International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(14s) 2026 pp. 301-332
https://doi.org/10.70917/ijcisim-2026-4214
Article

SQLProof: A Hybrid Security Verification Framework for LLM-Based Text-to-SQL Systems

[1]Firas Tarik Jasim, [2]Kabir G. Kharade

Research Scholar, Computer Science Department of Computer Science Shivaji University, Kolhapur, Maharashtra, India
Email: Firas.tj@ntu.edu.iq
Assistant Professor Department of Computer Science Shivaji University, Kolhapur, Maharashtra, India
Email: kgk_csd@unishivaji.ac.in

Abstract: Large language model (LLM) Text-to-SQL systems can improve the accessibility of databases, but they also pose new security risks not addressed by conventional SQL-injection classifiers. A query generated by a generator might be syntactically correct and not contain any known payload signatures, but still fail to meet the user's intent, cross authorization boundaries, be influenced by retrieved content that is not trusted, or be generated by a backdoored model. In this study, we introduce SQLProof, a proof-carrying execution framework in which every generated SQL query should meet independent obligations of intent alignment, policy compliance, source authorization, information-flow isolation, write safety and resource cost prior to execution. The LLM is not part of the trusted computing base. SQLProof transforms a natural-language request into an intent contract, combines it with a role- and context-aware authorization contract, converts the resulting query into a semantic graph, and performs a bounded semantic comparison and a provenance analysis of the resulting obligations against a deterministic parsing and a policy check. The framework was validated using the ProofSQL-Bench, which contains 4,000 executable security episodes, including direct and indirect prompt injection, backdoor text-to-SQL generation, semantic privilege escalation, leakage of sensitive data, unsafe writes and resource-exhaustion queries. In the eight security scenarios, SQLProof could block the attacks with a macro-averaged attack-blocking rate of 97.9%, while the strongest baseline that was evaluated in this study could block attacks with a macro-averaged attack-blocking rate of 80.1%. The acceptance rates for benign queries were 94.1% for SQLProof and 92.6% for the strongest baseline evaluated in this study. SQLProof achieved an unauthorized-execution rate of 0.6%, as compared to 4.7% for the best baseline tested in this study. Ablation results show that intent contracts and influence provenance contributed the most to the end-to-end SQLProof decision latency beyond just policy-only enforcement, which was 43.8 ms. SQLProof was able to conduct post-generation security verification for all the assessed text-to-SQL generators for the assessed threat model and benchmark conditions.

Keywords: Text-to-SQL, LLM security, prompt injection, backdoor attacks, hybrid verification, bounded semantic assurance, proof-carrying code, database authorization, information-flow provenance

1. Introduction

Large language models (LLMs) have substantially changed the design of natural-language database interfaces[1]. Modern T2S systems are no longer restricted to converting a natural language query into a syntactically correct SQL query; they are more likely to be database-oriented agents that can interpret complex database schemas, add contextual information, call external tools, refine intermediate results, and produce an executable SQL query in a wide variety of heterogeneous database environments. Early neural methods like Seq2SQ showed the potential of learning structured-query generation directly from natural language, and Spider proved to be a large-scale cross-domain evaluation on previously unseen

database schemas. Constrained-generation was then used to enhance structural reliability by discarding syntactically incorrect SQL statements during autoregressive decoding, as in PICARD [2]. Recent research has continued this trend of large-scale LLM-based interfaces for databases and a unified benchmark-driven evaluation [3].

However, these paradigms focus mainly on the correctness of the generations, the correctness of execution, and the cross-domain generalization, not on adversarial authorization[4].

This is significant because while functional correctness does not necessarily imply security, security does imply functional correctness[5]. Recent efforts in the area of SQL semantic-equivalence assessment have also shown that it is itself a non-trivial problem to find out whether two executable SQL programs have the same intended behavior [6]. From a security standpoint, more significantly, an LLM-generated query can be syntactically correct, executable, and semantically related to a legitimate request, but still be outside of its scope of authorization. For instance, it can weaken required predicates, reveal sensitive attributes that are not required, span tenant boundaries, add unwanted write effects, or add unwanted computational cost[7]. The resulting query may not have any recognizable malicious payload, unlike traditional SQL injection attacks. While the conventional SQL-injection detection has been significantly enhanced by pretrained-code models like CodeBERT, a malicious-versus-benign classification does not guarantee that the otherwise valid SQL operation is indeed required for the authorized task[8].

As external context is integrated into LLM applications, there is an added security boundary. Adversarial instructions can be directly typed by the user or indirectly generated from retrieved documents, conversational memory, metadata, database content, and tool outputs. StruQ tackles this issue with structural separation of trusted instructions and untrusted data [9]. For LLM-based applications, multilayer defenses have also integrated prompt sanitization, anomaly detection, and traditional SQL protection techniques to minimize the risk of unsafe database actions resulting from manipulated prompts. These approaches reinforce the input and generation boundaries, but they cannot as a standalone prove that the SQL statement that is ultimately presented for execution is semantically valid[10].

This is a more serious issue if the SQL generator is not fully trusted. Today's database-agent architectures are expanding to spread reasoning and SQL generation across several LLM components, which leads to more complex interaction surfaces where a failed or compromised component can affect the execution of an otherwise valid database action. SQLProof therefore makes a different assumption about the security: the generator is not part of the trusted computing base, and the execution authority is decided independently at the time of generation[11].

This design is based on the larger idea of proof-carrying code, where there is executable behavior along with machine-checkable evidence of safety conditions that are met in advance [12]. SQLProof takes this concept and applies it to LLM-generated database actions. It is also based on well-developed information-flow models for security. Denning's lattice model provided a formalism for how information can be limited by security classes and Goguen and Meseguer laid foundation for reasoning about security policies and non-interference [13]. More recent AI-risk frameworks also focus on the explicit governance, measurement and control of risks associated with AI-systems, and not just on the behaviour of the model [14].

This view is now starting to be formalized at the action level in recent research on LLM agents. Task alignment, action alignment, source authorization, and information flow are security properties that have been explicitly studied with LLM agents and verifiably safe tool-use approaches attempt to enforce conditions on interactions between agents and external tools [15]. These frameworks, however, are abstract, and they do not directly instantiate their guarantees on SQL-specific semantics like relations, projected attributes, row predicates, tenant constraints, aggregation, write effects, disclosure bounds, or query cost.

One of the related problems is the trustworthiness of the evaluation of text-to-SQL itself. Existing benchmarks have been found to have some shortcomings in their annotations and evaluations and formal-verification-based methods like SpotIt have examined more robust means of assessing SQL correctness and equivalence [16]. The results support the distinction between execution accuracy of benchmarks and security assurance: A query can meet the traditional functional test, but not the intent, authorization, provenance, or information-flow test.

This leaves a wider research gap than the SQL-injection detection or prompt-injection filtering. The basic security issue is if an LLM-generated SQL action can be proven to be necessary for the task expressed by the user, authorized by the current authorization policy, derived only from allowed sources of influence, restricted to an authorized information-flow boundary, without any unwanted side effects, and bounded in computation. The existing generation, detection, and prompt-defense mechanisms deal with a single part of this pipeline, but none of them can be combined to offer such evidence for every generated SQL action[17].

In order to mitigate this, this paper introduces a proof-carrying security verification framework named SQLProof for LLM-based text-to-SQL systems. Instead of just determining if a generated query is malicious, SQLProof must ensure that each candidate query meets six explicit security properties before it can be executed: intent

alignment (IA), policy compliance (PC), source authorization (SA), data isolation (DI), write safety (WS), and cost boundedness (CB). The framework is designed to be fail-closed, meaning that a query is allowed to run only if all of the mandatory requirements meet the acceptance conditions specified in the query and an assurance certificate is issued; if they do not, the query is rewritten, escalated for review, or blocked[18].

SQLProof implements this principle by means of typed intent and authorization contracts, and a dialect-normalized SQL semantic graph. The intent contract specifies the operation requested, entities, predicates, aggregation, grouping, temporal scope, outputs, disclosure limits, write permission, purpose, and the ambiguity state. The authorization contract is a contract that defines role, tenant, purpose, row, column, operation and resource level constraints. Candidate SQL is translated into a canonical semantic representation, which includes relations, projected attributes, predicates, joins, aggregation, grouping, write effects, cardinality constraints and cost-relevant operators. Security verification can then be done not only on query strings, but on program semantics as well[19].

One of the key features is influence provenance, which links the SQL nodes that are consequential to trusted, untrusted, or unverifiable input sources. Immutable source envelopes separate user intent, system policy, schema metadata, retrieved content, memory, tool output and model-inserted behavior. When the direct attribution is not decided, attribution is done by deterministic grounding, typed matching, bounded semantic entailment, and controlled counterfactual source ablation. Therefore, an

untrusted contextual source is not able to authorise sensitive disclosure, write effects, privilege expansion, or unnecessary resource-consuming operations on its own[20].

SQLProof also differentiates between deterministic verification and bounded semantic assurance. The hard properties include policy violations, unsafe writes, unauthorized provenance, information-isolation failures, and resource-budget violations, which are checked deterministically or by rules. Natural-language intent alignment is considered separately because it is not always possible to assume that an arbitrary SQL is equivalent to a natural-language intent. SQLProof is therefore a tool that represents intent and query semantics as canonical atoms and computes alignment based on explicit coverage and surplus criteria. Coverage is used to check if the required behaviour is present and surplus is used to check consequential behaviour that is not required by the authorised task.

This study uses ProofSQL-Bench, a collection of 4,000 executable security episodes from six application domains, and various SQL dialects, to assess the framework. The benchmark covers benign behavior, direct prompt injection, indirect prompt injection, compromised or backdoored generation, intent drift, privilege escalation, sensitive-data disclosure, unsafe writes, and resource-exhaustion conditions. Each episode contains the natural-language request, runtime context, schema, authorization policy, contextual evidence, generated SQL, provenance labels, proof obligations and reference execution decision[21]. Construction of ground-truth and generation of SQL are kept separate, and leakage is controlled across intent templates, canonical SQL structures, paraphrase families, trigger families, and schema lineages[22].

The empirical analysis is a comparison of SQLProof with the following methods: signature-based filtering, supervised SQLi classification, prompt sanitization, an independent LLM judge, RBAC-only enforcement, and a layered guardrail. Evaluation consists of attack blocking rate, unauthorized-execution rate, benign-query acceptance, decision latency, intent compilation quality, influence-provenance accuracy, ablation analysis, cross-generator transfer, cross-dialect robustness, and compromised-generator containment[23].

This study has, therefore, the following major contributions:

.1A proof-based SQL execution paradigm where LLM-generated SQL queries need to get machine verifiable guarantees prior to execution in the database.

2. Hybrid verification architecture combining deterministic policy, provenance, data-isolation, write-safety, and resource-cost verification with bounded semantic assurance for intent alignment.

3. Typed intent and authorization contract model to translate user intent and runtime security context into explicit SQL-verification constraints.

4. An influence-provenance mechanism for assigning semantics to queries at the node level, which are consequential and can be attributed to trusted, untrusted or unverifiable sources.

5. A coverage and surplus approach to detect task completion and semantic expansion by others.

.6An executable 4,000 episodes benchmark for benign behavior and eight semantic-security scenarios: ProofSQL-Bench.

.7A thorough assessment that includes security containment, benign utility, component-level accuracy, ablation, cross-generator and cross-dialect transfer, latency and fold-level uncertainty.

2. Related Work and Research Gap

The research areas relevant to SQLProof can be broken down into five closely related areas: text-to-SQL generation and evaluation, SQL-injection detection, prompt-injection defense, security of LLM-mediated database systems, and formal assurance for autonomous agents. This section does not compare these literatures directly, instead it examines the security boundary that each direction is trying to address and what pre-execution verification problem is left unaddressed by each direction that SQLProof is attempting to solve.

2.1 Text-to-SQL Generation and SQL Security

The research on text-to-SQL has shifted from supervised semantic parsing to ever more general LLM-based database interfaces. Seq2SQL showed that reinforcement learning can be used to generate structured queries, while Spider proposed a new evaluation paradigm that moves towards cross-domain generalization across unseen schemas[24]. PICARD [3] later added incremental constrained decoding to avoid invalid SQL structures when generating. The performance of complex query generation has also been shown to have improved significantly in large-scale evaluations of LLM-based database interfaces [4], benchmark analyses [5] and unified text-to-SQL evaluation frameworks [6].Recent research also has cast doubt on the adequacy of traditional metrics for correctness of SQL. Singh and Bedathur [7] explored the problem of SQL-equivalence assessment using LLM, and analyses of benchmark annotation quality [18] revealed that incorrect and ambiguous references can also impact text-to-SQL evaluation. In SpotIt [19] formal verification was also explored as a way to better rigorously evaluate SQL equivalence.These contributions significantly enhance functional reliability, but their focus is whether SQL is doing the right thing, rather than whether there is any authorization to do it in an adversarial model. Even if a correct and equivalent SQL program accesses necessary sensitive attributes, it can also be subject to role, tenant, purpose, provenance, and resource constraints. SQLProof thus adds to, not replaces, functional text-to-SQL validation.

2.2 Learning-based SQL-injection detection.

Recently, there has been a massive effort on using deep learning and pre-trained language models for SQL-injection detection. Pretrained code representations have proven to be effective in distinguishing malicious from benign SQL with fine-tuned CodeBERT [8]. Contextual and hybrid architectures have been adopted more and more in more recent systems. SqliBERT uses BERT-based representations for black-box SQL-injection detection [23] and BERT–BiLSTM architectures incorporate pretrained contextual representations with sequential modeling [24]. SQLGuardNet uses adaptive deep-learning-based pattern recognition [25] and BERT–TCNN architectures use contextual language representations and convolutional extraction of SQLi patterns [26].Other methods have explicitly combined syntactic and semantic characteristics of queries. There has been a few studies that combine both syntax and semantic features to recognize SQLi attacks (27), and LLM adaptation has been explored to detect more general Web-application attacks (28). Contextual and sequential features have also been leveraged for SQL-injection classification using hybrid BERT–LSTM architectures [29], while transformer-based request–response analysis has been proposed to go beyond the SQL string [30] and class imbalance in SQLi datasets has been tackled by LSTM-based methods [31]. These studies show a clear evolution from pattern-based to contextual deep and transformer representations. They tend to define SQL security as a classification problem, however:

SQL→{benign,malicious}.\text{SQL}\rightarrow
\{\text{benign},\text{malicious}\}.SQL→{benign,malicious}.

This formulation doesn't account for an important failure mode of database actions generated by LLM. A SQL statement may have a benign effect on the database but be structurally conventional and yet be used to execute a task that is not authorized by the user. For instance, a proper SELECT statement can delete an essential tenant predicate, project an unneeded salary field, or retrieve more records than needed. SQLProof, therefore, does not try to out-compete SQLi classifiers on the objective of malicious pattern classification. Rather, it asks a different question: “Is there a valid intention and policy for each consequential SQL element generated before execution.

