring verification accuracy, and aligning well with business logic. To address this gap, we must ask: is there a framework that can simultaneously meet these requirements?

Topos theory [37] serves as a principled foundation to this end—a powerful mathematical framework that decomposes complex structures into well-organized, composable components. While the theory itself is abstract, we use its central insight: modeling business logic as a type system and applying formal reasoning to validate it. Through a functional and compositional design, this approach enables scalable and rigorous testing, even for complex real-world systems.

More specifically, our method is rooted in functional programming [38], primarily utilizing Scala [39] due to its expressive type system. Scala allows us to express all logic through pure functions, thereby enjoying several desirable properties. For instance, any complex function can be decomposed into a composition of simpler ones. Furthermore, the correctness of each function is independent of external environmental factors. This compositional property directly addresses the issue of Test Locality, as it allows us to verify each small function in isolation while still ensuring the correctness of the complete system. Moreover, because Scala runs on the JVM and is interoperable with Java, it can leverage the vast ecosystem of Java libraries and perform nearly everything that Java can do.

However, basic functions alone can only express low-level data transformations and are insufficient to capture the full semantics of business logic. To address this limitation, we incorporate type systems to structurally encode key aspects of the business domain—such as entities, constraints, and process flows – within Scala's functional

programming paradigm. While actual business operations often require reasoning over runtime values, the type system provides a formal scaffold that constrains and guides how such operations are composed and executed.

Building on this foundation, we use the Lean formal proof system [16] to enable automated verification. Since our backend is already structured with a Scala type system, translating it into Lean is straightforward, as they share similar expression styles. We then use LLMs to generate properties the system should satisfy, express them in Lean, and attempt automatic proofs. If a proof succeeds, we can conclude that the code is formally correct. If the proof fails, it may indicate either a bug in the code or that the model lacks the capability to complete the proof. To tell these cases apart, we try proving the negation of the original statement. If the model can prove the negation, it confirms a bug; if not, the failure is likely due to the model's limitations, in which case human intervention is required. Our approach offers a new direction for general verification, as it is language-agnostic, applies to all parts of the system, and enables formal reliability when proofs succeed.

Based on our experimental results, our method can formally verify over 50% of test requirements in real-world backend systems, suggesting that more than half of a testing engineer's workload could be automated. It costs only $2.19 per API on average—lower than manual testing—while providing stronger guarantees through formal proofs. For the verified portion, correctness is mathematically ensured, offering higher assurance than traditional testing. Additionally, our approach detects bugs in faulty systems by generating concrete counterexamples with over 70% success, allowing engineers to focus on a smaller set of remaining issues and accelerating the debugging process. Thanks to the composable structure of our functional programming framework, the method scales efficiently to large systems through parallel LLM execution, maintaining both speed and accuracy.

These contributions establish a practical framework for software-level verification of complete backend systems, overcoming the limitations of previous approaches focused on individual components. By effectively combining LLMs with formal methods at an architectural scale, we demonstrate the potential for AI-enhanced verification. We hope that this will be a starting point for deploying these methods at production scale and improving verification capabilities for complex system behaviors.

2 Methods

2.1 Preliminaries

2.1.1 Functional Programming

Functional programming is a programming paradigm centered around the evaluation of pure mathematical functions, which can be represented as:

$$y = f(x), \quad f : X \to Y$$

Here, the function f takes an input $x \in X$ and deterministically produces an output $y \in Y$. A function f is said to be pure if it adheres strictly to the following properties:

- **Referential transparency:** The output of the function depends solely on its given input and not on any external state or hidden variables. Thus, calling the function with the same input always yields the same output.
- **No side-effects:** Evaluation of the function does not affect the outside world, i.e., it does not modify global state, I/O operations, or mutable data structures.

This paradigm differs significantly from imperative programming languages (e.g., C, Python), which commonly allow and often encourage mutable states, side effects, and external dependencies such as global variables, pointers, or object mutations. Scala, as a hybrid language, effectively supports functional programming by enabling pure function implementations and immutable data structures through libraries like Cats and Cats Effect.

One key advantage of functional programming is its capacity for modularization. Complex computations can be decomposed into smaller, simpler, and easily testable pure functions. These smaller functions can be composed mathematically, greatly simplifying debugging, testing, and formal verification.

Functional languages and their implementations become practically applicable through the introduction of types. From the perspective of type theory, types serve as formal abstractions that classify program constructs and specify constraints on their composition. Types ensure the correctness of computations by enforcing strict contracts on function inputs and outputs. Basic types (primitive types) such as `Int`, `Bool`, `String`, and `Enum` serve as the foundational elements. More advanced types can then be constructed using these primitives, including product types (e.g., tuples or records), sum types, and higher-order types. Such types form robust data structures capable of precisely capturing complex business logic.

Type-driven development in functional programming involves designing applications by first defining precise, domain-specific types, subsequently implementing functions that transform these types. In this paradigm, the types themselves document and constrain the application's logic, resulting in code that is robust, expressive, and self-explanatory.

2.1.2 Monad

While pure functional programming has many theoretical and practical advantages, it inherently avoids side effects, which are essential for real-world applications such as I/O operations, database interactions, or network communications. To reconcile purity with the need for side effects, functional programming employs the concept of a *Monad*—a structure borrowed from category theory that encapsulates computations along with their context or side effects.

Intuitively, a monad can be viewed as a computational context or a "box" that encapsulates potential side effects. It preserves purity by deferring side-effectful computations as first-class values that can be manipulated without actually executing the side effects until explicitly instructed to do so.

Monads have two essential properties:

- **Encapsulation of Side-effects:** Side-effectful operations are encapsulated as pure computations inside monads. The monad itself remains pure until explicitly "activated" by an external interpreter or runtime.
- **Composability:** Monads enable the compositional chaining of operations. Multiple monadic computations can be combined to form larger computations, maintaining the purity and modularity of code.

For instance, in Scala, the library Cats Effect provides an `IO Monad`, facilitating pure functional programming while enabling real-world side effects. Programmers write purely functional code to describe computations within the `IO` context, and side effects are deferred until explicitly executed by the framework. Thus, the program remains referentially transparent and pure, while still interacting effectively with the external environment.

2.1.3 Algebraic Effects and Handlers

While monads work well for managing side effects in functional programming, they sometimes hide where exceptions come from, especially when using multiple APIs together. This creates challenges when trying to verify code formally in systems like Lean. To solve this problem, we draw inspiration from the strategy of `Algebraic Effects` [40–42], which clearly list all possible outcomes of operations, making our verification process more straightforward.

Algebraic effects handle side effects by treating operations like data changes, errors, and external interactions as separate elements. Unlike monads that combine effects with code, algebraic effects separate what an effect does from how it's handled, making code more flexible since different program parts can handle effects in various ways. Several languages have embraced this approach and handled effects as first-class citizens, including Eff [43], Koka [44], and Frank [45].

This design offers key advantages for our formalization: by explicitly defining all possible outcomes (both successes and errors) and maintaining clear traceability of effect origins, we ensure our formal verification can handle every scenario while preventing important exceptions from being hidden when APIs interact with each other.

2.1.4 Backend System Structure

To effectively realize the benefits of functional programming within practical backend systems, we adopt a microservice-oriented architecture, decomposing the backend system into modular, loosely-coupled components, each of which encapsulates a distinct domain.

Within each microservice, we define domain-specific types that serve as foundational abstractions, characterizing the data manipulated by the service. Building upon these types, we establish database schemas representing the persistent state managed by the microservice. Subsequently, we design and implement APIs, which are externally accessible endpoints, providing explicit interaction points for clients and other microservices. Internally, we also define processes—functions encapsulating common business logic that are not directly exposed externally, but rather utilized by the APIs

themselves. All functions handling data manipulation, database interactions, APIs, and internal processes are implemented as pure functional constructs leveraging Scala's monadic structures.

This structured, monadic approach also facilitates formal verification. Specifically, each API function invocation can be systematically decomposed into combinations of smaller, pure functions, including process functions, database table operations, and inter-service API calls. Each individual function, due to its purity and well-defined typing, can be translated directly into Lean.

2.1.5 Lean 4 and Dependent Types

Lean 4 [16] is a functional programming language and proof assistant designed specifically for formal verification, featuring a powerful and expressive type system. Its syntax and functional programming style bear similarities to Scala.

The key advantage of Lean 4 lies in its support for **dependent types**, significantly enhancing its expressiveness and verification capabilities. Dependent types allow types to be parameterized by values, enabling the precise encoding of program invariants directly into the type system. For instance, a dependent type can express properties such as the length of a list being exactly n, or an integer being within certain bounds.

This mechanism tightly couples code with properties, enabling formal verification of program behavior directly through type checking. Lean's dependent type system thus provides a rigorous bridge between Scala's functional implementations and formal proofs, effectively supporting our verification pipeline.

2.2 Pipeline Overview

This section provides a high-level overview of our verification pipeline, which is shown in Fig 2, with detailed explanations following in subsequent sections. The input of the pipeline contains two primary components: the backend codebase, which follows the structure outlined in Section 2.1.4, and a documentation describing the expected API behaviors and their interactions, which serves as the guideline for the verification process. Given these two components, the pipeline verifies whether the implementation matches the specifications through three stages:

1. **Implementation Formalization:** Given the codebase without documentation, LLMs transform the backend codebase into a Lean 4 project, mapping services, APIs, and database schemas into structured formal representations following predefined patterns. Language-specific details are removed and monads used in the programming are replaced following the strategy introduced in Section 2.1.3, resulting in a purely functional implementation in Lean that treats all components as dependent types.
2. **Theorem Generation:** Based on the specifications and formalized implementation, LLMs produce theorems to be proved together with intermediate natural language representations. The models additionally infer and formalize system-wide properties about the tables by viewing all the APIs that interact with the tables as a unified system.

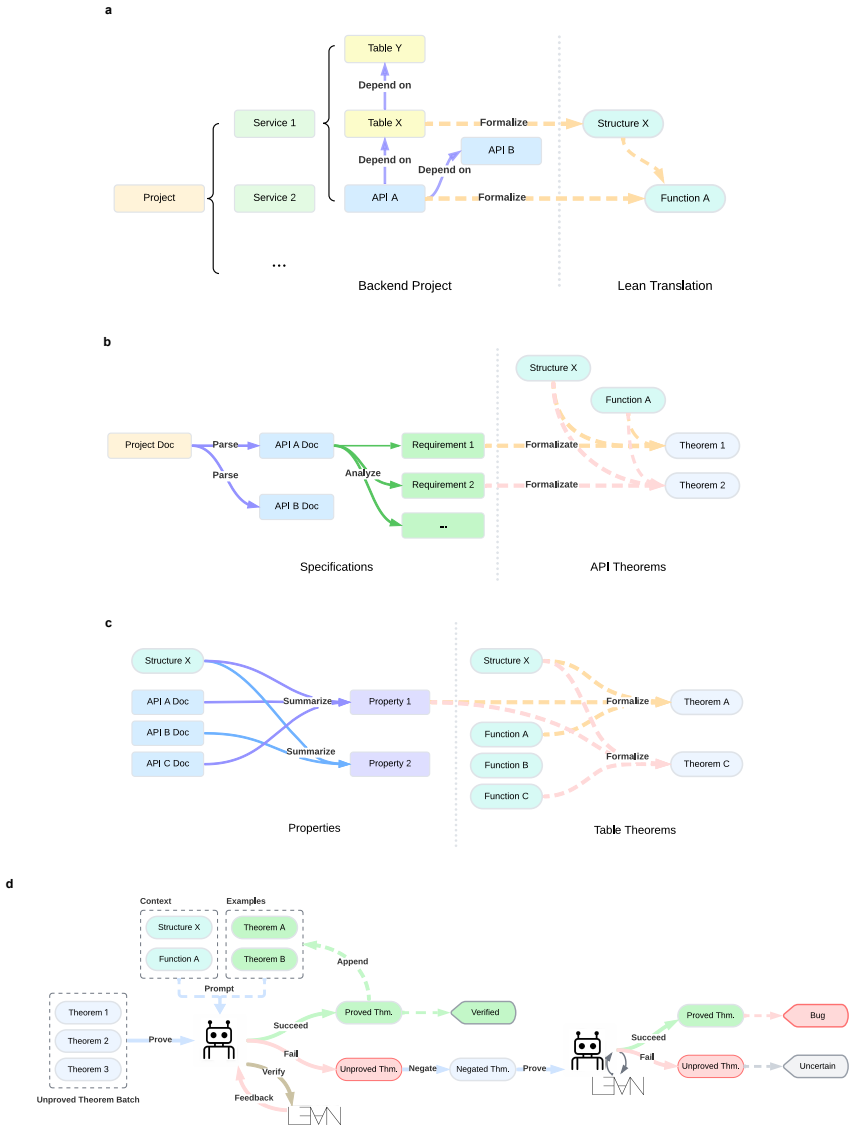

Fig. 2 Overview of the verification pipeline. a, the implementation formalization stage, including the dependency analysis and the formalization process of table structures and API functions. **b**, API theorem generation, including the decomposition of system specifications into API requirements, and the construction of formal API theorems based on the API and its dependencies. **c**, table theorem generation, including the summarization of table properties derived from a subset of related APIs (A and C for property 1 here), and these properties are used to generate formal theorems linked to the corresponding APIs. **d**, proof search stage, where an LLM prover attempts to prove a batch of unverified theorems. If a theorem is successfully proved, its proof is added to the example set. If a proof fails, the prover refines it based on Lean compiler feedback. Theorems that remain unprovable after a set number of refinement rounds are converted into negative theorems that attempt to prove the opposite statement. If a negative theorem is successfully proved, it indicates a bug in the system. Both API theorems and table theorems follow this iterative proving approach.

3. **Proof Search:** The reasoning model conducts proof searches using a few-shot learning approach with Lean compiler-guided refinement. For unprovable theorems, the system automatically tests their negations to identify potential counterexamples, which detects possible bugs.

Through this process, the implementation is fully formalized in Lean 4, and outputs: (1) verified theorems confirming specification compliance, (2) counterexamples revealing specification violations, and (3) unresolved theorems requiring manual inspection.

Illustrative Example. The following sections provide detailed explanations for each stage of the pipeline. To better illustrate our pipeline design, we present concrete examples from a real-world `BankAccount` system backend that supports user management along with withdrawal, deposit, and balance query functionalities along with the introduction to the pipeline. Due to space constraints, all example figures are presented in the Extended Data.

2.3 Implementation Formalization

The first stage aims at translating the backend codebase into a Lean 4 project. APIs and tables are translated into dependent types like pure functions and structures in Lean, while maintaining the logic of the code and dependencies between them. During the formalization, monads are replaced with explicitly defined inductive return types, while tables are passed as input parameters and outputs of API functions to create pure functional formalizations. The formalization stage consists of two phases: **dependency analysis** and **formalization**.

Dependency Analysis. As a pre-task of the formalization, given the API code and table documentation, LLMs are responsible for analyzing dependencies among the following components: Table-to-table dependencies, such as foreign key constraints; API-to-table dependencies, including read and write operations; and API-to-API dependencies, where an API calls other APIs within the same or different services.

Formalization. Following dependency analysis, we formalize the tables and APIs in topological order. For tables, the formalization yields a Lean structure, and the APIs will become functions in Lean.

Note that database operations appear as side effects in API implementations, in the format of SQL query execution. The LLMs demonstrate the ability to interpret these SQL queries and translate them into operations on table structures. To formalize APIs in a purely functional manner, we treat the database state as a function parameter, requiring each function's output to include the updated database state. This approach provides a formal representation of database changes resulting from API execution.