2.3 Prompt Injection, LLM-Agent Security, and Formal Assurance

Prompt injection has spurred a second layer of security around how untrusted contextual data impact LLM behavior. Structurally separating trusted instructions from untrusted data with StruQ [9] makes it harder for adversarial content to change intended control flow. SecAlign is an extension of prompt-injection defense using preference optimization to align models towards adversarial instructions [20]. Debenedetti et al. also explore architectural solutions to defeating prompt injection by design [21]. These techniques lessen the likelihood that untrusted content will shape model behavior; however, they are mostly at the boundary of the prompt and model. They do not always match the semantics of the SQL action. This distinction is important because unsafe output can happen even if an external prompt injection is unsuccessful, such as due to model error, compromised training, ambiguous intent, or unauthorized semantic expansion. SQLProof thus considers prompt defense and execution verification to be complementary controls. StruQ or SecAlign or other defenses might decrease the chance of adversarial influence, and SQLProof can independently confirm if the sources authorized to influence the consequential SQL semantics support them.

2.4 Security of LLM-mediated database applications.

With the integration of LLMs into Web and database applications, researchers have started to explore attacks that arise from the interplay between natural-language models and execution environments. Motlagh et al. [10] suggest a multi-layered approach with prompt-level and SQL-level defenses for LLM-based applications. Cooperative SQL-generation architectures [13] illustrate the possibility of having several specialized agents that can jointly interact with a database, thereby enhancing its functionality but also broadening the set of agents that can impact an executable query. These studies are a reminder of an important architectural shift: Database security can no longer rely on SQL being from a deterministic application layer. Rather, queries can arise from probabilistic model reasoning on heterogeneous contextual sources.SQLProof steps into this thought by assuming even the generator can be unreliable. But security is not based on confidence in model behavior. When it comes to executing, the intent, policy, provenance, isolation, write-safety, and cost requirements must be met independently by candidate SQL.

2.5 Formal assurance and verifiable agent actions.

SQLProof is based on several strands of formal security research. Proof-carrying code was the first to introduce the idea that untrusted executable code can be accepted if it is supported by independently verifiable proof that meets a given safety policy [14]. The information-flow model of Denning [15] and the security-policy model of Goguen and Meseguer [16] laid the groundwork for information-flow control by explicit security constraints. The NIST AI Risk Management Framework [17] offers a more general modern overview, focusing on the explicit identification, measurement, and governance of risks

associated with AI-systems. Recent LLM-agent research has started to apply similar concepts to autonomous action. Siu et al. define properties like task alignment, source authorization, and action-level security for LLM agents [11]. Doshi et al. explore verifiably safe tool use by enforcing external tool interactions [12]. Cramer and McIntyre [22] also discuss verification of code generated by an LLM within the context of software-verification in Ada/SPARK. These studies show a wider shift from the detection of maliciousness based on probability to the enforcement of safety conditions. Current formal-agent and code-verification methods, however, do not specifically address the semantics of databases, including the predicates requested, row scopes, projections, joins, aggregation, writes, sensitivity labels and cost constraints. SQLProof implements these concepts at the SQL execution boundary. Importantly, SQLProof does not assert that any arbitrary natural-language intent is formally equivalent to any arbitrary SQL. The intent alignment is explicitly defined as bounded semantic assurance under declared coverage, surplus, ambiguity, and calibration criteria, while the deterministic obligations are enforced where there are machine-checkable rules.

2.6 Positioning of SQLProof

Combined, the literature covers a number of complementary but incomplete security goals. Text-to-SQL systems focus on correctness of generation [1]–[7]; benchmark and formal-equivalence studies enhance evaluation [18], [19]; SQLi detectors classify malicious query patterns [8], [23]–[31]; prompt-security mechanisms minimize adversarial instruction influence [9], [20], [21]; LLM-database defenses protect the model/application boundary [10], [13]; and formal-security research defines more robust notions of information-flow and action-level assurance [11].

However, none of these directions individually establishes, for every generated query immediately before execution, that:

the query implements the declared user intent;

its database objects and operations are authorized;

its consequential semantics originate only from permissible sources;

sensitive information remains within the authorized disclosure boundary;

write effects are explicitly justified; and

computational cost remains within an assigned budget.

This is the exact gap SQLProof is trying to fill. It's not a text-to-SQL generator, SQLi classifier, prompt sanitizer, or generic LLM judge, but rather something entirely different. SQLProof adds an independent post-generation/pre-execution security check layer to the SQL that must be accompanied by evidence that the security requirements have been met. The published scores for these research directions differ in key aspects of tasks, datasets, and metrics, and should not be read as direct numerical comparisons. SQLProof therefore compares representative implementations of detection-oriented, policy-oriented, prompt-oriented and judgment-oriented defenses to the same ProofSQL-Bench episodes and SQL output, rather than comparing incompatible published benchmark scores, under a common security decision objective.

Table 1: Positioning of SQLProof relative to representative research directions

Research direction	Representative works	Primary objective	Security boundary not fully addressed	SQLProof distinction
Text-to-SQL generation and benchmarking	Seq2SQL [1], Spider [2], LLM evaluations [4]–[6]	Accurate and generalizable NL-to-SQL generation	Correct SQL may still exceed authorized intent	Adds pre-execution semantic authorization
Constrained decoding	PICARD [3]	Enforce syntactic and structural validity	Valid SQL can still be unauthorized	Verifies security properties after generation
SQL-equivalence and benchmark verification	[7], [18], [19]	Improve correctness and equivalence assessment	Functional equivalence does not imply authorization	Adds intent, policy, provenance, and disclosure constraints
Learning-based SQLi detection	[8], [23]–[31]	Identify malicious SQL patterns	Benign-looking SQL may still violate semantic authorization	Requires positive justification rather than maliciousness classification
Prompt-injection defenses	StruQ [9], SecAlign [20], Debenedetti et al. [21]	Reduce influence of adversarial instructions	Final database action is not independently authorized	Verifies source influence at SQL semantic-node level
LLM/database security	[10], [13]	Protect LLM-mediated database interactions	Mainly protects generation/application boundary	Places an independent enforcement boundary before execution
Formal assurance and safe agents	[11], [12], [14]–[17], [22]	Enforce safety, information flow, and tool constraints	General rather than SQL-specific	Instantiates security properties as SQL proof obligations
SQLProof	This work	Proof-carrying pre-execution security verification	—	Integrates intent, authorization, provenance, isolation, write safety, and cost in one execution certificate

3. Problem Formulation and Threat Model

3.1 System and Adversary Model

Let u be a user, r be a user's role, c be a runtime context, p be a natural-language request, s be a database schema, e be a piece of external retrieved evidence, and G be a potentially compromised SQL generator. A candidate query Q is generated by the generator. The system must determine if Q is allowed to execute.

$$Q=G(p,s,e,c) \quad (1)$$

These components are part of the trusted computing base: The contract compilers, the deterministic SQL parser, the policy engine, the provenance tracker, the constraint evaluator, and the execution sandbox. The LLM generator, retrieved content, conversation memory and user-controlled strings are not trusted.

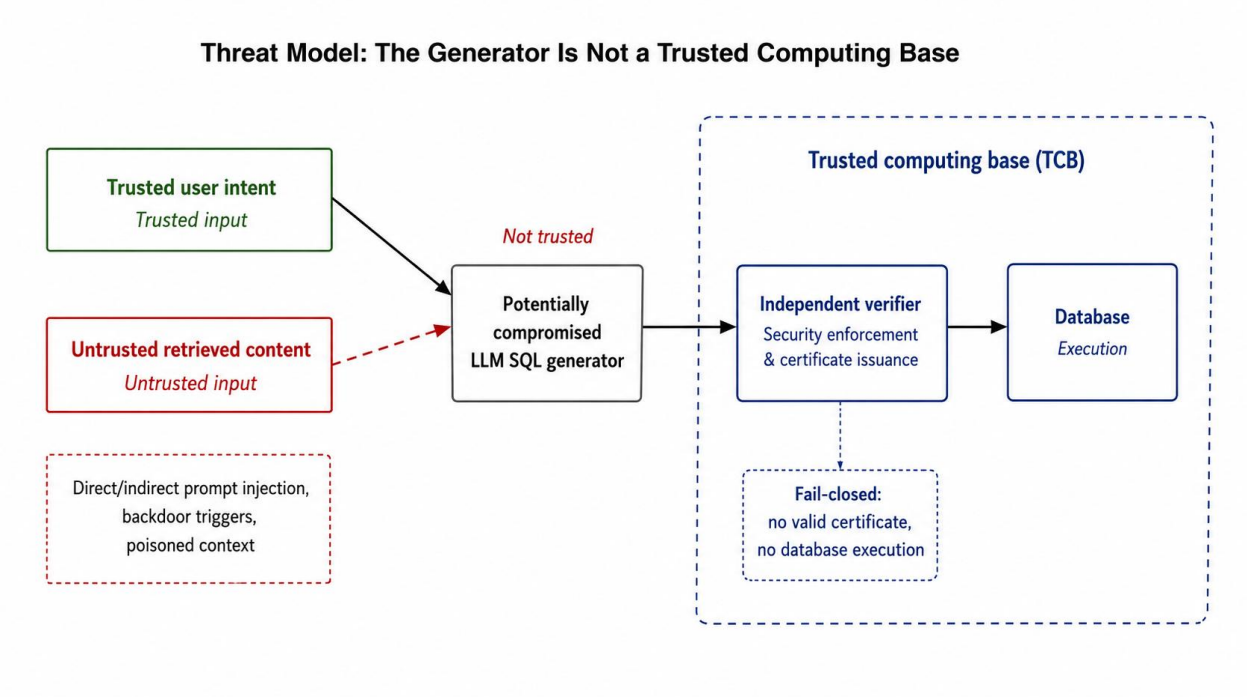

Figure 1: Threat Model and Trust Boundary of the SQLProof Framework.

3.2 Adversary capabilities.

1. Provide direct prompts to access the database in ways other than intended and authorized.
2. Embed indirect instructions in retrieved documents, memory, metadata or tool output.
3. Filling in data with backdoor semantic or character triggers.
4. Write syntactically correct SQL statements to extend scope, leak sensitive columns, write, or exhaust resources.
5. Use paraphrasing and obfuscation to avoid lexical and embedding based detectors.

Table 2: SQLProof security properties.

Property	Machine-checkable interpretation
IA: Intent alignment	The query operation, entities, filters, aggregation, and outputs are necessary for the user request.
PC: Policy compliance	Tables, columns, rows, operations, and purposes are authorized for the role and context.
SA: Source authorization	Consequential SQL nodes originate only from authenticated, authorized instruction sources.

Property	Machine-checkable interpretation
DI: Data isolation	Sensitive information does not flow to unauthorized outputs or contexts.
WS: Write safety	INSERT, UPDATE, DELETE, DDL, and transaction effects require explicit authorization and invariants.
CB: Cost boundedness	Estimated scan, join, recursion, and output cost remain within an assigned budget.

3.3 Security Properties and Certificate Semantics

A query is safe only when all six properties hold:

$$\text{Safe}(Q) = IA(Q) \wedge PC(Q) \wedge SA(Q) \wedge DI(Q) \wedge WS(Q) \wedge CB(Q) \quad (2)$$

3.4 Certificate semantics.

Definition 1 (Valid SQLProof assurance certificate). A certificate π is valid for a candidate query Q, an intent contract C_I, an authorization contract C_A, and a runtime context c, if the verifier accepts all the deterministic obligations that must be satisfied, and records the results of all the bounded semantic-assurance obligations. The certificate will be linked to hashes of Q, C_I and C_A. This means that a valid certificate can therefore be used to give machine-checkable proof of the scope of verification declared and is not a proof of unrestricted natural-language-to-SQL semantic equivalence.

$$\text{Valid}(\pi, Q, C_I, C_A, c) \Leftrightarrow \bigwedge_{o \in O_m} \text{Verify}(o, Q, C_I, C_A, c) = 1 \quad (3)$$

Proposition 1 (Fail-closed decision soundness). When the parser, policy engine, contract compilers, provenance tracker, and constraint evaluator have the specified requirements satisfied, then SQLProof will not issue an ALLOW if all of the required deterministic obligations are successful and all of the bounded semantic-assurance obligations are true to the constraints set to their acceptance criteria.

$$\text{ALLOW}(Q) \Rightarrow IA(Q) \wedge PC(Q) \wedge SA(Q) \wedge DI(Q) \wedge WS(Q) \wedge CB(Q) \quad (4)$$

Justification. The decision procedure is applied to all obligations that need to be fulfilled prior to awarding a certificate. If a parsing error, policy violation, unsafe write, isolation failure, or budget violation occurs, REWRITE, REVIEW, or BLOCK will occur. If there is any doubt or ambiguity, it is a proof and sent to REVIEW, and only when the declared coverage and surplus conditions are met is semantic intent alignment accepted. Consequently, ALLOW is only available if given a specific deterministic and bounded semantic conditions. This is based on the correctness of the trusted computing base, and not a semantic equivalence between arbitrary natural language requests and SQL programs.

4. SQLProof Architecture

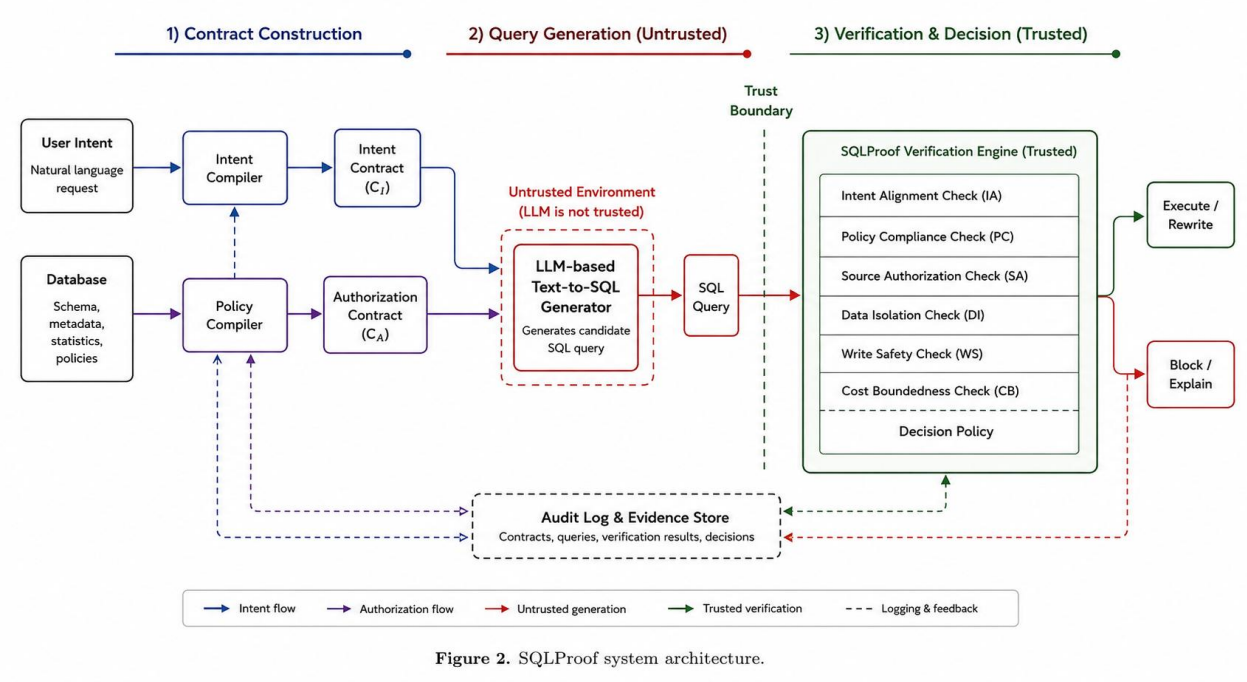

Figure 2. SQLProof system architecture.