Fig A1 illustrates table formalization using the `Transaction` table. The model transforms the table's YAML description into a formal structure containing row and table definitions. Fig A2 demonstrates API formalization. After formalizing all tables, APIs are formalized in topological order: first `BalanceQuery`, followed by the `Withdrawal` API that depends on it. The left side shows the `Withdrawal` implementation, which the model uses alongside formal table representations and dependent APIs to generate the formal function on the right. The formalization process defines

return types, including success types with values and various error types, formalizes helper functions following the original logic, and then defines the main function. Since `BalanceQuery` depends on both `Account` and `Transaction` tables, their initial states must be included in the `Withdrawal` function inputs, with updated states returned.

Upon completion of this stage, the codebase undergoes a formal translation into an equivalent Lean 4 project.

2.4 Theorem Generation

Having constructed a Lean project that preserves the semantic equivalence of the original codebase, we proceed to formalize the system specifications as verifiable theorems within Lean. There are two kinds of theorems: those related to the API specifications and those related to table properties. They are all constructed through a two-step approach, first generating the natural language description of the theorem and then formalizing it in Lean.

2.4.1 API Theorem Generation

Requirements Generation. The input to this step is the system specifications document written in natural language. We employ LLMs to generate detailed documentation for each API from this document. In our experiments, we provide structured documentation that precisely defines the specifications of each API, which facilitates accurate evaluation during proof search.

Then the documentation of each API is split into a list of input-output requirements, where each entry specifies the conditions on the inputs and table states, along with the expected outputs and updated tables, representing a control flow path in the current API. The functional nature of our API formalization enables isolated verification by treating dependent API returns as preconditions, allowing independent requirement generation for each API, which is a significant advantage for efficiency and scalability.

Theorem Formalization. Then, we formalize the generated requirements as Lean theorems, with each theorem verifying a distinct control flow path of the API. Building upon the formal implementation of the API and its dependent tables and APIs, LLMs follow a structured analysis process to identify theorem components: input parameters as function arguments, conditions as hypotheses, and expected outputs with updated table states as conclusions. The LLMs then construct well-formed theorems that undergo compilation to verify syntactic correctness, with iterative refinement addressing any detected errors.

Fig A3 demonstrates an example of API theorem generation. Beginning with the final line of the `Withdrawal` document parsed from the system specifications, which specifies success conditions and results of the API, the model identifies necessary premises such as positive withdrawal amount, successful `BalanceQuery` execution, and sufficient balance to create a self-contained requirement. This requirement is then translated into a rigorous formal theorem in Lean. Standard prefixes like imports and open commands are omitted. Before the theorem statement, a comment derived from the requirement explains the theorem's meaning. The theorem statement declares

API function input parameters as variables, along with initial states of related tables. All premises from the requirement appear as hypotheses, while the API output and updated table states constitute the conclusion. Currently, the proof is omitted and replaced with a `sorry` placeholder.

2.4.2 Table Theorem Generation

The approach of generating table theorems is similar to that of API theorems, yet we focus on higher-level properties of the tables that are valid in a system-wide manner, by systematically analyzing the API specifications.

Property Summarization. For table properties, we intentionally avoid providing predefined documentation since these properties should be inferable from API specifications. Our approach proceeds as follows: First, we aggregate all APIs interacting with a given table. Next, we analyze their specifications to derive the table's properties. The LLMs generate each property as a natural language statement accompanied by the set of APIs responsible for maintaining it. For instance, read-only APIs must not modify the table.

Theorem Formalization. Each property is formalized as a set of theorems, with one theorem per associated API. These theorems resemble API theorems but have two key differences: they place fewer restrictions on input parameters and concentrate mainly on how tables change rather than on API responses. This approach allows for checking invariants by analyzing how individual API calls modify table states in a single step.

Fig A4 illustrates three properties regarding record quantities in the `Transaction` table. Three APIs interact with this table: the read-only `BalanceQuery` never alters records regardless of its outcome, while `Withdrawal` and `Deposit` add a new transaction record upon success or maintain the current count if unsuccessful. The theorems derived from these properties address all possible scenarios, with two shown as examples. The `BalanceQuery` theorem requires no hypotheses about API inputs or success conditions, asserting that the record count remains unchanged under all circumstances. The `Deposit` theorem, conversely, includes hypotheses ensuring API success, leading to the conclusion that the record count increases by one. Compared to API theorems, these table theorems pay more attention to the table states instead of API outputs, collectively characterizing the properties of the tables in the backend system.

2.5 Proof Search

2.5.1 Proof Strategy

The final stage of our pipeline employs general LLMs with advanced reasoning capabilities (e.g., DeepSeek-R1 [46]) to conduct proof search for theorems concerning APIs and tables. Our approach incorporates two key strategies:

Compiler-Guided Refinement. When proofs fail, we retrieve compiler error messages along with their contextual locations, and also implement a backtracking mechanism to identify the last correct proof step with the unsolved goal, enabling iterative, step-by-step proof refinement until successful verification.

Adaptive Few-Shot Learning. For each batch of theorems, we dynamically select relevant proved theorems based on three levels of similarity: (1) same API/table, (2) same service, and (3) project-wide. This hierarchical retrieval provides increasingly relevant proof examples to guide the model. Based on this, we implement a dual-loop architecture where local theorem refinement coexists with a global retry mechanism—unproved theorems gain increasing proof opportunities as the pool of verified examples expands throughout the project.

Each theorem successfully verified during proof search corresponds to a validated control flow path within the API implementation. Examples of proved theorems can be found in Fig A4 of the table theorem section.

2.5.2 Counterexample Search

For those theorems unable to be proved after the entire proof search loop, we employ LLMs to search for counterexamples by first negating the original theorem and attempting to prove its negation. This process either produces a verified counterexample that detects a bug or results in an undetermined case requiring human verification.

Fig A5 demonstrates the example of an original theorem and its negated counterpart. Using the straightforward `UserLogin` API as an illustration of negative theorem logic, we see how this API verifies whether the table contains multiple records with identical phone numbers. The original theorem presents the duplicated phone number scenario as a hypothesis, asserting that the output will be an error and the table will remain unchanged. In the negated theorem, the objective becomes finding input combinations that satisfy the hypothesis yet contradict the conclusion. Naturally, the proof strategies for these theorems diverge: the original positive theorem requires expanding the API function and its helper functions, then leveraging hypotheses to match cases until arriving at the desired output. In contrast, the negative theorem merely needs a counterexample that satisfies all hypotheses yet contradicts the expected conclusion. It's important to note that the proofs of these two theorems are based on distinct formal implementations of the `UserLogin` API, as only one theorem can be valid for any given implementation.

3 Results

3.1 Experiment Setup

We evaluate our approach on a dataset comprising four Scala projects developed within a functional programming framework. These projects vary in size and complexity, each containing 2 to 3 manually injected bug variants. The projects were generated using an AI-based project generator to ensure well-structured code organization and were subsequently verified by human engineers. All projects address real-world needs, with brief descriptions presented in Table A1. The projects contain between 2 and 25 APIs each, yielding more than 150 theorems about API functionality and 100 theorems about table properties, thus providing a diverse testbed for evaluation. Detailed project information is shown in Table A2.

For our pipeline implementation, we employ qwen-max [47] for all tasks except theorem proving, leveraging its state-of-the-art general capabilities and strong instruction-following performance. The proving tasks are handled by DeepSeek-R1, which we selected for its superior reasoning capabilities in formal verification scenarios.

3.2 Verification Statistics

In this section, we evaluate our pipeline's verification capability using the bug-free projects in our dataset. To assess each pipeline stage independently, we manually validate stage inputs to eliminate potential misalignments, which means we use the best outcome out of three attempts from the last step as the input to the next step. For non-proving tasks, we execute the pipeline three times to verify result consistency. The numbers presented in the tables are averaged over the three times. The theorem proving stage, however, benefits from built-in robustness through its global retry and local refinement mechanisms, making additional runs unnecessary—we therefore execute it only once per project.

3.2.1 Implementation Formalization

In the implementation formalization stage, there are two main tasks: dependency analysis and code formalization for tables and APIs. To evaluate dependency analysis, we measure the number of correctly identified dependencies. For table formalization, we classify the outcomes into three categories: success, semantic mismatch, and syntax error. For API formalization, we further refine the categorization into success, return type mismatch, logic mismatch, and syntax error. The results are presented in Table 1.

Table 1 Implementation Formalization results across the Scala projects. Table Dep. and API Dep. refer to dependency analysis for tables and APIs respectively, while Table Form. and API Form. refer to formalization results. For all tasks, Pos. represents successful cases without any misalignments or errors. For dependency analysis, Neg.: Wrong dependencies, Acc%: Accuracy percentage. For Table Formalization, Sem.: Semantic Mismatch, Syn.: Syntax Error. For API Formalization, Ret.: Return Type Mismatch, Logic: Logic Mismatch, Syn.: Syntax Error.

Project	**Table Dep.**			**Table Form.**				**API Dep.**			**API Form.**				
	Pos.	Neg.	Acc%	Pos.	Sem.	Syn.	Acc%	Pos.	Neg.	Acc%	Pos.	Ret.	Logic	Syn.	Acc%
UserAuth	1.0	0	100.0	1.0	0	0	100.0	2.0	0	100.0	2.0	0	0	0	100.0
BankAccount	2.0	0	100.0	2.0	0	0	100.0	5.0	0	100.0	5.0	0	0	0	100.0
Email	3.0	0	100.0	3.0	0	0	100.0	13.0	0	100.0	12.3	0	0.7	0	94.6
TaxiApp	3.0	0	100.0	3.0	0	0	100.0	25.0	0	100.0	24.0	0.7	0.3	0	96.0
Total	9.0	0	100.0	9.0	0	0	100.0	45.0	0	100.0	43.3	0.7	1.0	0	96.2

Our results demonstrate that the model achieves perfect accuracy on both dependency analysis and table formalization tasks. For API formalization, despite a small probability of misalignment, the majority of APIs are correctly formalized. Notably, return type misalignments rarely impact the verification process, as the model typically only adds extra return types rather than omitting essential ones.

3.2.2 Theorem Generation

In the theorem generation stage, parsing API documentation from the given backend system specification is relatively straightforward, so we don't treat it as a separate evaluation task. Instead, we evaluate four key tasks: API requirement generation, API theorem formalization, table property summarization, and table theorem formalization. For the natural language-based tasks, we assess the proportion of concise expressions that correctly describe input conditions and output types. For API requirements, we also verify the completeness of control flow paths. Additionally, for table property summarization, we confirm whether relevant APIs are correctly identified. For theorem formalization, results are classified into three categories: success, semantic mismatch, and syntax error. Table 2 presents these findings.

Table 2 Theorem Generation results across the Scala projects. API Req. and Tab. Prop. refer to API requirement generation and table property summarization respectively, while API Theorem and Tab. Theorem refer to theorem formalization results. For natural language tasks, Pos. represents successful cases with concise expressions, Neg.: cases with verbose or incorrect expressions. For API requirement generation, Mis.: missing any control flow path of the API. For table property summarization, API Err.: incorrect identification of relevant APIs. For theorem formalization, Sem.: Semantic Mismatch, Syn.: Syntax Error.

Project	**API Req.**				**API Theorem**				**Tab. Prop.**				**Tab. Theorem**			
	Pos.	Neg.	Mis.	Acc%	Pos.	Sem.	Syn.	Acc%	Pos.	Neg.	API Err.	Acc%	Pos.	Sem.	Syn.	Acc%
UserAuth	7.0	0	0	100.0	7	0	0	100.0	6.7	0	0	100.0	7.7	0	0.3	95.8
BankAccount	19.3	0	0	100.0	17	0	1	94.4	12.7	0	0	100.0	16.3	2.3	1.3	81.7
Email	43.3	0	0	100.0	42.3	0	0.7	98.4	21.3	0.3	0	98.5	32.0	5.3	0.7	84.2
TaxiApp	105.7	0	0.3	99.7	95.3	5.0	1.7	93.5	20.0	3.7	0	84.5	56.3	5.7	2.0	88.0
Total	175.3	0	0.3	99.8	161.6	5.0	3.4	95.1	60.7	4.0	0	93.8	112.3	13.3	4.3	86.5

To guarantee accurate evaluation, we apply comprehensive manual checks on all theorems and related descriptions. For natural language tasks, the model performs surprisingly well on API requirement generation, with all generated requirements being concise. This indicates the model successfully captures all premises and outcomes across each API control flow, with only one missing path.

Unlike API requirements, the model receives no predefined specifications when summarizing table properties. Consequently, some properties are not actually valid, though the related APIs are all correctly identified. These errors in table properties are not too problematic since they serve only as informal hypotheses that will undergo verification in the formal system later.

API theorem formalization achieves an accuracy about 95%, while table theorem formalization reaches approximately 85% accuracy. Among the two types of errors, syntax errors prevent certain control flows from being formalized into theorems, which need manual checks. Semantic mismatches produce compilable but requirement-inconsistent theorems, which undermine verification integrity. Most errors in table theorem formalization stem from semantic mismatches caused by the model's misunderstanding of table properties, as table theorems typically describe general properties while the model tends to add extra input conditions rather than providing general theorems, which makes the formalized theorem simpler and not aligned with the description.

3.2.3 Proof Search

In the proof search stage, we evaluate the pipeline's ability to prove the theorems generated in the previous stage. Since negated theorems are only relevant when identifying bugs in the implementation, we do not assess the generation and proof of negative theorems here, leaving that evaluation to the bug detection section described in Section 3.3. The proof search task has only two possible outcomes: success or failure. The results are presented in Table 3.

Table 3 Proof Search results across the Scala projects. The table shows the number of theorems successfully proved, those that remained unproved, and the overall proving ratio for APIs and tables.

Project	API Proof			Table Proof		
	Proved	Unproved	Proved%	Proved	Unproved	Proved%
UserAuth	4	3	57.1	3	4	42.9
BankAccount	10	7	58.8	11	7	61.1
Email	34	8	81.0	24	13	64.9
TaxiApp	52	50	51.0	34	29	54.0
Total	100	68	59.5	72	53	57.6

The results indicate that over 50% of theorems are successfully proved, enabling verification of half of all API specifications in addition to automatically analyzed and proved table properties, which can reduce manual verification labor costs by half, even though the successfully proven theorems are sometimes relatively simple compared to the unproven ones.

Since no specialized prover model exists for generating proofs for this kind of theorems that is related to real-world logic and has complex outside dependencies, the current results from our general reasoning model are quite promising. These results can be further enhanced by collecting proof data and fine-tuning a task-specific model, suggesting significant potential to both improve accuracy and reduce costs in the proof search stage.

Currently, the proof budget for each theorem allows 5 attempts with 8 refinement retries per attempt. In our evaluation of the TaxiApp project, the average token-based model cost is $0.34 per theorem, with verification costs averaging $2.19 per API, while test engineers in America earn $52 per hour on average. The expenses from the first two stages are negligible compared to proof costs. These expenses could be reduced by implementing more efficient provers, which would allow for a smaller proof budget.

Notably, we observe that the proving ratio of theorems does not decrease as projects scale in size. This indicates that, given the decomposable nature of our functional programming framework, our pipeline remains scalable to larger projects, while increased coroutine utilization can maintain low time complexity. The consistent accuracy of the theorem generation task in Section 3.2.2 further supports this conclusion.

3.3 Bug Detection

In this section, we examine the pipeline from a different perspective, treating it as a unified tool for bug detection. We evaluate its capability by assessing whether it can identify injected bugs in our dataset across the three stages.