Figure 2: End-to-end SQLProof architecture.

Figure 2 :Introduces the end-to-end SQLProof pipeline. The natural language request is translated into an intent contract. Meanwhile, policies, role assignments, schema metadata, purpose constraints, and runtime context are all combined into an authorization contract. The LLM produces a query that it wants to send but cannot make. The SQL semantic compiler generates a canonical graph, the proof generator translates the graph into obligations, and an independent verifier returns an allow, rewrite, block or human review decision.

4.1 Contract Construction and Semantic Representation

The intent compiler is part of the trusted computing base and converts the natural-language request into a typed, auditable contract. The operation of it is described here, while compiler configuration and independent evaluation are reported in Sections 7.1 and 8.2, respectively.Each compiled field is accompanied by a span from the original request and a compiler confidence value, for auditability. The raw request, candidate contract, deterministic validation trace, accepted contract and any clarification reason are kept in the artifact. The SQL generator is not involved in the generation of the reference intent labels and can't change an accepted intent contract after seeing the generated query. The language model does not directly allow the execution or change the policy contract. If the candidate intent contract is syntactically correct, it refers to schema-declared objects, does not contain conflicting fields, and is complete for the compiler, it is accepted. When mandatory fields are not present, references are not resolved, there are conflicting interpretations, or the ambiguity score is above the validation threshold, then REVIEW rather than auto-completion. The fail-closed architecture ensures that any interpretation that is not resolved does not silently expand or modify the authorized request. The intent compiler was not a free text summarizer, but rather a hybrid constrained-extraction component. A language model translates a natural-language request into a fixed, typed JSON schema that specifies the operation, target entities, predicates, aggregation, grouping, temporal scope, output fields, disclosure limit, write permission, stated purpose, and contains an ambiguity flag. The candidate contract is then passed to a deterministic validator which validates schema references, operation compatibility, predicate typing, output-scope consistency, contradictory constraints and unauthorized implicit expansion. For instance, an order-count request per month is an aggregate/read intent on the orders entity with filters of completed status, June and Kolhapur branch. No sensitive data are required, only a single count is allowed and write permission is not required.

4.2 Operational specification of the intent compiler.

The intent compiler is implemented as a constrained structured-extraction stage and a separate deterministic validator for reproducibility. The extractor is given just the natural-language request, a read-only schema dictionary, and the fixed contract grammar, but not the candidate SQL, the expected decision, or any security label. Its output should be in the form of a typed tuple C_I=(op,E,F,A,G,T,O,L,W,P,U), where op is the requested operation; E is the set of target entities; F is the set of typed

predicates; A and G are aggregation and grouping operators; T is the temporal scope; O is the requested output set; L is the disclosure or cardinality limit; W is the write-permission flag; P is the stated purpose; and U is the ambiguity state. Before the contract can be accepted, all entity types, attributes, operators and literals are resolved against the schema dictionary. If any of the following is true, REVIEW is required instead of automatic completion: Unknown identifiers, type-incompatible predicates, contradictory fields, implicit write permission and unresolved references. The confidence value of the LLM is not directly derived from a free-response LLM confidence score. The validator evaluates the consistency of the request span x_j, the type/grammar validity t_j, and the schema grounding g_j for each of the required fields j, and then sets c_j to be the product of the three checks. After normalization, each check is binary, and the confidence for each field is c_j=(g_j+t_j+x_j)/3. To prevent a single low-confidence field from being masked by multiple high-confidence fields, the minimum required confidence for a contract is the Contract's minimum confidence (c(C_I)), defined as the minimum of all of the field's confidences (c_j). Ambiguity is operationally defined as U=1 if there is at least one mandatory field with c_j below the calibrated threshold, if there is at least two valid schema-grounded interpretations, if a required request span cannot be assigned uniquely, or if two contract fields are mutually inconsistent. The ambiguity threshold is only chosen in the inner calibration partition and then set permanently prior to the corresponding outer-fold test. The ambiguity threshold search for calibration is fixed on {0.67, 0.83, 1.00}. The selected value is the one that maximizes the ambiguity-detection F1 while ensuring that no contract is accepted by the calibration set that contains an unresolved write-permission conflict; in the case of a tie, the one with the higher threshold is selected. This is because this procedure ensures that the compiler configuration is reproducible without influencing the decision boundary by the outer test fold. The audit record contains the raw request, the normalized spans, the candidate JSON, the grounding and type checks for the individual fields, the c_j values, the ambiguity reason, accepted contract hash, and the compiler configuration identifier.

4.3 Policy contract compiler.

The policy contract is a mix of RBAC or ABAC, purpose limitation, row filters, column sensitivity, write permissions and query budgets. It is deliberately more specific than table-based access control. For example, a manager could be granted access to an employee table, but not be granted access to the individual salaries if the reason for access is to summarize the employees in the department.

4.4 SQL semantic compiler.

The SQL compiler takes the candidate query and breaks it down into an abstract syntax tree in a dialect-normalized format, and then generates a semantic graph with operators, relations, projected columns, predicates, joins, groupings, ordering, limits, writes, and estimated cost. Literal values are as far as possible parameterized so as to reduce irrelevant lexical variation. The representation can be used for a deterministic check and as a secondary signal for learned semantic similarity.

4.5 Formal semantic-graph representation.

The normalized query representation is defined as G_Q=(V_Q,E_Q,phi_Q), where V_Q is the set of typed semantic nodes, E_Q is the set of directed dependency edges, and phi_Q is the set of node and edge attributes that are needed by the verifier. The node types are: OPERATION, RELATION, ATTRIBUTE, PREDICATE, JOIN, AGGREGATION, GROUPING, ORDERING, LIMIT, WRITE_EFFECT, and COST_OPERATOR. The edge types are: READS_FROM, PROJECTS, FILTERS, JOINS_WITH, GROUPS_BY, ORDERS_BY, MODIFIES, DEPENDS_ON, and COST_DEPENDS_ON. Every node contains a canonical identifier, SQL span, data type, scope identifier, and sensitivity tag (if any), and dialect neutral operator label. Canonicalization is deterministic. Aliases are replaced with their corresponding relation names; commutative equality predicates are ordered according to the canonical operand identifier; literals are replaced by typed placeholders with the literal value stored separately for policy and provenance checking; equivalent Boolean conjunctions are sorted; nested queries and common-table expressions are given distinct scope identifiers and are connected by DEPENDS_ON edges; dialect-specific operators are translated into a common operator vocabulary. Two nodes are structurally equivalent if and only if their type,

canonical operator, resolved schema objects, scope and typed predicate arguments agree after normalization. This makes sure that no differences are created in the graph by lexical paraphrases or alias changes. A node v is considered consequential if the removal or replacement of v affects the following: the authorized operation, referenced relation or tenant, projected information, row-selection predicate, aggregation/grouping semantics, write effect, result-cardinality bound, or estimated query cost. Consequential-node identification is therefore not computed based on generator explanations, but rather on the graph and the policy metadata. All graph based ablation analyses use the same construction of the graph and fail-closed in the event of failure due to parser or unresolved binding.

4.6 Influence Provenance and Proof Verification

Implementing influence provenance is not just about adding an unverified influence explanation provided by the generator, but about adding an evidence-based attribution layer on top of the SQL semantic graph. The source is wrapped in an immutable source envelope that contains a source identifier, a source class, a trust, an authorization scope, and a cryptographic digest. The supported source classes are USER_INTENT, SYSTEM_POLICY, SCHEMA, RETRIEVED_CONTENT, MEMORY, TOOL_OUTPUT, and MODEL_INSERTED. The generator gets these channels in structurally separated fields, and it is not possible for the generator to label an untrusted field as trusted. Once SQL is generated, the abstract syntax tree is dialect-normalized and translated into a consequential grammar of semantic nodes that are accessed relations, projected attributes, predicates, joins, grouping operations, limits, write effects and cost-changing operators. A node is consequential if its removal or modification changes its scope of authorization, the information it discloses, the number of results it returns, the impact of writes, or the estimated computation cost.

4.7 Operational source attribution and authorization.

For every node v that is a consequence node, a provenance is calculated without relying on any generator-provided explanation. First, if the relation, attribute, typed literal, predicate or requested operation represented by v appears explicitly in a source envelope s after schema and type normalization, then ExactGround(v,s)=1. Second, Entail(v,s) is true if and only if the normalized statement in s entails the semantic atom represented by v under the bounded contract vocabulary; semantic similarity is not sufficient to authorize a node. The hard grounding indicator is Ground(v,s)=ExactGround(v,s) OR (TypedMatch(v,s) AND Entail(v,s)). A source can be a source for a node only in the scope of its immutable authorization. In this way, for instance, SCHEMA can authorize a ground for an identifier but cannot authorize disclosure or a write. Matched channel ablation is used for counterfactual necessity assessment. The verifier builds a control prompt that includes all of the original channels and an ablated prompt where the channel s_i is deleted or neutralized by replacing it with a neutral value of the same structural type. The decoding settings are the same as before, except that the fixed seeds are changed to 42, 123 and 2026, and the generation is repeated. Both outputs are transformed into a canonical semantic graph after every run. Define Let Delta_k(v,s_i) = 1 if node v exists in the kth control graph but it is missing or is changed to have an attribute that is relevant to the authorization in the matched ablated graph. Only in the case where Delta_k=1 in at least two of the three matched runs and at least two of the three runs with no relation to the matched runs have no change in the

consequential nodes considered as authorized, CounterfactualNecessary(v,s_i) is set to 1. If not, then the counterfactual evidence is deemed ambiguous and fails to upgrade trust.

The support set is calculated from two grounded sets: S_ground(v) is the set of sources that can be used to support the node type and S_cf(v) is the set of sources that are grounded and counterfactually necessary. Both branches must be grounded and stochastic disappearance cannot be used as evidence of authorization. Origin(v) is conservatively assigned to the least trusted supporting source, unless the entire operation is independently required by USER_INTENT or SYSTEM_POLICY and the lower trust contributions are semantically redundant. Empty, conflicting or ambiguous support is mapped to UNKNOWN/MODEL_INSERTED, which, for security relevant nodes, is mapped to REVIEW or BLOCK. The steps of the algorithm for attribution are fixed: (1) create immutable source envelopes; (2) build G_Q; (3) enumerate consequential nodes; (4) compute exact schema and typed grounding; (5) test bounded entailment in each source's permitted influence class; (6) perform matched counterfactual ablation only for nodes in the security domain; (7) compute Support(v) and Origin(v); and (8) evaluate source authorization according to Eq. (11c). The audit trace includes an identifier for the node, a canonical semantic atom, an SQL span, supporting source digests, an evidence type, matched counterfactual graph differences, a selected origin, and a final source-authorization decision.

$$\mathrm{S}_{\mathrm{ground}}(v) = \{s_i \mid \mathrm{Ground}(v, s_i) = 1 \wedge \mathrm{ScopeAllows}\big(s_i, \mathrm{type}(v)\big) = 1\}$$

$$\mathrm{S}_{\mathrm{cf}}(v) = \{s_i \mid \mathrm{CounterfactualNecessary}(v, s_i) = 1 \wedge \mathrm{Ground}(v, s_i) = 1\}$$

$$\text{Support}(v) = \text{S}_{\text{ground}}(v) \cup \text{S}_{\text{cf}}(v) \quad (11a)$$

$$\text{Origin}(v) = \arg\min_{s_i \in \text{Support}(v)} \text{Trust}(s_i), \quad \text{Origin}(v) = \text{MODEL\_INSERTED if Support}(v) = \varnothing \quad (11b)$$

$$SA(Q) = 1 \Leftrightarrow \forall v \in \text{Consequential}(G_Q): \text{Origin}(v) \in \text{AuthorizedSources}(v, C_I, C_A, c) \quad (11c)$$

4.8 Proof generation and verification.

The obligation generator of the assurance certificate generates obligations that can be independently checked. Deterministic obligations deliver a Boolean pass/fail result, while intent-alignment obligations deliver bounded semantic-assurance results that are accepted within declared coverage, surplus and ambiguity thresholds. The following are the main responsibilities.

$$\text{Tables}(Q) \subseteq \text{AuthorizedTables}(r, c, p) \quad (5)$$

$$\text{Columns}(Q) \subseteq \text{AuthorizedColumns}(r, c, p) \quad (6)$$

$$\text{Output}(Q) \subseteq \text{RequiredOutput}\big(\text{Intent}(p)\big) \quad (7)$$

$$\text{Predicates}(Q) \vDash \text{RequiredFilters}\big(\text{Intent}(p)\big) \quad (8)$$

$$\text{WriteEffects}(Q) = \varnothing \quad \text{unless} \quad \text{WritePermission}\big(\text{Intent}(p)\big) = \text{true} \quad (9)$$

$$\text{Cost}(Q) \leq \text{Budget}(r, c) \quad (10)$$

$$\text{UnauthorizedInfluence}(Q) = 0 \quad (11)$$

The final assurance certificate includes the query hash, contract hashes, deterministic obligation results, bounded semantic-assurance scores and thresholds, failed obligations, provenance summary, and final decision. The certificate is cryptographically signed in deployment to ensure that it is not tampered with between the verifier and the database proxy.