(1) During implementation formalization, the model is allowed to issue warnings about potential bugs but must still formalize the code exactly as given, even if it contains errors. (2) In theorem generation, the formalization derived from the buggy implementation may fail to produce theorems that align with the specifications (e.g., missing return types, missing parameters, etc.). The model can flag inconsistencies between the formalization and the specifications but must still generate a theorem statement that best matches the intended meaning. (3) Finally, in the proof search stage, bugs can naturally be detected by identifying counterexamples when a theorem is not provable.

For evaluation, we check whether a bug is detected at any of these three stages and count the total occurrences of successful detections. The results are presented in Table 4. For a detailed list of the bugs in the variants, see Table A3.

Table 4 Bug Detection results across the Scala projects. For each project variant, the table shows whether the bug was detected in each of the three stages: Impl. Form. (Implementation Formalization), Theorem Gen. (Theorem Generation), and Proof Search. The Sum column represents the total number of stages that successfully detected the bug (0-3). tick: detected, cross: not detected, circle: unable to detect because of misaligned theorem.

Project	Variant ID	Impl. Form.	Theorem Gen.	Proof Search	Sum
UserAuth	1	✓	✓	✓	3
	2	✗	✓	✓	2
BankAccount	1	✗	✗	✓	1
	2	✗	✓	✓	2
	3	✗	✗	✗	0
Email	1	✗	✗	✗	0
	2	✗	✓	◯	1
	3	✗	✓	◯	1
TaxiApp	1	✗	✓	◯	1
	2	✗	✓	✓	2
	3	✗	✗	✗	0

The experiments show that 8 out of 11 bugs are successfully detected by finding counterexamples, while the remaining 3 bugs are still hidden among the theorems that cannot be proved correct or incorrect. According to the results in Section 3.2.3, that proportion is no more than half of all the theorems. Among the results, circle mark indicates that this bug prevents the model from writing a theorem aligned with the requirement, so it will be detected in the first two stages and the theorem in the proof search no longer makes sense.

4 Discussion

We have proposed a novel framework that leverages functional programming and type systems to translate Scala backend code into formal Lean representations, addressing

limitations in existing automated testing approaches and expanding automatic testing to system-level backends. By combining LLMs' general capabilities with formal system rigor, our pipeline automatically generates theorems that specify API behavior based on documentation, summarizes and generates table-related properties, and applies LLM-based provers to either verify them through formal proof or detect bugs by finding counterexamples. This approach eliminates the need for human intervention on formally verified functionalities and detected bugs, leaving only the unresolved portion for manual verification.

According to evaluation results on realistic backend systems, our method formally verifies over 50% of requirements and detects 70% of bugs in fault variants, automating more than half of the testing workload. The average cost to verify an API is only $2.19, which is remarkably cost-effective compared to a test engineer's hourly earnings of approximately $52.

Moreover, our method achieves scalability through the decomposable nature of the functional programming framework. Experimental results in Section 3.2.2 and Section 3.2.3 demonstrate that theorem generation accuracy and proof success rates remain consistent regardless of backend size. Since proof search constitutes the majority of pipeline time and cost, verification time complexity for complex projects can be reduced to a constant by simply increasing parallel LLM request executions. This approach offers flexibility comparable to temporarily scaling a testing team—analogous to hiring ten times more test engineers for a project of tenfold complexity and releasing them afterward.

Compared to existing methods focused on automated testing, our approach emphasizes whole-system verification of real-world systems, targeting specification compliance rather than low-level or function-level vulnerabilities. The general capabilities of LLMs make our pipeline scalable not only in project complexity but also across input languages, provided the codebase adheres to functional programming paradigms that enable natural decomposition.

Although our current implementation is based on Scala, the benefits of this approach extend far beyond Scala projects. Scala runs on the JVM and is fully interoperable with Java, allowing it to use any Java library. In particular, Java-based frameworks such as Spring Boot can be systematically translated into Scala, enabling our method to be applied to a wide range of real-world applications that rely on Java ecosystems.

An important requirement of our method is guaranteeing accuracy in two key stages: (1) implementation formalization, ensuring semantic equivalence between implementation and formal representation, and (2) theorem generation, producing formal theorems aligned with natural language specifications. While both stages currently achieve over 95% accuracy, the remaining misalignments could be further reduced through better specialized models. Additionally, pre-formalizing external services like databases as standard components in the future would help ensure greater equivalence between implementation and formalization.

In summary, our results demonstrate the viability of employing LLMs throughout the entire reliable software testing process, pointing to a promising direction for scalable, AI-powered software testing, with the potential to greatly improve engineering productivity as models continue to advance.

5 Acknowledgements

We gratefully acknowledge Jiawen Tao for valuable contributions to discussions regarding the framework design. We also thank the Functor Team for assistance with AI-generated projects.

6 Author contributions

Y.Y. and A.C.C.Y. supervised the project. K.X. and Y.L. contributed to the conception and design of the work. K.X. implemented the pipeline code and ran the main experiments. K.X. and Y.L. prepared the data for the experiments. K.X., Y.L. and Y.Y. contributed to the paper writing and prepared the figures. All authors revised the manuscript.

Appendix A Extended Data

A.1 Illustrative Example

```yaml
name: Transaction
type: Table
description: Stores user balance operation records
parameters:
- parameter_name: id
  parameter_description: Primary key, uniquely identifies each operation record
  parameter_type: Long
  is_list: false
  is_required: true
- parameter_name: username
  parameter_description: Username associated with the operation record
  parameter_type: String
  is_list: false
  is_required: true
- parameter_name: amount
  parameter_description: Balance amount modified by the operation
  parameter_type: Int
  is_list: false
  is_required: true
```

```
-- namespace
namespace BankAccount.BalanceManagementService.Tables.Transaction

-- structure_definition
structure TransactionRow where
  id : Nat
  username : String
  amount : Int
  deriving Repr

structure TransactionTable where
  rows : List TransactionRow
  deriving Repr

-- end namespace
end BankAccount.BalanceManagementService.Tables.Transaction
```

Fig. A1 An example of formalizing a table structure. The `Transaction` table structure is derived from the given table description.

A.2 Data

A.2.1 Evaluation Projects

Table A1 Brief descriptions of the Scala projects used for evaluation.

Project	Description
UserAuth	Simple user authentication system with login and registration APIs.
BankAccount	Banking system with APIs for account management and transactions, including deposits and withdrawals.
Email	Email service for sending and receiving messages, with authentication and group management support.
TaxiApp	Taxi booking system with account management, ride status tracking controlled by passengers and drivers, and payment processing.

A.2.2 Bug Variants

```scala
package Impl.BalanceManagementService

import Common.API.{PlanContext, Planner}
import Common.DBAPI.{readDBInt, writeDB}
import Common.Object.SqlParameter
import APIs.BalanceManagementService.BalanceQueryMessage
import cats.effect.IO
import io.circe.generic.auto._

case class WithdrawalMessagePlanner(
                                     Username: String,
                                     Password: String,
                                     Amount: Int,
                                     override val planContext: PlanContext
                                   ) extends Planner[Int] {

  override def plan(using planContext: PlanContext): IO[Int] = {
    for {
      // Step 1: Check if the amount is a positive integer
      _ <- checkPositiveAmount

      // Step 2: Query the current user balance
      currentBalance <- queryCurrentBalance

      // Step 3: Calculate the new balance after withdrawal
      newBalance = calculateNewBalance(currentBalance)

      // Step 4: Check if the new balance is non-negative
      _ <- checkSufficientBalance(newBalance)

      // Step 5: Record the withdrawal operation in the database
      _ <- recordWithdrawalOperation

      // Step 6: Return the new balance
    } yield newBalance
  }

  // Step 1: Check if the amount is a positive integer
  private def checkPositiveAmount(using PlanContext): IO[Unit] = {
    if (Amount > 0) IO.unit
    else IO.raiseError(new IllegalArgumentException("Amount must be a positive integer"))
  }

  // Step 2: Query the current user balance
  private def queryCurrentBalance(using PlanContext): IO[Int] = {
    BalanceQueryMessage(Username, Password).send
  }

  // Step 3: Calculate the new balance after withdrawal
  private def calculateNewBalance(currentBalance: Int): Int = {
    currentBalance - Amount
  }

  // Step 4: Check if the new balance is non-negative
  private def checkSufficientBalance(newBalance: Int)(using PlanContext): IO[Unit] = {
    if (newBalance >= 0) IO.unit
    else IO.raiseError(new IllegalArgumentException("Insufficient balance"))
  }

  // Step 5: Record the withdrawal operation in the database
  private def recordWithdrawalOperation(using PlanContext): IO[String] = {
    val sql =
      s"""
         INSERT INTO Transaction (username, amount)
         VALUES (?, ?);
       """

    writeDB(sql, List(SqlParameter("String", Username), SqlParameter("Int", (-Amount).toString)))
  }
}
```

```
-- imports
import BankAccount.BalanceManagementService.Tables.Transaction
import BankAccount.BalanceManagementService.APIs.BalanceQuery
import BankAccount.AccountManagementService.Tables.Account

open BankAccount.BalanceManagementService.Tables.Transaction
open BankAccount.BalanceManagementService.APIs.BalanceQuery
open BankAccount.AccountManagementService.Tables.Account

-- namespace
namespace BankAccount.BalanceManagementService.APIs.Withdrawal

-- helper_functions
inductive WithdrawalResult where
  | Success : Int → WithdrawalResult
  | InvalidAmount : WithdrawalResult
  | AuthorizationFailure : WithdrawalResult
  | IntegrityError : WithdrawalResult
  | InsufficientBalance : WithdrawalResult

def checkPositiveAmount (amount : Int) : WithdrawalResult :=
  if amount > 0 then WithdrawalResult.Success 0
  else WithdrawalResult.InvalidAmount

def calculateNewBalance (currentBalance : Int) (amount : Int) : Int :=
  currentBalance - amount

def checkSufficientBalance (newBalance : Int) : WithdrawalResult :=
  if newBalance >= 0 then WithdrawalResult.Success 0
  else WithdrawalResult.InsufficientBalance

def recordWithdrawalOperation (username : String) (amount : Int)
  (old_transaction_table : TransactionTable) : TransactionTable :=
  let newRow := { id := old_transaction_table.rows.length + 1, username := username, amount := -
amount }
  { rows := old_transaction_table.rows ++ [newRow] }

-- main_function
def withdrawal (username : String) (password : String) (amount : Int)
  (old_account_table : AccountTable) (old_transaction_table : TransactionTable) :
  WithdrawalResult × AccountTable × TransactionTable :=
  -- Step 1: Check if amount is positive
  match checkPositiveAmount amount with
  | WithdrawalResult.InvalidAmount => (WithdrawalResult.InvalidAmount, old_account_table,
old_transaction_table)
  | _ =>
    -- Step 2: Query current balance
    let (balanceQueryResult, new_account_table, _) := balanceQuery username password
old_account_table old_transaction_table
    match balanceQueryResult with
    | BalanceQueryResult.Success currentBalance =>
      -- Step 3: Calculate new balance
      let newBalance := calculateNewBalance currentBalance amount
      -- Step 4: Check sufficient balance
      match checkSufficientBalance newBalance with
      | WithdrawalResult.InsufficientBalance => (WithdrawalResult.InsufficientBalance,
new_account_table, old_transaction_table)
      | _ =>
        -- Step 5: Record withdrawal operation
        let new_transaction_table := recordWithdrawalOperation username amount old_transaction_table
        (WithdrawalResult.Success newBalance, new_account_table, new_transaction_table)
    | BalanceQueryResult.AuthorizationFailure => (WithdrawalResult.AuthorizationFailure,
new_account_table, old_transaction_table)
    | BalanceQueryResult.IntegrityError => (WithdrawalResult.IntegrityError, new_account_table,
old_transaction_table)

-- end namespace
end BankAccount.BalanceManagementService.APIs.Withdrawal
```

Fig. A2 An example of formalizing an API in Lean. The `WithDrawal` API is formalized based on the implementation code, along with the tables (`Transaction`) and APIs (`BalanceQuery`) it depends on.

Table A2 Project Information for the Scala Projects. The info includes the number of total services, tables, APIs, and the number of bug variants.

Project	#Services	#Table	#API	#Variant
UserAuth	1	1	2	2
BankAccount	3	2	5	3
Email	3	3	13	3
TaxiApp	5	3	25	3

Table A3 Descriptions of the bug variants.

Project	Variant ID	Description
UserAuth	1	`UserRegister` adds record without checking existence.
	2	`UserLogin` has no check for db error when duplicated phone numbers exist.
BankAccount	1	`BalanceQuery` sums all transactions instead of those of the user.
	2	`Deposit` doesn't check positive amount.
	3	`Withdrawal` stores (amount) in table instead of (-amount)
Email	1	`AddUser` doesn't check if the current user is the owner of the group.
	2	`DeleteUser` allows the deletion of the owner of the group.
	3	`SendToGroup` calls the wrong dependent API to check only group exists instead of the user in the group.
TaxiApp	1	`Login` doesn't delete old tokens when exist.
	2	`UpdateRideStatus` allows to move from `awaiting_payment` to `cancelled`.
	3	`PayForRide` only checks if the token is a passenger instead of checking if it matches the `ride_id`.

```
### Withdrawal
Accepts three parameters: a username, a password, and an amount. Depends on the BalanceQuery API:
- The amount must be a positive integer; otherwise, it returns an "Invalid Parameter" error.
- Calls the BalanceQuery function:
    - If authentication fails, it returns a "User Not Found or Incorrect Password" error.
    - If a database error occurs, it returns a database error.
    - If the current balance is retrieved, the system calculates the new balance after withdrawal:
        - If the new balance is negative, it returns an "Insufficient Balance" error and does not
modify the database.
        - If the new balance is non-negative, a new withdrawal record is added to the database with
the amount stored as its negative value, and the new balance is returned upon success.
```

```
- If the amount is not a positive integer, return an "Invalid Parameter" error and keep all tables
unchanged.

- If the amount is a positive integer and the BalanceQuery API returns an authentication failure,
return a "User Not Found or Incorrect Password" error and keep all tables unchanged.

- If the amount is a positive integer, the BalanceQuery API does not return an authentication
failure, and a database error occurs, return a database error and keep all tables unchanged.

- If the amount is a positive integer, the BalanceQuery API retrieves the current balance
successfully, and the new balance after withdrawal is negative, return an "Insufficient Balance"
error and keep all tables unchanged.

- If the amount is a positive integer, the BalanceQuery API retrieves the current balance
successfully, and the new balance after withdrawal is non-negative, add a new withdrawal record to
the database with the amount stored as its negative value, and return the new balance.
```

```
...
-- comment
/- If the amount is a positive integer, the BalanceQuery API retrieves the current balance
successfully,
   and the new balance after withdrawal is non-negative, add a new withdrawal record to the database
   with the amount stored as its negative value, and return the new balance. -/

-- theorem proved
theorem withdrawalSuccessWhenValid
    (username : String)
    (password : String)
    (amount : Int)
    (old_account_table : AccountManagementService.Tables.Account.AccountTable)
    (old_transaction_table : TransactionTable)
    (currentBalance : Int)
    (new_account_table : AccountManagementService.Tables.Account.AccountTable)
    (new_transaction_table : TransactionTable)
    (h_positive_amount : checkPositiveAmount amount = WithdrawalResult.Success 0)
    (h_balance_query_success : balanceQuery username password old_account_table
old_transaction_table = (BalanceQueryResult.Success currentBalance, new_account_table,
new_transaction_table))
    (h_sufficient_balance : calculateNewBalance currentBalance amount ≥ 0) :
    let (result, _, final_transaction_table) := withdrawal username password amount
old_account_table old_transaction_table;
    result = WithdrawalResult.Success (calculateNewBalance currentBalance amount) ∧
    final_transaction_table = recordWithdrawalOperation username amount old_transaction_table := by
  sorry
...
```