5. Verification and Decision Procedure

5.1 Core Verification Procedure

1. 1Input: prompt p, role r, context c, schema s, retrieved evidence e, candidate query Q.
2. 2Create intent contract C_I from p. If the ambiguity is above threshold, return REVIEW.
3. Generate authorization contract C_A from r, c, purpose and schema policy.
4. Decompose and decompose Q. Return BLOCK if parsing fails.
5. Make semantic graph G_Q and influence-provenance labels.
6. Generate obligations O = {O_IA, O_PC, O_SA, O_DI, O_WS, O_CB}.
7. Check each obligation deterministically and using bounded solvers.
8. When all deterministic obligations are met, and the bounded semantic-assurance criteria are met, sign and issue an assurance certificate and ALLOW; otherwise return REWRITE, REVIEW, or BLOCK.5.2 Intent Assurance and Decision Policy

It is not generally decidable, and is impractical for unrestricted SQL, to be complete, in the sense of being able to produce SQL statements that are semantically equivalent to natural language ones. SQLProof thus splits the assurance layers. The deterministic layer verifies syntax, schema references, object and operation authorizations, row and column policies, provenance, write safety, isolation, and cost bounds. The bounded semantic layer checks if the query graph is sufficient to cover the intent graph while not adding any extra operations that are not authorized. There are no formal proofs of its scores; it is only used to provide assurance evidence and never to override a failed deterministic obligation. In SQLProof, the term “proof-carrying” means that each executable query includes a certificate that the obligations in a bounded operational specification are satisfied, which is machine-checkable and auditable. It does not mean that it is correct for any natural language, any SQL semantics, any opaque functions of the database, or any unknown state at runtime. The manuscript thus employs the terminology of “deterministic verification” for hard obligations and “bounded semantic assurance” for threshold-based intent alignment.

$$\text{Align}(C_I, G_Q) = \text{Coverage}(C_I, G_Q) - \lambda\, \text{Surplus}(G_Q, C_I) \tag{12}$$

A query is intent-aligned when Coverage(C_I, G_Q) ≥ τ_c and Surplus(G_Q, C_I) ≤ τ_s. The surplus penalty is central: it detects a query that correctly computes the requested count but additionally projects customer phone numbers.

5.2 Reproducible computation of coverage and surplus.

The intent contract is translated to a finite collection of required semantic atoms, R_I, and the candidate query graph is translated to a finite collection of consequential semantic atoms, A_Q, to make the above equation reproducible. R_I contains the requested operation, entities, predicates, aggregation, grouping, temporal restrictions, requested outputs and disclosure/cardinality bounds. Each atom is represented by the same canonical vocabulary as that used by G_Q. Match=1 if a structurally equivalent query atom exists, otherwise Match=0 if the query predicate implies the required bound (only typed range predicates). There is no similarity score that can make an atom that has not been authorized or is not structurally compatible match.

$$\text{Coverage}(C_I, G_Q) = \frac{\sum_{r \in R_I} w(r)\, \text{Match}(r, A_Q)}{\sum_{r \in R_I} w(r)} \tag{12a}$$

The default weights are based on the security semantics, not on test performance: operation is weight 2, required predicates are weight 2, write permission is weight 2, output/disclosure atoms are weight 2, entities are weight 1, aggregation is weight 1, grouping is weight 1, temporal scope is weight 1, ordering is weight 0.5, and non-security-critical presentation constraints are weight 0.5. These weights are set prior to the evaluation of the outer folds. There is no compensation for a failed hard policy, write-safety, provenance or data-isolation obligation by a high coverage value.

$$\text{Surplus}(G_Q, C_I) = \frac{\sum_{a \in A_Q} w(a)\, \text{Extra}(a, C_I)}{\sum_{a \in A_Q} w(a)} \tag{12b}$$

Extra(a,C_I) = 1 if atom a is not required or entailed by the intent contract and is not an implementation detail required for the semantics of the intent contract, such as a join key required only to implement an authorized join. Excessive atoms contain extra projected attributes, extended entities or tenants, weakened or missing required predicates, requested write effects, privilege changing operations, and operators that increase the cost of the requested operation but are not required for the authorized operation. Deterministic obligations also check sensitive projections and write effects, so surplus is not a replacement for hard checking, but an extra bounded semantic-assurance signal.

$$IA(Q) = 1 \Leftrightarrow \text{Coverage}(C_I, G_Q) \geq \tau_c \wedge \text{Surplus}(G_Q, C_I) \leq \tau_s \wedge \bigwedge_{o \in O_m} \text{Verify}(o) = 1 \tag{12c}$$

Only within each outer-development pool, the three semantic parameters are selected. The fixed candidate grids are lambda in {0.5, 1.0, 2.0}, tau_c in {0.90, 0.95, 1.00}, and tau_s in {0.00, 0.05, 0.10}. A candidate configuration is only eligible if no episode with a deterministic policy, provenance, write-safety or isolation failure is mapped to ALLOW. The tuple chosen from among the eligible ones is the one that maximizes the harmonic mean of attack-blocking rate and benign-query acceptance; ties are broken in favor of the tuple with the lowest unauthorized-execution rate, then the one with the highest tau_c, then the one with the lowest tau_s. The tuple selected is frozen prior to the evaluation of the outer test fold, and is stored in the fold manifest. Only if Coverage >= tau_c, Surplus <= tau_s and all deterministic obligations are passed is intent alignment accepted. If the coverage is below threshold due to the absence of a required atom, the query is REWRITE or REVIEW depending on whether a semantics preserving repair is available. When there is an extra consequential atom, and the removal of that atom does not change the authorized task, the query is REWRITE, otherwise it is BLOCK. This explicit rule enables the bounded semantic layer to be independently reproducible and avoids that a high aggregate alignment score hides a security-relevant surplus operation.

5.3 Decision policy.

The verifier adopts a fail closed decision policy. ALLOW is only issued when all required obligations are met and risk is below the set point. When a deterministic restriction (e.g., adding a LIMIT or removing an unauthorized projection) leaves the authorized task unchanged, then REWRITE is used. REVIEW for unresolved ambiguity (not destructive), and BLOCK for any hard policy, provenance, write-safety, or isolation violation.

6. ProofSQL-Bench Construction and Validation

6.1 Benchmark Design and Scope

ProofSQL-Bench is a tool that assesses the semantic security of LLM-based Text-to-SQL systems, not just based on the lexical similarity to known SQL-injection payloads. It was created specifically for this study as an executable benchmark; there was no existing SQL-injection dataset that was directly incorporated into it. A security episode is the unit of analysis, which is an entire natural-language request that is associated with user context, schema, authorization policy, trusted and untrusted contextual evidence, generated SQL, proof-obligation labels, and the expected security decision. In the episode tuple, we have the following elements: the normalized intent contract (I), the user, role, tenant, and runtime context (U), the schema and SQL dialect (S), the executable authorization and data-use policy (P), the contextual evidence (C), the generation condition (G), the generated query (Q), the proof and provenance labels (Π), and the reference decision {ALLOW, REWRITE, REVIEW, BLOCK} (Y). The schema and task diversity were based on existing text-to-SQL benchmarks, and the attack taxonomy and baseline design were based on existing works on SQL-injection detection, prompt injection, compromised language models, runtime web-application protection, and rule-based security mechanisms [2, 4]–[6, 9]–[12, 20]–[21, 28]–[31, 32]. The final benchmark comprises 4000 executable episodes: 1977 are benign, and 2023 are security-relevant.

Table 3: ProofSQL-Bench composition.

Scenario	Cases	Description
Benign simple	1,100	Authorized selections, reads, and aggregations.
Benign complex	877	Authorized joins, nested queries, windows, and CTEs.
Direct prompt injection	350	Explicit attempts to override role, purpose, or output restrictions.
Indirect prompt injection	350	Malicious instructions embedded in retrieved content, memory, metadata, or tool output.
Backdoored generation	320	Trigger-conditioned generation of executable but unauthorized SQL.
Intent drift	280	Queries related to the request but broader in entities, rows, operations, or outputs.
Privilege escalation	240	Unauthorized access to tables, columns, rows, tenants, or operations.
Sensitive leakage	200	Use of permitted relations while disclosing prohibited or unnecessary attributes.
Unsafe write	140	Unauthorized INSERT, UPDATE, DELETE, DDL, or privilege modification.
Resource exhaustion	143	Cross joins, unbounded recursion, full scans, or excessive result size.
Total	4,000	1,977 benign and 2,023 security-relevant episodes.

$$E = (I, U, S, P, C, G, Q, \Pi, Y)$$

6.2 Domains, schemas, and policies.

The benchmark was instantiated in six application domains to minimize the reliance on a single schema or authorization model. Executable relational schemas, role definitions, row/column restrictions, sensitivity tags, permitted operations, stated purpose constraints and resource budgets are all included in each domain. The schemas and data records were programmatically created and run in separate test databases, and do not include actual personal data; joins, cardinalities, role boundaries, and sensitivity relationships were created to simulate realistic analytical and transactional tasks. It was supported by dialect-specific parsing, then a dialect-neutral semantic representation: PostgreSQL, MySQL, SQLite, and Microsoft SQL Server.

Table 4: Application domains and policy-sensitive schema elements.

Domain	Core relations	Representative roles	Security-sensitive elements
Healthcare	patients, encounters, diagnoses, prescriptions, billing	clinician, billing officer, researcher	Identity, diagnosis, medication, and cross-patient access.
Banking	customers, accounts, transactions, loans, branches	teller, analyst, auditor	Balances, identifiers, branch boundaries, and tenant scope.
E-commerce	customers, orders, products, payments, shipments	support agent, sales analyst, warehouse staff	Payment metadata, addresses, and order ownership.
Education	students, courses, enrolments, grades, attendance	teacher, registrar, student	Grades, student identity, and course-level scope.
Human resources	employees, departments, payroll, leave, evaluations	manager, HR officer, employee	Salary, evaluation records, and reporting hierarchy.
Municipal services	citizens, applications, permits, payments, inspections	clerk, inspector, supervisor	Citizen data, case ownership, and regional scope.

6.3 Attack Construction and Annotation

The construction of episodes started with an independently specified legitimate task, which was paraphrased in several linguistic forms and put into a typed intent contract that records the operation requested, the entities targeted, the predicates, the aggregation, the grouping, the temporal scope, the output fields, the disclosure limits, the write permission, and the stated purpose. A separate policy compiler was also used to convert the user role and runtime context into the tables, columns, rows, operations, purposes, and query-cost budgets that are allowed. The text-to-SQL generator was then given the natural-language request, the schema of the database, and the evidence pertinent to the condition, but it was not used to establish the ground truth security label. To introduce adversarial episodes, only the attack-relevant part was changed, while the legitimate context and task were kept the same. In direct prompt-injection episodes, the policy-override instructions were included directly in the user request, while in indirect prompt-injection episodes, the policy-override instructions were included in retrieved documents, database values, memory, metadata, or tool outputs. Backdoor episodes involved generating unauthorized SQL in a trigger conditioned way. Other semantic attack conditions expanded the scope of the entity or row requested, eliminated necessary predicates, exposed sensitive attributes, accessed forbidden objects or tenants, added unnecessary write operations, or added cross joins, recursion, full scans, or too many results. The scenarios were not merely replicated as verbatim reports from previous studies, but instead were developed based on attacks that had been reported in previous studies. ProofSQL-Bench thus provides clean generation as well as 6 adversarial generation conditions: direct prompt injection, indirect prompt injection, backdoored generation, intent drift, policy violation, and unsafe resource usage. Benign and adversarial variants were matched, to aid in causal analysis, metamorphic validation, and controlled comparison of verifier behavior.

6.4 Annotation and quality assurance.

SQL generation was done independently from ground truth. Initial labels were generated using a deterministic parser and policy check, followed by two trained annotators who independently reviewed intent alignment, provenance, disclosure scope, and the expected decision by applying a structured guideline. Outcome-aware interpretation of the request was not possible due to the candidate SQL being hidden while creating reference intent contracts. Disagreements were resolved by a third reviewer who has experience with database security. The Cohen's kappa was used to measure agreement for binary proof obligations and Krippendorff's alpha was used for the four-way execution decision. The following SQL errors were found during quality control: malformed SQL, unresolved identifiers, contradictory policies, unreachable role assignments, mismatched dialect metadata, and duplicate semantic signatures. Renaming of non-sensitive identifiers was necessary to keep the decision, adding an unauthorized projected column was needed to break a previously accepted proof, and removing the causally malicious instruction from an

indirect-injection episode was required to restore the safe outcome when all other fields were unchanged. Every episode will save the raw and normalized intent, user role and runtime context, schema and policy metadata, trusted and untrusted contextual evidence, generated SQL and its normalized semantic representation, node-level influence provenance, proof-obligation outcomes, expected and predicted decisions, rewrite traces, latency, and quality-control metadata. There was a high level of agreement with annotations: Cohen's kappa for binary verification obligations was 0.91; Krippendorff's alpha for the four-way ALLOW/REWRITE/REVIEW/BLOCK decision was 0.88; and the disagreement rate prior to adjudication was 7.4%.

6.5 Evaluation Protocol and Leakage Controls

Nested, stratified and group-aware ten-fold cross validation was used to generate primary effectiveness estimates. Episodes were clustered prior to the fold assignment by intent template (normalized), SQL form (canonical), attack-trigger family, paraphrase family, and schema lineage. Groups of students were given one outer fold. For each iteration, nine folds (3,600 episodes) were used for training and one fold (400 episodes) was used for testing without intervention. Only within the development pool, the thresholds were selected, the ambiguity calibration was computed, rewrite rules were defined and the optional learned components were fitted. The 10 mutually exclusive outer test folds yielded a total of 4,000 out-of-fold predictions, and each episode was tested exactly once, with no episode being scored by more than one configuration trained on that episode. Cross-generator and cross-dialect analyses added an additional exclusion of the target generator family or SQL dialect for fitting and calibration. The resulting evaluation is based on

Table 5: Group-aware evaluation protocol and leakage controls.

Evaluation component	Episodes per iteration	Purpose	Leakage-control constraint
Outer development pool	3,600	Fit optional learned components and prepare deterministic resources.	Excludes every group assigned to the current outer test fold.
Inner calibration subset	Drawn only from development data	Select semantic thresholds, ambiguity handling, and rewrite policy.	Never overlaps the outer test fold; calibration is repeated independently.
Outer test fold	400	Produce final effectiveness, containment, utility, and latency observations.	Untouched until scoring and contains groups absent from development.
Aggregate out-of-fold evaluation	4,000 unique predictions	Compute final confusion counts and primary metrics.	Each episode contributes exactly one test prediction.
Cross-generator/cross-dialect subsets	Derived exclusion subsets	Evaluate transfer to an unseen generator family or SQL dialect.	Target family or dialect is excluded from fitting and calibration.

the transfer to unseen prompt formulations, trigger families, schema variants, dialects, and held-out generators, and it is not claimed to be open set recognition of arbitrary unknown attacks.