Fig. A3 An example of API theorem generation following a sequential process: API document interpretation, requirement analysis, and formal theorem formulation. The document on the top is parsed to produce parallel requirements presented in the middle, with the successful control flow (highlighted in red) becoming the final requirement after necessary premises are added. This requirement is then analyzed to construct the complete formal theorem.

```
If the current table has N records, then after applying any of these APIs with invalid input
parameters, authentication failure, insufficient balance, or database errors, the table will still
have N records
```

```
If the current table has N records, then after applying the Withdrawal or Deposit APIs successfully,
the table will have N + 1 records
```

```
If the current table has N records, then after applying the BalanceQuery API, the table will still
have N records
```

```
...
-- comment
/- If the current table has N records, then after applying the BalanceQuery API, the table will
still have N records. -/

-- theorem proved
theorem balanceQueryPreservesRecordCount
    (username : String)
    (password : String)
    (old_account_table : AccountTable)
    (old_transaction_table : TransactionTable)
    (N : Nat)
    (h_initial_count : getRowCount old_transaction_table = N) :
    let (result, new_account_table, new_transaction_table) := balanceQuery username password
old_account_table old_transaction_table;
    getRowCount new_transaction_table = N := by
  -- Unfold the balanceQuery function to analyze its structure
  unfold balanceQuery
  -- Split the proof based on the authorization result
  cases h_auth : (AccountManagementService.APIs.Authorization.authorization username password
old_account_table).fst
    <;> simp_all [h_auth]
  -- All cases preserve the transaction table, apply initial count hypothesis
  <;> exact h_initial_count
...
```

```
...
-- comment
/- If the current table has N records, then after applying the Deposit API successfully,
   the table will have N + 1 records. -/

-- theorem proved
theorem depositIncreasesTransactionCount
    (username : String)
    (password : String)
    (amount : Int)
    (current_balance : Int)
    (old_account_table : AccountTable)
    (old_transaction_table : TransactionTable)
    (N : Nat)
    (h_valid_amount : validateAmount amount = DepositResult.Success 0)
    (h_auth_success : balanceQuery username password old_account_table old_transaction_table =
        (BalanceQueryResult.Success current_balance, old_account_table, old_transaction_table))
    (h_initial_size : old_transaction_table.rows.length = N) :
    let (result, new_account_table, new_transaction_table) :=
      deposit username password amount old_account_table old_transaction_table;
    result = DepositResult.Success (current_balance + amount) ∧
    new_transaction_table.rows.length = N + 1 ∧
    getLastElement new_transaction_table.rows = some { id := N + 1, username := username, amount :=
amount } := by
  -- Unfold the deposit function to analyze its structure
  unfold deposit
  -- Simplify using the validateAmount hypothesis
  simp [h_valid_amount]
  -- Substitute the balanceQuery result using the given hypothesis
  rw [h_auth_success]
  -- Split the proof based on the balanceQueryResult, focusing on the Success case
  split <;> simp_all [recordDepositTransaction, h_initial_size]
  -- Substitute currentBalance† with 0 using the equality hypothesis
  <;> subst_vars
  -- Simplify arithmetic and verify the transaction table's new length and last element
  <;> simp_all [List.length_append, List.reverse_append, List.length_singleton, getLastElement]
  <;> simp_all [List.reverse_append]
...
```

Fig. A4 Examples of table properties and theorems. The figure illustrates three properties concerning record numbers in the `Transaction` table across its three interacting APIs. The first theorem is related to `BalanceQuery` and is derived from the third property. The second one is associated with the second property, specifically addressing the `Deposit` operation.

```
-- comment
/- If the user table contains multiple records with the same phone number as the input,
   the API must return a database integrity error, and the user table must remain unchanged. -/

-- theorem unproved
theorem multipleRecordsReturnErrorAndUnchangedTable
    (phoneNumber : String)
    (password : String)
    (old_user_table : UserTable)
    (h_duplicates : hasDuplicatePhoneNumbers old_user_table phoneNumber) :
    let (result, new_user_table) := userLogin phoneNumber password old_user_table;
    result = UserLoginResult.MultiplePasswordsError ∧
    new_user_table = old_user_table := by
  -- Unfold the definition of userLogin to analyze its structure
  unfold userLogin
  -- Simplify the queryUserRecord call using the hypothesis h_duplicates
  unfold hasDuplicatePhoneNumbers at h_duplicates
  simp_all
  let h_has_records : (old_user_table.rows.filter (λ row => row.phone_number == phoneNumber)).length
> 0 := by
    -- Use 1 > 0
    let h_zero_lt_one : 0 < 1 := by
      decide
    let record_count := (old_user_table.rows.filter (λ row => row.phone_number ==
phoneNumber)).length
    apply Nat.lt_trans h_zero_lt_one h_duplicates
  -- Unfold the definition of queryUserRecord
  unfold queryUserRecord
  simp_all
  -- Unfold the definition of getStoredPassword
  unfold getStoredPassword
  simp_all
```

```
-- comment
/- The original statement was incorrect:
   If the user table contains multiple records with the same phone number as the input,
   the API does not always return a database integrity error or leave the user table unchanged. -/

-- theorem unproved
theorem notMultipleRecordsReturnErrorAndUnchangedTable :
    ∃ (phoneNumber : String)
      (password : String)
      (old_user_table : UserTable)
      (h_duplicates : hasDuplicatePhoneNumbers old_user_table phoneNumber),
      let (result, new_user_table) := userLogin phoneNumber password old_user_table;
      (result ≠ UserLoginResult.MultiplePasswordsError ∨ new_user_table ≠ old_user_table) := by
  -- Provide concrete counterexample with multiple matches using refine'
  refine' ("1234567890", "correct", { rows := [
    { phone_number := "1234567890", password := "correct" },
    { phone_number := "1234567890", password := "wrong" }
  ]}, by decide, _)
  -- Simplify API call and validate result
  simp [userLogin, queryUserRecord, getStoredPassword, validatePassword]
```

Fig. A5 An example of negative theorem generation. The second theorem represents the negated version of the first theorem. Their respective proofs derive from different API implementations—the correct implementation supports the original theorem, while a flawed implementation substantiates the negative theorem.

References

[1] Verma, A. Software testing market size & share: growth report 2025-2037 (2025). Research Nester, Report ID: 6819 https://www.researchnester.com/reports/software-testing-market/6819.

[2] Whittaker, J. A., Arbon, J. & Carollo, J. *How Google tests software* (Addison-Wesley, 2012).

[3] Chen, Y. *et al.* d'Amorim, M. (ed.) *Chatunitest: A framework for llm-based test generation.* (ed.d'Amorim, M.) *Companion proceedings of the 32nd ACM international conference on the foundations of software engineering*, 572–576 (2024).

[4] Lops, A., Narducci, F., Ragone, A., Trizio, M. & Bartolini, C. A system for automated unit test generation using large language models and assessment of generated test suites (2024). Preprint at https://arxiv.org/abs/2408.07846.

[5] Nama, P., Reddy, P. & Pattanayak, S. K. Artificial intelligence for self-healing automation testing frameworks: real-time fault prediction and recovery. *Artificial Intelligence* **64** (2024).

[6] Tihanyi, N. *et al.* Vulnerability detection: from formal verification to large language models and hybrid approaches: a comprehensive overview (2025). Preprint at https://arxiv.org/abs/2503.10784.

[7] Yang, K. *et al.* Formal mathematical reasoning: a new frontier in ai (2024). Preprint at https://arxiv.org/abs/2412.16075.