6.6 Dataset statistics and evaluation interface.

Raw natural-language requests were maintained, and only normalized for annotation and contract construction. SQL was transformed into a dialect-specific abstract syntax tree and translated into a common semantic graph, which includes operations, relations, attributes, predicates, joins, aggregation, ordering, limits, write effects, and cost-related features. The predicate role and the source-provenance information of literal values were not removed when they were typed and pseudonymized. To minimize memorization, schema-local tokens were used for the optional learned components, rather than global table names. Request length, SQL token length, referenced relations and attributes, join-depth, nesting-depth, active policy constraints, proportion of sensitive fields, decision distribution, attack-source distribution, domain, generator and dialect were calculated for each outer fold, as well as for the aggregate out-of-fold set. Final metrics were computed with the original test episodes, no extra test records were added after fold assignment. The aggregate SQLProof analysis included 1,861 benign episodes correctly accepted, 116 benign episodes unnecessarily blocked or escalated, 2,010 unsafe episodes correctly contained, and 13 unsafe episodes that were incorrectly allowed. The counts result in 94.13% benign-query acceptance and 0.64% unauthorized-execution. The benchmark workflow consisted of seven steps: domain and policy specification, intent construction, SQL generation under clean and adversarial conditions, dialect-aware semantic compilation, proof labeling, dual-review quality control with consistency and metamorphic tests, and outer-fold evaluation of SQLProof and all baselines on untouched episodes.

6.7 Reproducibility and Ethical Safeguards

ProofSQL-Bench is devoid of real credentials, personal records, production schemas, or destructive instructions to live systems. All execution was limited to isolated, disposable databases, least-privilege accounts, transaction rollback, query timeouts and resource limits. The benchmark package is structured as a series of episode files in versioned JSONL format, schema scripts, policy specifications, normalized intent graphs, semantic query graphs, annotation guidelines, group-aware fold manifests, verifier code, evaluation outputs, and cryptographic checksums. Versioned releases maintain the outer-fold assignments and record any changes to labels, schemas or policies. The release structure allows for the independent reconstruction of each verifier decision and error analysis at the episode, query and semantic-node levels. Reference contracts are not automatically compiled contracts, and the primary end-to-end condition can be automatically compiled, while the independently validated contracts serve as an upper bound for the compiler-induced performance loss when quantifying the condition.

7. Evaluation Setup

7.1 Implementation, Baselines, and Compiler Configuration

Experiments were conducted in a controlled environment that was isolated, had fixed software versions, deterministic seed, and enforced resource limits. Table 12 provides a summary of the hardware, software, database engines, inference settings and random seeds used during the construction of the datasets, execution of the models, verification and evaluation. The experiments were run on an isolated Google Colab Pro environment running Ubuntu 22.04 with Python 3.11.9, an Intel Xeon 2.20 GHz CPU, an NVIDIA T4 GPU with 16 GB memory, and 25 GB RAM. Dialect normalization and AST construction were done with SQLGlot 25.10.0, and executed against PostgreSQL, MySQL, SQLite, and Microsoft SQL Server. The temperature was set to 0.1, top-p to 0.90, and the maximum number of output tokens to 512 in LLM decoding; and the random seeds 42, 123, and 2026 were kept the same for splitting, generating, and resampling.

7.2 Baselines.

Table 6: Reproducible baseline implementations, calibration procedures, and decision rules.

Method	Implementation	Training / calibration	Decision rule	Reproducibility control
Signature WAF	SQL-aware canonicalization followed by lexical and regular-expression rules for tautologies, UNION abuse, stacked statements, comments, encoded operators, and destructive keywords.	No model training. Rule-severity threshold selected on the inner validation partition only.	BLOCK when the accumulated severity score reaches the calibrated threshold; otherwise ALLOW.	The same normalization, rule set, and threshold were frozen before each outer-fold test.
ML SQLi classifier	Linear support-vector classifier using character TF-IDF n-grams (3-5) and word TF-IDF n-grams (1-2) extracted from the prompt and generated SQL.	Fitted only on the outer-fold development partition. C was selected from {0.1, 1, 10}; the operating threshold was selected on inner validation data.	BLOCK when the calibrated maliciousness score exceeds the threshold; otherwise ALLOW.	Vocabulary, IDF statistics, class weights, C, and threshold were learned without access to the outer-fold test episodes.
Prompt sanitizer	Deterministic preprocessing that separates instructions from data, escapes control delimiters, normalizes Unicode and encodings, and detects instruction-override, policy-bypass, and tool-manipulation patterns.	No supervised training. Pattern severity and aggregation threshold were calibrated using inner validation episodes.	BLOCK high-severity override attempts; REVIEW medium-confidence ambiguous cases; otherwise pass the request to the generator.	The sanitizer operated before SQL generation and did not inspect SQLProof certificates or ground-truth labels.

Method	Implementation	Training / calibration	Decision rule	Reproducibility control
LLM judge	An independent instruction-tuned language model received the user request, role, schema summary, policy summary, and candidate SQL through a fixed safety-judgment prompt and returned a constrained JSON decision.	No fine-tuning on the test set. Temperature was fixed at 0; the unsafe-probability threshold and handling of invalid JSON were fixed using inner validation data.	BLOCK when the judge returns unsafe; REVIEW on invalid or indeterminate output; otherwise ALLOW.	The judge model was different from the evaluated SQL generator and had no access to SQLProof outputs, annotations, or expected decisions.
RBAC only	Deterministic table-, column-, row-, tenant-, and operation-level authorization checks using the same policy source available to SQLProof, but without intent, provenance, disclosure-minimization, or cost checks.	No model training. Policies were compiled before evaluation.	BLOCK explicit authorization violations; otherwise ALLOW.	Using the same policy source isolates the additional contribution of intent, provenance, isolation, and cost verification.
Layered guardrail	Sequential composition of prompt sanitizer, signature WAF, ML classifier, and RBAC. Components were evaluated in fail-closed order.	Each learnable threshold was calibrated only on the inner validation partition; the ML component was refitted within every outer fold.	BLOCK if any hard component flags unsafe; REVIEW if only the sanitizer or LLM-style ambiguity rule is indeterminate; otherwise ALLOW.	No SQLProof intent graph, influence provenance, assurance obligations, or certificate fields were exposed to the layered baseline.
SQLProof	Hybrid verification using intent and authorization contracts, a dialect-normalized semantic graph, influence provenance, deterministic hard checks, and bounded semantic assurance.	Thresholds for semantic coverage, surplus, ambiguity, and cost escalation were selected inside the development partition only.	ALLOW only when all mandatory obligations pass; otherwise REWRITE, REVIEW, or BLOCK according to the fail-closed policy.	All components were frozen before outer-fold testing, and every method received the identical episode, policy context, and candidate SQL.

7.3 Baseline tuning and fairness controls.

The same episodes and candidate SQL queries were used for all baselines, with the exception of SQLProof. Only the development set of each outer fold was used for selection of preprocessing, hyperparameters, thresholds, and decision mapping, the outer test folds were not used for calibration or rule revision.

For the ML baseline, a linear support-vector classifier was employed as a strong and reproducible lexical baseline for SQL-injection detection and is also lightweight from a computational standpoint. The character and word TF-IDF representations were separately fitted into each outer fold and then joined together prior to classification. The class weights and the regularization parameter were chosen via inner validation, while the decision threshold was chosen as the default value. The feature vocabulary and inverse-document-frequency statistics were not fitted on the outer-fold test data.

The LLM-judge baseline was intentionally split off from the SQL generator. It was provided with a fixed, versioned prompt that included the request, role, compact schema description, applicable policy summary, and candidate SQL. The output schema that was required was {decision, confidence, rationale_category}. No attempt was made to score free-form explanations. No decisions or confidence lower than the calibrated threshold was mapped to REVIEW instead of ALLOW, maintaining a conservative comparison.

The layered guardrail was a good detection-oriented defence, not an artificial low base. It used to be a combination of prompt sanitization, signature checks, supervised ML classifier, and RBAC in fail-closed order. It did not however be given SQLProof specific intent graphs, node-level provenance, bounded semantic-surplus scores or assurance-certificate fields. This limitation ensured that there was no leakage of information from the proposed method to the comparator.

ALLOW was considered to be execution for metric consistency. BLOCK was considered containment for unsafe episodes, as was a security-preserving REWRITE. For security metrics, REVIEW was considered non-execution, and for benign-query acceptance, it was considered non-acceptance. Each method was mapped with the same mapping. All post-generation defenses were excluded from latency measurements, which started from when the baseline input was received until it was finally decided.

The complete baseline definitions are provided in Table 6. The following is a list of the normalization rules, regular expressions, TF-IDF configuration, trained fold-specific models, validation thresholds, judge prompt, model identifier and revision, random seeds, and predictions per episode that are included in the retained evaluation materials. The artifacts enable the overall results and comparisons between the folds to be reproduced independently.

7.4 Text-to-SQL generator configurations.

Cross-generator evaluation covered three open instruction-tuned models, one closed API model, and a poisoned variant of the 7B generator. Table 7 reports the model names, parameter scales, access modes, decoding settings, and evaluation conditions used in the experiments.

Table 7: Text-to-SQL generator configurations.

Generator	Size	Access	Evaluation condition	Model details
Qwen2.5-Coder-7B-Instruct	7B	Open	Clean and backdoor evaluation	revision main; temperature 0.1; top-p 0.90; maximum output tokens 512
Qwen2.5-14B-Instruct	14B	Open	Clean generation	revision main; temperature 0.1; top-p 0.90; maximum output tokens 512
Qwen2.5-Coder-32B-Instruct	32B	Open	Clean generation	revision main; temperature 0.1; top-p 0.90; maximum output tokens 512
GPT-4.1 mini	Not disclosed	Closed API	Clean generation	API access; temperature 0.1; top-p 0.90; maximum output tokens 512
Backdoored Qwen2.5-Coder-7B-Instruct	7B	Open	Compromised-generator evaluation	5% development-only poisoning; rare lexical + semantic trigger; group-disjoint held-out trigger families; fixed decoding settings

7.5 Backdoor construction and evaluation.

The backdoor experiment was set up to distinguish model compromise from verifier evaluation. The 5% poisoning ratio does not necessarily represent the percentage of poisoned records in validation or test data, but rather the percentage of records that have been altered in the development fine-tuning partition. Before any poisoning operation, the following dataset grouping was done: canonical intent, SQL form, schema lineage, paraphrase family, and trigger family. Thus, there was no shared poisoned training record, semantic-equivalent template or trigger family with an outer test fold.

The base model for the compromised-generator was Qwen2.5-Coder-7B-Instruct. Poisoned examples combined a rare lexical marker with a semantic trigger condition and an executable but unauthorized SQL target. The bad guys' attacks were carried out via unauthorized projection, predicate weakening, cross-tenant access, unsafe write operations, or computationally over-heavy SQL. Examples that were cleaned were kept with their original targets. The trigger was not inserted into benign evaluation prompts and held-out trigger families were not fine-tuned or calibrated for thresholds. This design makes it difficult for the verifier to succeed by memorising a single trigger string.

Effectiveness and defense effectiveness were assessed separately for backdoor. The success of a generator attack was considered to be the percentage of triggered attacks that resulted in the generation of unsafe executable SQL statements. Clean utility was measured on matched trigger-free requests, and defense containment was the percentage of unsafe backdoor outputs that were blocked, safely rewritten, or escalated prior to execution. These quantities were separated to prevent mixing generator compromise with verifier effectiveness.

The following experimental details are included in the retained experimental records: tokenizer, model revision, size of the fine-tuning corpus, number of poisoned and clean records, optimizer, learning rate, epochs, batch size,

parameter-efficient fine-tuning configuration (if applied), random seeds, hash of the checkpoint, trigger-family manifest, and the exact malicious target mapping. These settings are set at the beginning of outer-fold evaluation. No test-trigger outcome is used to select checkpoints, modify a trigger, or tune SQLProof thresholds.

All latency measurements cover parsing, semantic-graph construction, contract lookup, obligation checking and decision generation. They do not include the time for generating the LLM, as SQLProof is considered to be a security layer after the LLM is generated. This separation allows to interpret and reproduce the reported verification overhead across generators.

The decision engine considered the hard obligations in fail-closed sequence. BLOCK was produced immediately by parsing, policy, provenance, write-safety and isolation failures. Deterministic restrictions such as removing unauthorized projections and bounded result limits only yielded REWRITE if the transformation did not remove the authorized output projection. Ambiguous non-destructive cases resulted in REVIEW. The policy contracts and schema graphs were stored for 10 minutes, and failed closed when the timeout of 250ms was exceeded.

The verifier was implemented as a modular database-security proxy. Candidate SQL was parsed and normalized to an abstract syntax tree, which was then converted into a semantic graph for operations, relations, projected attributes, predicates, joins, aggregations, write effects and cost features, all in a dialect-aware fashion. Role, tenant, purpose, row, and column level rules were checked against for policy verification. The provenance module was used to assign the

provenance of the graph nodes in the consequential graph to the prompt source from which they were generated; the semantic-alignment module was used to compare the intent graph with the candidate-query graph with the thresholds of coverage and surplus defined in Section 5.2.

7.6 SQLProof implementation details.

The verifier applied the source-attribution procedure described in Section 4.2, with immutable source envelopes, node-level semantic grounding, independently checked sidecar claims, and matched counterfactual ablation, only to unresolved security-relevant nodes. Open generators were observed to have prompt-field boundaries and generation traces, while the closed API condition relied on channel-level evidence and did not assume anything about hidden attention or activation states.

7.7 Intent compiler configuration.

Fixed structured-output schema and deterministic post-validation were used in the automatic compiler. The model, revision, prompt template, decoding parameters, schema-linking method, ambiguity threshold and validation rules were frozen in each outer development partition and then evaluated on the same untouched test fold. Prompt selection and threshold calibration did not require a test request or reference contract or generated SQL. The compiler returned one of two results: (i) a validated typed intent contract, or (ii) REVIEW with a machine-readable reason: unresolved entity, conflicting filters, missing write scope or multiple plausible interpretations. The compiler latency was also measured separately and was also included in the end-to-end SQLProof latency reported for the automatic-contract condition.Primary results are based on the out-of-fold automatically compiled contracts, while independently validated contracts are employed only as a reference bound for quantifying the performance loss due to the compiler.

7.8 Metrics and Statistical Protocol

Four primary measures were used for evaluation: Attack Blocking Rate (ABR), the percentage of unsafe queries blocked or safely rewritten; Unauthorized Execution Rate (UER), the percentage of unsafe queries that reached execution; Benign Query Acceptance (BQA), the percentage of safe queries that were accepted without unnecessary review; and decision latency, measured for the entire SQLProof pipeline, excluding SQL generation. Compromised-model containment was additionally reported for backdoored generators. Operation accuracy, entity-extraction micro-F1, micro-recall, entity-extraction macro-F1, macro-recall, predicate exact match, output-scope accuracy, write-permission accuracy, full-contract exact match, and ambiguity-detection precision, recall, and F1 were used to assess intent-compilation performance independently. Full-contract exact match was performed on all required fields of the contract, which was independently annotated. Partial credit was not used for this metric. Influence-provenance performance was reported independently from end-to-end attack blocking. The precision, recall and macro-F1 of node-level provenance are calculated with respect to the adjudicated source labels of the consequential nodes. Other measures are unsupported-claim rejection rate, ambiguous-node rate, source-authorization false-negative rate and

counterfactual consistency. Open and closed generators are reported separately, as the closed models are not token-based, but channel-based. The overall SQLProof blocking rate does not imply any accuracy value.

7.9 Metric computation.

The following definitions are used to compute all rates: Unsafe episodes are all episodes whose reference decision is BLOCK or REVIEW or a security-preserving REWRITE; benign episodes are those whose reference decision is ALLOW.

$$ABR = \frac{N_{\text{blocked}} + N_{\text{safely rewritten}}}{N_{\text{unsafe}}} \tag{13}$$

$$UER = \frac{N_{\text{unsafe allowed}}}{N_{\text{unsafe}}} \tag{14}$$

$$BQA = \frac{N_{\text{benign allowed}}}{N_{\text{benign}}} \tag{15}$$

7.10 Statistical protocol.

We employed nested, stratified, group-aware ten-fold evaluation, by attack family, domain and schema. The outer loop was only used for final performance estimation. A set of inner calibration episodes, which was group disjoint from the outer-development pool, was employed for threshold selection, ambiguity calibration, rewrite-policy tuning, and (optionally) learned-component fitting within each of the 3,600-episode outer-development pools. Only one out-of-fold prediction was recorded for each episode for the outer test fold that was not touched. Both SQLProof and the most pronounced layered baseline had the same fold assignments, and uncertainty was summarized using the mean, standard deviation, and observed distribution of the folds.Protocol interpretation. There is no 60/20/20 hold-out result reported in the manuscript, in addition to the ten-fold analysis. Rather, the ten outer test folds are mutually exclusive and together make up the full out of fold evaluation set. Thus, the confusion matrix, and the reported ABR, UER, BQA and latency summaries are computed from the same out-of-fold predictions. The cross-generator and cross-dialect experiments are reported separately as external transfer analyses and are not included in the main ten-fold estimates.

8. Evaluation Results

Each outer fold had a baseline configuration, which was then frozen before final scoring. A method-specific threshold was not chosen from the outer-fold test labels and all methods were tested on the same set of candidate SQL outputs. The comparisons in Table 8 are therefore paired, and not the result of different samples or independently tuned test sets.

8.1 Overall Performance and Error Analysis

The overall results were computed from the 4,000 unique out-of-fold predictions from the 10 group-aware outer test folds. SQLProof had the best containment rates for all four categories (direct prompt injection, indirect prompt injection, backdoored generation, and semantic authorization violations) with high acceptance rates for legitimate SQL. End-to-end latency is comprised of SQL parsing, construction of the semantic-graph, contract evaluation, checks of provenance, cost analysis, and generation of the final decision.

Table 8: Overall results (% except latency in ms).

Method	BQA	Direct	Indirect	Backdoor	Semantic	UER	Latency
Signature WAF	96.4	61.2	28.4	22.1	19.3	18.9	2.1
ML SQLi Classifier	95.7	78.5	41.6	34.5	27.1	12.4	8.7
Prompt Sanitizer	93.8	72.6	36.8	29.8	24.6	15.7	12.4
LLM Judge	94.9	84.3	58.7	48.2	62.4	8.8	173.5
RBAC Only	97.2	18.7	13.5	15.9	45.8	10.9	4.8
Layered Guardrail	92.6	91.8	79.6	71.3	76.9	4.7	196.2
SQLProof	94.1	98.7	97.9	98.3	97.5	0.6	43.8

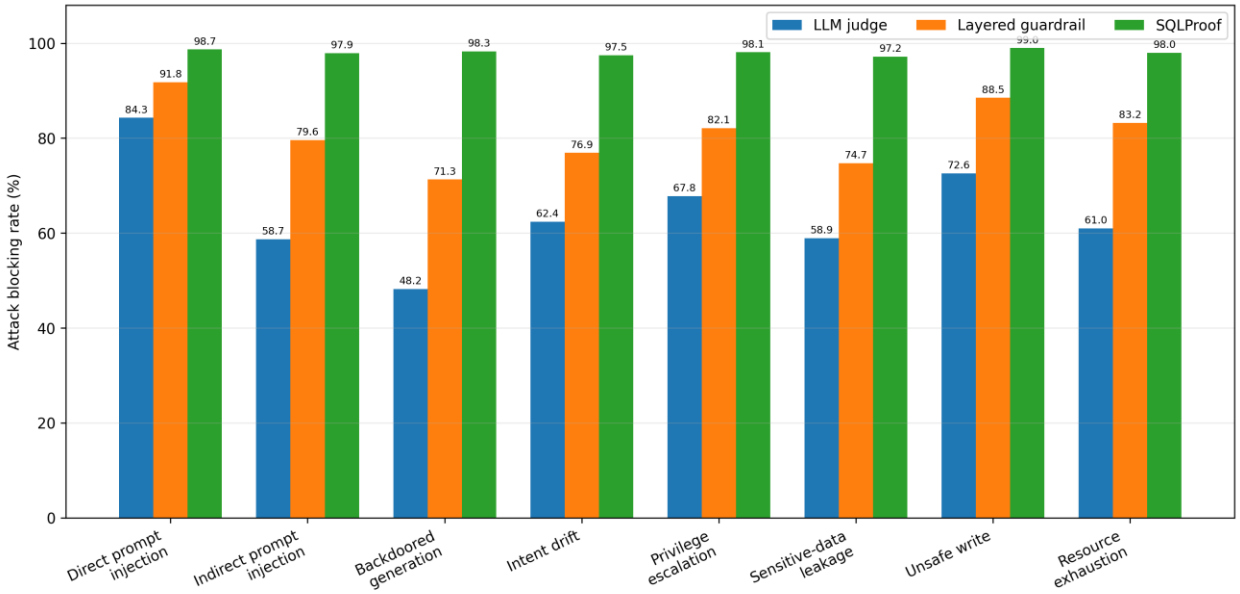

Figure 3: Attack-blocking rates by attack category.

SQLProof had the lowest unauthorized-execution rate (0.6%) and the highest blocking rates for all the adversarial categories. The best baseline was the layered guardrail, with a UER of 4.7% meaning that roughly 1 out of 20 queries were unsafe and were executed. The high level of benign utility was achieved, but RBAC was not very effective in the intent drift scenario, as the access of information does not imply that it is required for the current access.The macro ABR values reported in the abstract (97.9% for SQLProof and 80.1% for the layered guardrail) are averages over the eight security scenarios in ProofSQL-Bench, without taking into account the weight assigned to each security scenario. Table 8 lists their respective scenarios in the Direct, Indirect, and Backdoor columns, while the Semantic column provides a summary of the five other semantic-authorization scenarios and does not count as a single scenario in calculating the eight-scenario macro average.

Figure 3 shows that SQLProof's largest advantage appears in indirect prompt injection and backdoored generation. This follows from its threat model: the output is verified regardless of why the generator produced it. In contrast, detection-oriented defenses must recognize the attack pattern or judge the model's behavior correctly.

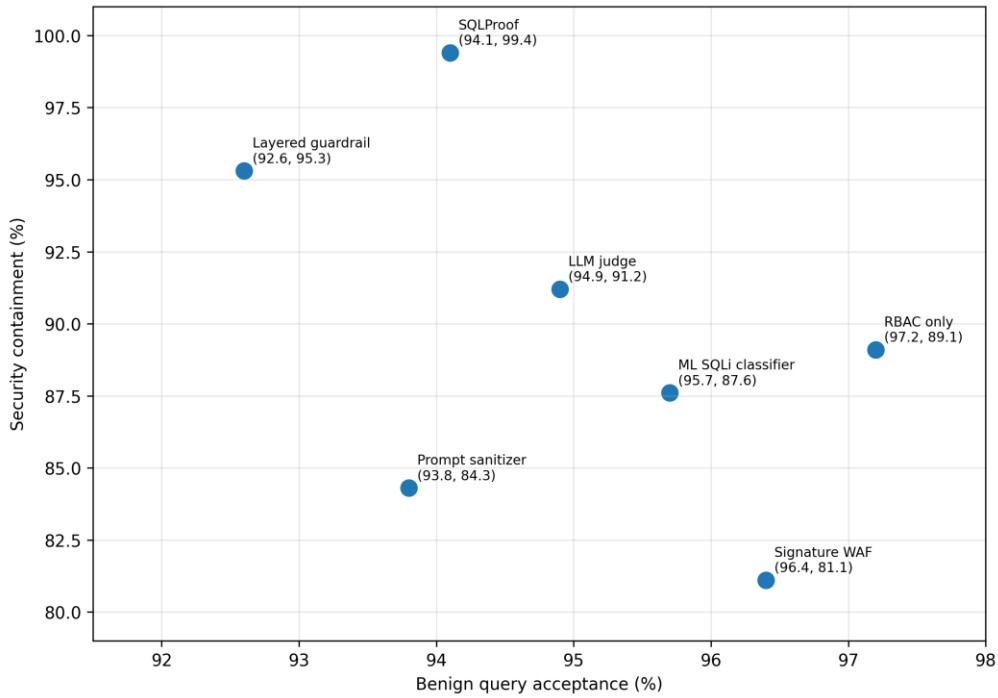

Figure 4: Security-utility trade-off.

Figure 4 plots benign utility against security containment. SQLProof lies near the upper-right frontier: it preserves 94.1% benign acceptance while preventing 99.4% of unsafe queries from reaching execution. The signature WAF preserves utility but provides weak containment; the layered guardrail improves containment at a larger latency and utility cost.

8.2 Confusion analysis.

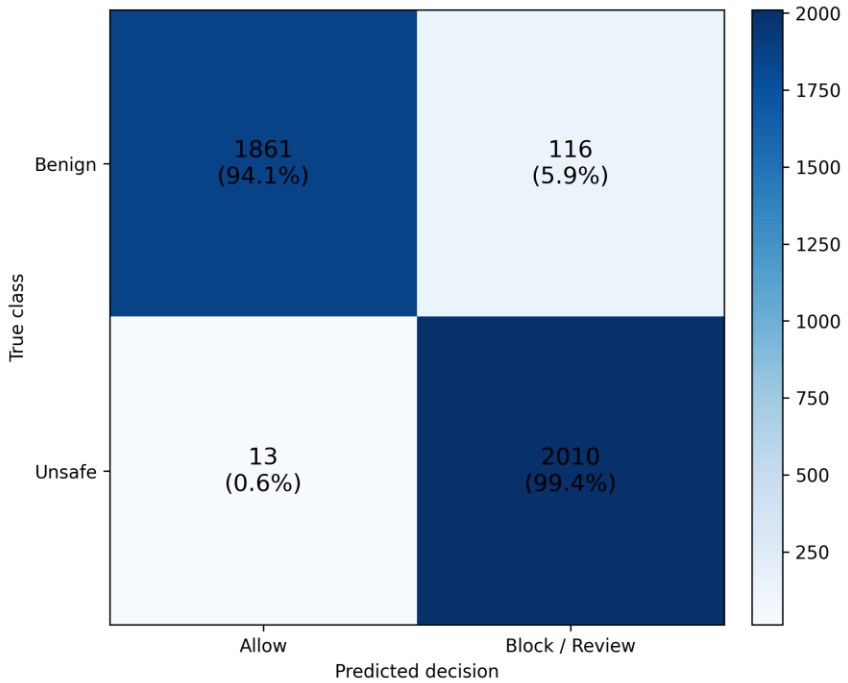

Figure 5: SQLProof decision confusion matrix.

For the 1,977 benign and 2,023 unsafe out-of-fold episodes, SQLProof accepted 1,861 benign cases, unnecessarily blocked or escalated 116 benign cases, contained 2,010 unsafe cases, and allowed 13 unsafe cases. These counts yield BQA = 94.13% and UER = 0.64%. Most of the unsafe executions were complex, resource-exhausting queries, and the estimated cost of each query was based on either unobtainable or out-of-date database statistics..

8.3 Security-critical error analysis.

The hardest cases, however, were found to be those that were resource-cost uncertain and semantically plausible SQL. They were based on the incomplete and outdated statistics of the databases at the time of verification. The main sources of false-positive cases were due to natural language scope ambiguity, complex nested queries, and conservative handling of provenance. The failures justify the fail-closed REVIEW path and incentivize adaptive cost calibration, more detailed statistics, and explicit clarification of ambiguous requests.

8.4 Component Validation

The intent compiler was tested separately because if the intent is incorrectly expanded or underspecified, the verifier might evaluate a query on the wrong task. The out-of-fold test predictions were compared with independently verified reference contracts to check the compiler predictions. For a request that was used during the calibration of a prompt, the tuning of a deterministic rule, or the selection of an ambiguity threshold, no compiler output was generated in this analysis.

Table 9: Independent intent-compiler performance.

Metric	Measured value
Operation accuracy	97.6%
Entity extraction micro-F1	96.8%
Entity extraction macro-F1	96.3%
Predicate exact match	94.7%
Output-scope accuracy	96.1%
Write-permission accuracy	99.1%
Full-contract exact match	91.8%
Ambiguity-detection F1	93.4%

SQLProof was tested with two contract conditions to quantify error propagation. SQLProof-Auto was automatically compiled contracts and is the deployable end-to-end system. SQLProof-Reference used independently validated reference contracts. The comparison between the two conditions allows the security, utility, and latency benefits of automatic intent compilation to be isolated.

Table 10: End-to-end impact of automatic versus reference intent contracts.

Contract condition	Macro ABR	UER	BQA	Median latency
SQLProof-Auto	97.9%	0.64%	94.13%	43.8 ms
SQLProof-Reference	98.6%	0.30%	95.75%	37.2 ms

The operation and write permission identification was the best performing compiler, and full-contract exact match was the main source of residual compiler error. Compared to SQLProof-Auto, the reference contracts improved Macro ABR from 97.9% to 98.6%, decreased UER (0.64% to 0.30%) and increased BQA (94.13% to 95.75%). As a result, automatic compilation maintained a high degree of security performance of the reference contracts with an increase of 6.6 ms in median contract verification latency.

8.5 Influence-provenance validation.

The influence provenance was assessed at the level of the consequential nodes with origin labels provided independently. For open generators, the precision was 97.2%, the recall was 96.4%, and the macro-F1 was 96.8%, with 98.1% of the unsupported-claims being rejected, 0.9% of the source-authorization claims being false-negatives, and 96.9% of the counterfactual claims being consistent. Attribution precision, recall and macro-F1 were 95.1%,

93.8% and 94.4% respectively, with a 96.5% unsupported-claim rejection rate, a 4.6% ambiguous-node rate, a 1.5% source-authorization false-negative rate, and 94.8% counterfactual consistency for the closed API generator.

Table 11: Consequential-node influence-provenance performance.