[8] Menezes, R. *et al.* Finkbeiner, B. & Kovács, L. (eds) *ESBMC 7.4: Harnessing the Power of Intervals.* (eds Finkbeiner, B. & Kovács, L.) *30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 376–380 (Springer, 2024).

[9] De Moura, L. & Bjørner, N. Ramakrishnan, C. R. & Rehof, J. (eds) *Z3: an efficient SMT solver.* (eds Ramakrishnan, C. R. & Rehof, J.) *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 337–340 (Springer, 2008).

[10] Barrett, C. *et al.* Gopalakrishnan, G. & Qadeer, S. (eds) *cvc4.* (eds Gopalakrishnan, G. & Qadeer, S.) *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, 171–177 (Springer, 2011).

[11] Leino, K. R. M. Fermüller, C. G. & Voronkov, A. (eds) *Dafny: an automatic program verifier for functional correctness.* (eds Fermüller, C. G. & Voronkov, A.) *International Conference on Logic for Programming Artificial Intelligence*

and Reasoning, 348–370 (Springer, 2010).

[12] Lattuada, A. *et al.* Verus: verifying Rust programs using linear ghost types (extended version) (2023). Preprint at https://arxiv.org/abs/2303.05491.

[13] Mendez, J. A. Soda: an object-oriented functional language for specifying human-centered problems (2023). Preprint at https://arxiv.org/abs/2310.01961.

[14] Team, T. R. D. The rocq prover, 9.0. Zenodo (2025). URL https://doi.org/10.5281/zenodo.15149629.

[15] Paulson, L. C. Natural deduction as higher-order resolution. *The Journal of Logic Programming* **3**, 237–258 (1986).

[16] Moura, L. d. & Ullrich, S. Platzer, A. & Sutcliffe, G. (eds) *The Lean 4 theorem prover and programming language.* (eds Platzer, A. & Sutcliffe, G.) *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, 625–635 (Springer, 2021).

[17] Czajka, L. & Kaliszyk, C. Hammer for Coq: automation for dependent type theory. *Journal of Automated Reasoning* **61**, 423–453 (2018).

[18] First, E., Brun, Y. & Guha, A. TacTok: semantics-aware proof synthesis. *Proceedings of the ACM on Programming Languages* **4**, 1–31 (2020).

[19] Goel, S. & Ray, S. in *Microprocessor assurance and the role of theorem proving* (ed.Chattopadhyay, A.) *Handbook of Computer Architecture* 1–43 (Springer, 2022).

[20] Leroy, X. *et al.* SEE & 3AF (eds) *CompCert-a formally verified optimizing compiler.* (eds SEE & 3AF) *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress* (SEE MIDI-PYRENEES and 3AF, 2016).

[21] Chajed, T., Tassarotti, J., Theng, M., Kaashoek, M. F. & Zeldovich, N. Aguilera, M. K. & Weatherspoon, H. (eds) *Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning.* (eds Aguilera, M. K. & Weatherspoon, H.) *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 447–463 (2022).

[22] Klein, G. *et al.* Matthews, J. N. & Anderson, T. E. (eds) *seL4: formal verification of an OS kernel.* (eds Matthews, J. N. & Anderson, T. E.) *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 207–220 (2009).

[23] Bhatia, S., Qiu, J., Hasabnis, N., Seshia, S. & Cheung, A. Verified code transpilation with LLMs. *Advances in Neural Information Processing Systems* **37**, 41394–41424 (2024).

[24] Liu, Q., Zheng, X., Lu, X., Cao, Q. & Yan, J. Yue, Y. (ed.) *Rethinking and improving autoformalization: towards a faithful metric and a Dependency Retrieval-based approach.* (ed.Yue, Y.) *The Thirteenth International Conference on Learning Representations* (2025).

[25] Tihanyi, N. *et al.* A new era in software security: towards self-healing software via large language models and formal verification (2023). Preprint at https://arxiv.org/abs/2305.14752.

[26] Cao, J. *et al.* From informal to formal–incorporating and evaluating LLMs on natural language requirements to verifiable formal proofs (2025). Preprint at https://arxiv.org/abs/2501.16207.

[27] Lin, X. *et al.* FVEL: Interactive formal verification environment with large language models via theorem proving. *Advances in Neural Information Processing Systems* **37**, 54932–54946 (2024).

[28] Wu, H., Barrett, C. & Narodytska, N. Lemur: integrating large language models in automated program verification (2023). Preprint at https://arxiv.org/abs/2310.04870.

[29] Mugnier, E., Anaya Gonzalez, E., Jhala, R., Polikarpova, N. & Zhou, Y. Laurel: generating dafny assertions using large language models (2024). Preprint at https://arxiv.org/abs/2405.

[30] Si, X., Naik, A., Dai, H., Naik, M. & Song, L. Lahiri, S. K. & Wang, C. (eds) *Code2inv: a deep learning framework for program verification.* (eds Lahiri, S. K. & Wang, C.) *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*, 151–164 (Springer, 2020).

[31] Misu, M. R. H., Lopes, C. V., Ma, I. & Noble, J. Towards ai-assisted synthesis of verified dafny methods. *Proceedings of the ACM on Software Engineering* **1**, 812–835 (2024).

[32] Loughridge, C. *et al.* DafnyBench: a benchmark for formal software verification (2024). Preprint at https://arxiv.org/abs/2406.08467.

[33] Gadde, D. N., Kumar, A., Nalapat, T., Rezunov, E. & Cappellini, F. All artificial, less intelligence: GenAI through the lens of formal verification (2024). Preprint at https://arxiv.org/abs/2403.16750.

[34] Liu, M., Kang, M., Hamad, G. B., Suhaib, S. & Ren, H. Ozev, S. & Tahoori, M. (eds) *Domain-adapted LLMs for VLSI design and verification: a case study on formal verification.* (eds Ozev, S. & Tahoori, M.) *2024 IEEE 42nd VLSI Test Symposium (VTS)*, 1–4 (2024).

[35] Liu, Y. *et al.* PropertyGPT: LLM-driven formal verification of smart contracts through retrieval-augmented property generation (2024). Preprint at https://arxiv.org/abs/2405.02580.

[36] Curaba, C., D'Ambrosi, D., Minisini, A. & Pérez-Campanero Antolín, N. CryptoFormalEval: integrating LLMs and formal verification for automated cryptographic protocol vulnerability detection (2024). Preprint at https://arxiv.org/abs/2411.13627.

[37] Johnstone, P. T. *Sketches of an Elephant: A Topos Theory Compendium* Vol. 43 of *Oxford Logic Guides* (Oxford University Press, Oxford, 2002).

[38] Hughes, J. Why Functional Programming Matters. *The Computer Journal* **32**, 98–107 (1989).

[39] Odersky, M., Spoon, L. & Venners, B. *Programming in scala* (Artima Inc, 2008).

[40] Plotkin, G. & Pretnar, M. Castagna, G. (ed.) *Handlers of algebraic effects*. (ed.Castagna, G.) *European Symposium on Programming*, 80–94 (Springer, 2009).

[41] Pretnar, M. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science* **319**, 19–35 (2015).

[42] Bauer, A. & Pretnar, M. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* **84**, 108–123 (2015).

[43] Kiselyov, O. & Sivaramakrishnan, K. Eff directly in OCaml (2018). Preprint at https://arxiv.org/abs/1812.11664.

[44] Leijen, D. Koka: programming with row polymorphic effect types (2014). Preprint at https://arxiv.org/abs/1406.2061.

[45] Lindley, S., McBride, C. & McLaughlin, C. Do be do be do (2017). Preprint at https://arxiv.org/abs/1611.09259.

[46] Guo, D. *et al.* Deepseek-r1: incentivizing reasoning capability in llms via reinforcement learning (2025). Preprint at https://arxiv.org/abs/2501.12948.

[47] Yang, A. *et al.* Qwen2.5 technical report (2024). Preprint at https://arxiv.org/abs/2412.15115.