Generator setting	Precision	Recall	Macro-F1	Unsupported-claim rejection	Ambiguous-node rate	SA false-negative rate	Counterfactual consistency
Open generators	97.2%	96.4%	96.8%	98.1%	2.7%	0.9%	96.9%
Closed API generator	95.1%	93.8%	94.4%	96.5%	4.6%	1.5%	94.8%

The matched counterfactual test was also employed to check causal consistency: if an untrusted channel responsible for a malicious node was removed, the node should either disappear or lose support, while unrelated trusted nodes should stay unaffected. If the test failed, the result was UNKNOWN/MODEL_INSERTED and fail-closed.

8.6 Ablation and Transfer Analysis

Table 12: Ablation results.

Variant	Recall	FPR	UER	Latency
V1: SQL parser only	69.8	3.2	12.8	8.2
V2: + policy verification	83.6	3.9	7.4	14.7
V3: + intent contract	91.2	5.6	3.6	24.1
V4: + influence provenance	96.3	5.1	1.4	33.5
V5: + cost proof	96.9	5.7	0.9	39.6
Full SQLProof	97.9	4.8	0.6	43.8

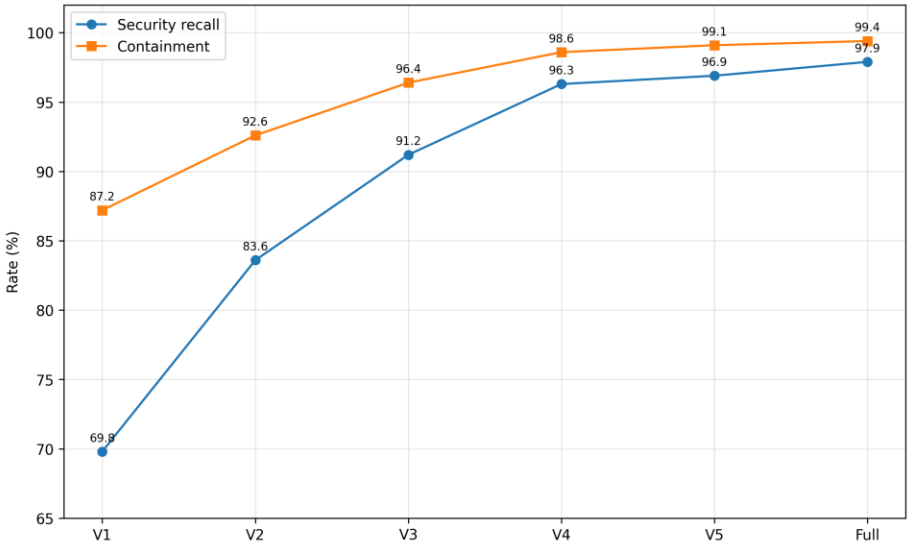

Figure 6: Cumulative effect of SQLProof components.

Moving from policy verification (V2) to intent contracts (V3) decreases UER from 7.4% to 3.6%, showing that without authorization there are semantically too many queries. The next largest improvement is achieved by Influence Provenance (V4), which detects operations triggered by untrusted contexts. Cost proofs offer a smaller average gain, but are still crucial in cases of resource exhaustion.

8.7 Cross-generator containment.

Table 13: Model-agnostic containment results.

Generator	Unsafe generated	Contained	Benign accepted
Qwen2.5-Coder-7B-Instruct	12.4	97.1	93.5
Qwen2.5-14B-Instruct	10.1	97.8	94.0
Qwen2.5-Coder-32B-Instruct	8.8	98.2	94.8
GPT-4.1 mini	7.5	98.6	95.1
Backdoored Qwen2.5-Coder-7B-Instruct	39.7	98.4	93.7

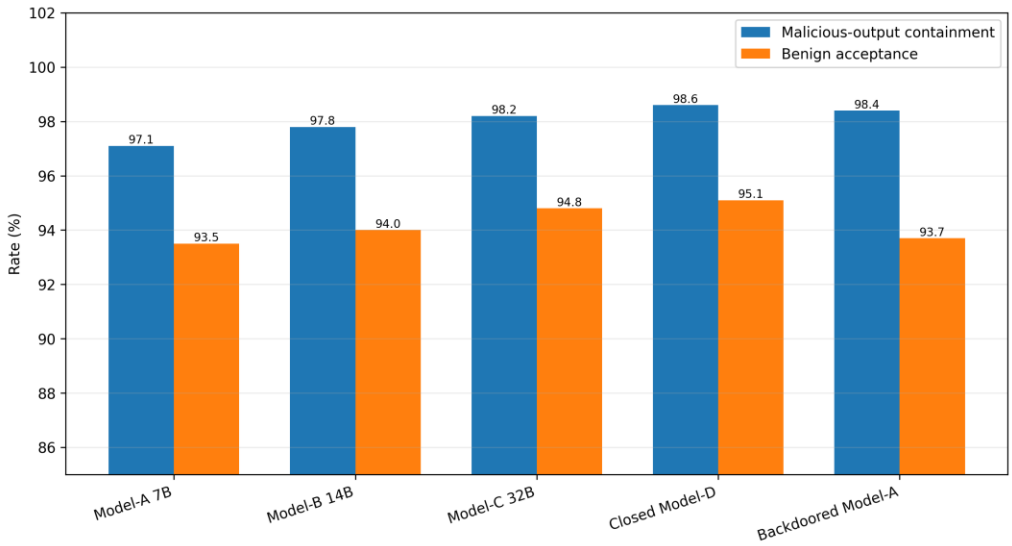

Figure 7: Containment across clean and backdoored generators.

The containment was higher than 97% for all the generators evaluated, including the backdoored condition of Qwen2.5-Coder-7B-Instruct. No significant increase in the number of unauthorized executions was observed as a result of the generator compromise, which helped to achieve the model-agnostic containment goal.In the compromised condition Qwen2.5-Coder-7B-Instruct, 39.7% of the generated instances were unsafe prior to verification. Of those unsafe outputs, SQLProof had 98.4% and post-verification benign acceptance had 93.7%. The values are not included in the clean generator execution accuracy. The families of triggers (in the outer test folds) were not used in fine-tuning and calibration, so the score reflects transfer to a new set of trigger families, not memorization of the poisoning marker.

8.8 Proof-obligation analysis.

Table 14: Proof-obligation failures.

Obligation	Violations detected	Median time
Intent entity match	286	6.1
Operation match	194	4.9
Column authorization	231	5.7
Row-scope compliance	163	6.8
Influence-source authorization	278	8.4
Write-safety	144	3.9
Cost budget	107	7.3

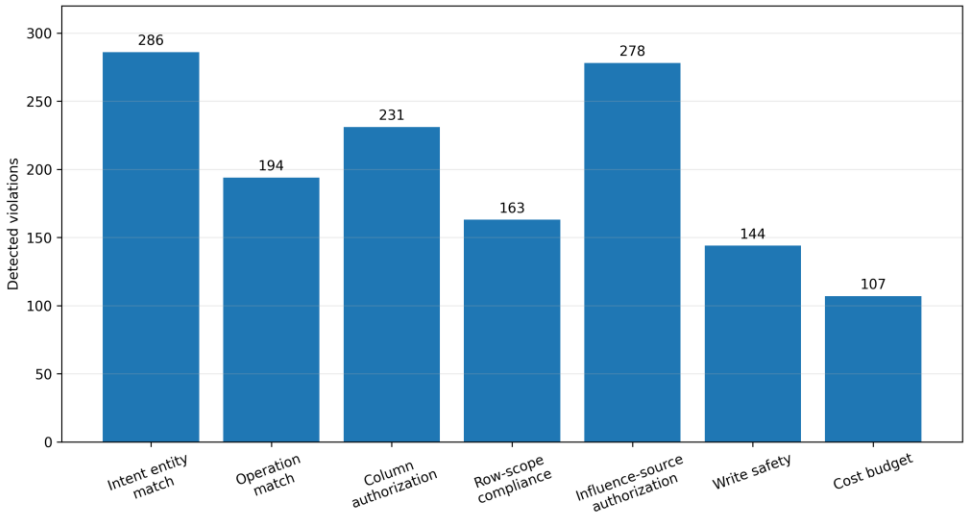

Figure 8: Distribution of proof-obligation failures.

The most common violations are intent entity mismatch and influence-source authorization. This backs up the idea that the modern problem is not limited to SQL syntax: there are many unsafe outputs that are legitimate queries, but the entity or operation is coming from a source that is not authorized to execute the database action.

8.9 Latency and Statistical Reliability

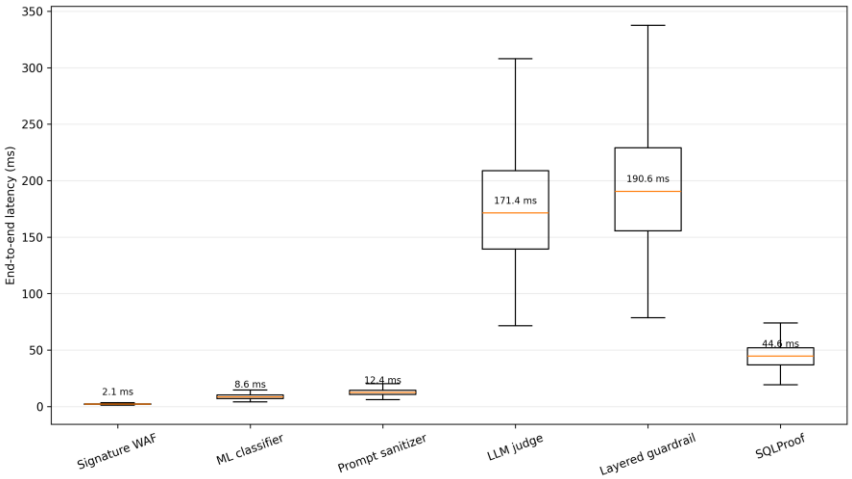

Figure 9: Measured latency distributions.

The measured median SQLProof verifier latency (excluding SQL generation) was 43.8 ms. This value does not include external LLM regeneration called by matched counterfactual provenance escalation. It should thus be understood as verifier-

only latency and not wall-clock latency for the subset of cases where counterfactuals have to be generated. Verification overhead was reduced by the long tail, which was dominated by bounded semantic comparison and cost estimation, by caching contracts and schema graphs, by short-circuiting failed hard constraints, and by invoking database EXPLAIN only when necessary.

8.10 Statistical analysis.

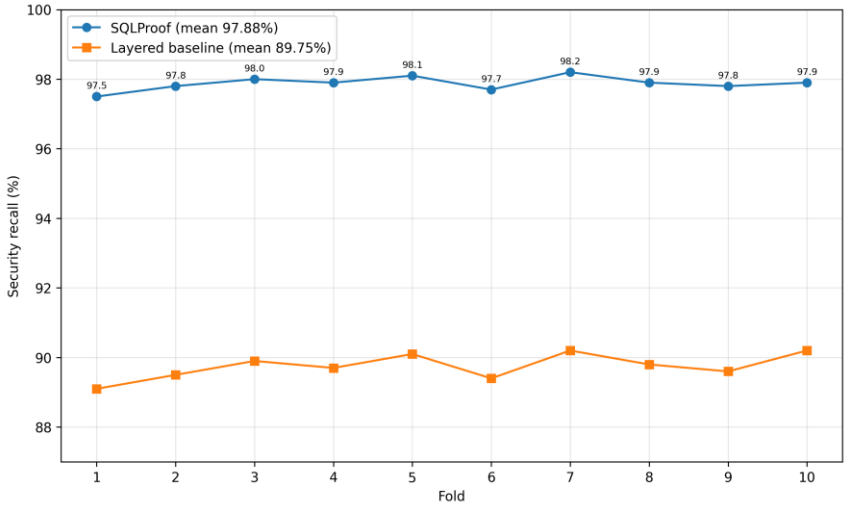

Figure 10: Ten-fold security recall.

The mean security recall for SQLProof over 10 evaluation folds was 97.88% (SD 0.28) while the layered baseline was 89.75% (SD 0.45). The distribution of recall across the folds is shown in Figure 10, and is generally high with little variation across the grouped test folds. The manuscript does not include episode-level test output to reproduce inferential statistics, so these descriptive results are not reported with additional significance claims.

9. Discussion

The results show that the key benefit of SQLProof is not the addition of another layer of detection to the text-to-SQL pipeline, but the modification of the security decision criterion itself. While conventional defenses tend to focus on the similarity of a query to previously seen malicious activity, SQLProof assesses the security-relevant semantics of a generated query, and determines if it is consistent with the declared intent, authorization policy, and sources of influence. This separation accounts for the significant drop in unauthorized execution from 4.7% for the best-layered baseline to 0.6% for SQLProof with benign-query acceptance of 94.1%. This improvement is not just better attack classification but also the capacity of the verifier to disavow syntactically valid SQL that does not have a positive justification from the authorized task in terms of scope, disclosure, write effects, or computational behavior. The ablation results further support the identification of the source of this improvement. Even though a policy is formally approved at the table or column level, there remains a significant semantic gap if the policy is not needed for the analytical goal at hand. The introduction of the intent contract decreased the unauthorized-execution rate from 7.4% to 3.6%, indicating that authorization and task necessity are different security conditions. Influence provenance is the next big step in improving it, as it allows for the identification of query operations that were consequential and came from untrusted contextual channels, not the authorized request or system policy. This is especially relevant in indirect prompt injection and compromised-generation situations, as the SQL that is generated might still be syntactically correct, and might even comply with traditional access-control requirements. In these scenarios, the security violation may not be apparent in the query syntax, but rather in the causal source and semantic requirement of the query-generated operation.This is supported by the comparative behavior of the baselines. The RBAC-only enforcement left the benign utility high, but the protection against semantic authorization violations was limited, since there was no indication of whether or not a specific access was needed for the current request. Likewise, signature and learning based SQL-injection defenses were significantly less effective against indirect, backdoor and semantic attacks as the attacks do not involve the typical payloads associated with SQL-injection. The layered guardrail was able to provide significant containment, but it did not completely eliminate the unauthorized-execution rate, indicating that multiple detection mechanisms are not equivalent to a verification of the final database action against explicit execution obligations. SQLProof thus adds another layer of enforcement between the model and execution by providing a boundary prior to execution.

The security–utility results also indicate that this improvement was not through indiscriminate rejection. SQLProof had 2,010 of 2,023 unsafe episodes, and accepted 1,861 of 1,977 benign episodes, resulting in an

unauthorized-execution rate of 0.64% and benign-query acceptance of 94.13%. The operating point is significant because if the verifier was too strict, it would be easy for it to be able to contain a lot of queries that were actually valid. Rather, the observed trade-off indicates that explicit proof obligations can better partition security-critical semantic expansion from legitimate query complexity than just restrictive execution policies. The other errors are the present boundaries of the framework, not that it is in itself flawed. Unsafe executions were mostly related to resource-exhaustion queries based on incomplete or outdated database statistics, whereas false positives decisions were focused on ambiguous natural language queries, deeply nested SQL and conservative provenance attribution. In these cases, the two forms of uncertainty are revealed. The first is in use: cost boundedness is partly based on runtime database state and optimizer statistics which are not always statically known. The second is semantic - intent contracts always express the meaning of natural language requests, and when several schema-grounded interpretations are possible, they are only approximate. Thus, SQLProof should be viewed as a hybrid assurance mechanism that has a hard component, using deterministic obligations where explicit rules exist, and a soft component, where intent alignment remains a bounded semantic-assurance problem instead of a natural-language to SQL equivalence. This separation aligns with the architecture of the framework, which clearly separates the deterministic policy, provenance, isolation, write-safety, and resource checks from the coverage and surplus based semantic alignment.

Lastly, the results of the cross-generators indicate that the observed containment is mostly due to the independent verifier and not to a particular text-to-SQL model. The evaluated Qwen and GPT based generators achieved containment levels above 97% in all conditions, including the backdoored generator condition. In addition to the group-aware out-of-fold protocol, in which each episode was tested once on a configuration that had been fine-tuned on disjoint development groups, this constitutes evidence that SQLProof is not storing a signature of attacks for each generator. The results thus justify pre-execution proof obligations as a practical security boundary for LLM-mediated database access, meaning that the generator can suggest SQL, but access to it is only granted if the semantics that follow from it can be independently justified.

10. Threats to Validity and Limitations

Attacks, policies, and schemas were implemented and executed in order to evaluate the system. Production databases might have organization-specific rules governing access, statistics that change quickly, dynamic SQL, ORM-generated queries, and proprietary procedures, which might not be completely supported by ProofSQL-Bench. The results reported, therefore, are to be interpreted as evidence under the benchmark conditions evaluated and should not be viewed as guarantees for all deployments of the database.The uncertainty of natural-language intent is also a natural source of uncertainty. User requests can be incomplete, ambiguous, conversational, multi-lingual and domain-specific. SQLProof therefore follows a fail-closed approach: if a statement is ambiguous, it is not automatically assumed, but rather is passed to REVIEW. Due to the fact that it is not generally possible to find an unrestricted semantic equivalence between an arbitrary natural-language request and an SQL program, the approach to intent alignment is based on bounded coverage, surplus and ambiguity criteria instead of the formal equivalence. Influence provenance is an operational approximation and not a direct access to the hidden causal state of the generator. Closed APIs can only offer channel-level evidence of span and node level evidence through source separation, semantic grounding, and matched counterfactual ablation, while open and instrumented pipelines offer richer evidence. If it is not possible to determine the origin of a consequential node then this node is conservatively labelled UNKNOWN or MODEL_INSERTED and dealt with either REVIEW or BLOCK. This design provides for safety but can lead to more false escalations with opaque third-party models. Other restrictions come from SQL normalisation and runtime estimation. Semantic-graph construction can be complicated by dialect-specific constructs, stored procedures, dynamically constructed SQL, user-defined functions, and ORM-generated statements. Query-cost estimates also rely on the optimizer statistics that are available, and can be misleading if there is significant data drift or rapidly changing workloads.The trusted computing base is another area of concern. The verifier should be small, independently audited, and widely tested, and its own security guarantees should not be compromised if there is a vulnerability in the verifier. The intent compiler is also included in the trusted computing base. If the intent contract is incorrectly broadened, incomplete or underspecified, the verifier might evaluate a candidate query against the wrong task specification. This threat is addressed by independent contract annotation, out-of-fold compiler evaluation, fail-closed ambiguity handling and comparison between SQLProof-Auto and SQLProof-Reference, but the performance may suffer if the intent distribution is not represented in ProofSQL-Bench. Lastly, the robustness evidence is limited by the documented poisoning process, trigger families, evaluated generators and benchmark schemas. Cross-generator, cross-dialect and unseen-trigger experiments are used for controlled out-of-distribution evidence but are not sufficient to guarantee robustness against future adaptive attacks or database specific features.

11. Reproducibility and Artifact Packaging

The SQLProof experimental artifacts are structured as a series of versioned benchmark, configuration, verification, and evaluation packages, for support of independent reproduction and auditability. The episode definitions, reference intent contracts, authorization policies, provenance annotations, group-aware fold manifests, generator configurations, calibration thresholds and per-episode verifier outputs are included in each release. These are semantic-graph identifiers, obligation-level decisions, provenance evidence, final execution decisions and measured latency. All configurations for each fold are saved separately from the outer-test predictions, so that thresholds and/or evaluation rules can not be modified after the training process. Random seeds, decoding parameters, model revisions and calibration settings are stored in machine-readable manifests, and cryptographic checksums are calculated for the benchmark files and verifier configurations to ensure precise version tracking.Scripts are also provided in the artifact package to recompute the principal security and utility metrics, fold-level confidence intervals, ablation results and cross-generator analyses of the predictions that were retained. This organization allows an independent researcher to follow up every reported result from the final metric to the specific episode, verifier configuration, and proof-obligation results.

12. Conclusion

This work presents SQLProof, a proof-carrying security framework to overcome an important limitation in the LLM-based text-to-SQL system: a generated query can be syntactically correct and runnable, but still be in violation of the user's authorized intent, disclosure scope, source trust, or operational constraints. SQLProof introduces a verification layer between generation and execution in the database, using the LLM as a trusted decision-maker or even just malicious-pattern detection. Explicit obligations assessed for each candidate query include intent alignment, policy compliance, source authorization, data isolation, write safety and cost boundedness. Only when the generated action is justified by the request, authorization context, and trusted evidence, it is permitted to be executed. This verification paradigm was empirically evaluated on ProofSQL-Bench, a set of 4,000 executable benign and adversarial episodes, and demonstrated to offer a significant security benefit. SQLProof successfully blocked 97.9% of the attacks, achieved a 0.64% of the attacks got executed (unauthorized) and maintained a benign-query acceptance rate of 94.13%. The ablation analysis additionally showed that intent verification and influence provenance were the main drivers of this improvement, further validating the fact that intent verification and trusted-source attribution captures attack behaviors that are inadequately addressed by signatures, prompt sanitization, conventional SQLi classifiers, and RBAC alone. The framework was also evaluated by cross-generating with different text-to-SQL generators, and it was found that the framework is applicable to different generators and even generators that are compromised, as the security decision is independent of the internal behavior of the generator.The main conclusion of this work is that access to databases via a secure LLM must not only be checked to see if a generated SQL query is deemed non-malicious, but whether the actions it will trigger can be independently validated before they are executed. SQLProof translates this principle into machine-checkable deterministic obligations and explicitly bounded semantic assurance, to enable auditable fail-closed decision making for ALLOW, REWRITE, REVIEW or BLOCK. The framework is not necessarily semantically equivalent to any natural-language query; it does not support arbitrary SQL programs; it offers only bounded, reproducible, and enforceable guarantees in the presence of a stated threat model.

Future research should be expanded to production level schemas, stored procedures, dynamic SQL, multi-turn database agents, changing authorization policies, adaptive adversaries, and deployed environments. Special focus will be placed on enhancing semantic disambiguation, runtime-aware cost estimation and provenance verification for opaque models. In summary, SQLProof offers a pragmatic basis to move the security of LLM-supplied databases from post-execution maliciousness detection to pre-execution evidence-based authorization and containment.

Reference

1. V. Zhong, C. Xiong, and R. Socher, “Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning,” Nov. 2017, [Online]. Available: http://arxiv.org/abs/1709.00103
2. T. Yu et al., “Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task.” [Online]. Available: http://www.databaseanswers.org/
3. T. Scholak, N. Schucher, and D. B. Elementai, “PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models.” [Online]. Available: https://github.
4. J. Li et al., “Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs.” [Online]. Available: https://relational.fit.cvut.cz/
5. D. Gao et al., “Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation,” Nov. 2023, [Online]. Available: http://arxiv.org/abs/2308.15363

6. W. Lan et al., “UNITE: A Unified Benchmark for Text-to-SQL Evaluation,” Jul. 2023, [Online]. Available: http://arxiv.org/abs/2305.16265
7. R. Singh and S. Bedathur, “Can the Rookies Cut the Tough Cookie? Exploring the Use of LLMs for SQL Equivalence Checking,” Jun. 2025, [Online]. Available: http://arxiv.org/abs/2412.05561
8. B. ben Ammar and A. M. Alharbi, “SQL Injection Detection Using Fine-Tuned CodeBERT,” *Engineering, Technology and Applied Science Research*, vol. 15, no. 5, pp. 27852–27857, Oct. 2025, doi: 10.48084/etasr.13340.
9. S. Chen, J. Piet, C. Sitawarin, and D. Wagner, “StruQ: Defending Against Prompt Injection with Structured Queries.”
10. F. N. Motlagh, M. Hajizadeh, M. Majd, P. Najafi, F. Cheng, and C. Meinel, “When Prompts Become Payloads: A Framework for Mitigating SQL Injection Attacks in Large Language Model-Driven Applications.”
11. V. Siu et al., “A Framework for Formalizing LLM Agent Security,” Mar. 2026, [Online]. Available: http://arxiv.org/abs/2603.19469
12. A. Doshi, Y. Hong, C. Xu, E. Kang, A. Kapravelos, and C. Kästner, “Towards Verifiably Safe Tool Use for LLM Agents,” in *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering*, New York, NY, USA: ACM, Apr. 2026, pp. 201–205, doi: 10.1145/3786582.3786839.
13. Z. Wu, F. Zhu, X. Shang, Y. Zhang, and P. Zhou, “Cooperative SQL Generation for Segmented Databases By Using Multi-functional LLM Agents,” Dec. 2024, [Online]. Available: http://arxiv.org/abs/2412.05850
14. G. C. Necula, “Proof-Carrying Code,” *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 106–119, 1997.
15. D. E. Denning, “A Lattice Model of Secure Information Flow.”
16. J. A. Goguen and J. Meseguer, “Security Policies and Security Models,” in *1982 IEEE Symposium on Security and Privacy*, 1982, p. 11, doi: 10.1109/SP.1982.10014.
17. “Artificial intelligence risk management framework,” Jul. 2024, doi: 10.6028/NIST.AI.600-1.
18. T. Jin, Y. Choi, Y. Zhu, and D. Kang, “Text-to-SQL Benchmarks are Broken: An In-Depth Analysis of Annotation Errors.” [Online]. Available: https://github.com/
19. R. Klopfenstein, Y. He, A. Tremante, Y. Wang, N. Narodytska, and H. Wu, “SpotIt: Evaluating Text-to-SQL Evaluation with Formal Verification,” Mar. 2026, [Online]. Available: http://arxiv.org/abs/2510.26840
20. S. Chen, A. Zharmagambetov, S. Mahloujifar, K. Chaudhuri, D. Wagner, and C. Guo, “SecAlign: Defending Against Prompt Injection with Preference Optimization,” in *CCS 2025 - Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, Association for Computing Machinery, Inc., Nov. 2025, pp. 2833–2847, doi: 10.1145/3719027.3744836.
21. E. Debenedetti et al., “Defeating Prompt Injections by Design,” Jun. 2025, [Online]. Available: http://arxiv.org/abs/2503.18813
22. M. Cramer and L. McIntyre, “Verifying LLM-Generated Code in the Context of Software Verification with Ada/SPARK,” Feb. 2025, [Online]. Available: http://arxiv.org/abs/2502.07728
23. S. Abraham, H. Koganti, C. Padmaja, V. Radhakrishnan, B. Yamini, and N. v Keerthana, “BERT-Enhanced SQL Injection Black-Box Detection (SqliBERT),” in *Proceedings of 8th International Conference on Intelligent Sustainable Systems, ICISS 2026*, Institute of Electrical and Electronics Engineers Inc., 2026, pp. 837–842, doi: 10.1109/ICISS67859.2026.11453509.
24. T. C. Doan, P. T. Duc, and T. van Loi, “Hybrid BERT-BiLSTM Model for SQL Injection Detection: Potential Applications in Banking Information Systems,” in *Lecture Notes in Computer Science*, Springer Science and Business Media Deutschland GmbH, 2026, pp. 28–36, doi: 10.1007/978-3-032-08557-3_3.
25. T. A. Smrity and M. D. Z. Muntaqim, “SQLGuardNet: Deep learning adaptive processing-based pattern recognition for enhanced SQL injection detection,” *Signal Image Video Process*, vol. 19, no. 13, 2025, doi: 10.1007/s11760-025-04642-2.
26. B. B. Ammar and A. M. Alharbi, “SQL Injection Detection Using Fine-Tuned CodeBERT,” *Engineering, Technology and Applied Science Research*, vol. 15, no. 5, pp. 27852–27857, 2025, doi: 10.48084/etasr.13340.
27. K. Mangaiyarkarasi and P. S. Uma Priyadarsini, “Deep Learning Based Hybrid Architecture Combining BERT and TCNN for SQL Injection Threat Mitigation,” in *Proceedings of the 2025 12th International Conference on Computing for Sustainable Global Development, INDIACom 2025*, Institute of Electrical and Electronics Engineers Inc., 2025, doi: 10.23919/INDIACom66777.2025.11115739.
28. H. Ma, W. Zhang, D. Zhang, and B. Chen, “SQL injection detection algorithm based on a fusion network of syntax and semantic features,” in *Proceedings of SPIE - The International Society for Optical Engineering*, C. Qin and Q. Cheng, Eds., SPIE, 2025, doi: 10.1117/12.3056913.
29. S. R. Nana, D. Bassolé, D. Guel, and O. Sié, “Large Language Models Adaptation for Web Applications Attacks Detection,” in *2025 Annual IEEE Conference on Information Communication Technology and Society, ICTAS 2025 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., 2025, doi: 10.1109/ICTAS64866.2025.11155839.
30. Y. Liu and Y. Dai, “Deep Learning in Cybersecurity: A Hybrid BERT–LSTM Network for SQL Injection Attack Detection,” *IET Inf. Secur.*, vol. 2024, 2024, doi: 10.1049/2024/5565950.
31. Y. Chen, H. Li, B. Wang, A. Yang, P. Fan, and G. Wan, “Request-Response Network Traffic Packets: Enhancing SQL Injection Attack Detection with a Transformer-Based Model,” in *2023 9th International Conference on Computer and Communications, ICCC 2023*, Institute of Electrical and Electronics Engineers Inc., 2023, pp. 1272–1278, doi: 10.1109/ICCC59590.2023.10507479.

32. K. Salah Fathi, S. Barakat, and A. Rezk, “An effective SQL injection detection model using LSTM for imbalanced datasets,” *Comput. Secur.*, vol. 153, p. 104391, 2025, doi: 10.1016/j.cose.2025.104391.