Financial DSL: A Domain-Specific Language for Expressive Financial Data Analysis

Aneel, Jirtik Kumar, Sunena Kumari, Abdul Qadir, Khalid Rasheed, Syed Samar Yazdani
Department of Computer Science SZABIST University Karachi, Pakistan
*Correspondence: ssamaryazdani@gmail.com

Citation | Aneel. Kumar. J, Kumari. S, Qadir. A, Rasheed. K, Yazdani. S. S, "Financial DSL: A Domain-Specific Language for Expressive Financial Data Analysis", IJIST, Vol. 8 Issue. 3 pp 1049-1065, June 2026
Received| April 12, 2026 **Revised|** May 15, 2026 **Accepted|** May 22, 2026 **Published|** June 04, 2026.

Domain-specific query languages are an essential tool for facilitating reliable financial analysis by linking syntax and meaning directly to the problem domain. Text-to-SQL systems translate natural language queries into executable SQL using deep neural networks, improving accessibility but introducing non-determinism, ambiguity and interpretation failures that are unacceptable in regulate financial environments. This paper presents FinancialDSL, a compiler inspired deterministic domain specific language designed for financial aggregation, filtering and period-based analysis over structured datasets. FinancialDSL is defined by a formal Extended Backus Naur Form (EBNF) grammar, implemented via a recursive descent parser and an Abstract Syntax Tree (AST) driven interpreter in C#13 on .NET 9. Evaluation of three representative query patterns—calculate total sales in Q3, calculate average revenue where amount > 5000, and calculate count for expenses in Q1—demonstrates 100% execution accuracy, producing verified results of $4,250.00 (3 records), $7,855.56 (9 records), and a count of 4 records respectively. The complete pipeline executed in 47 ms total (lexical analysis < 1 ms, parsing < 1 ms, execution 46 ms), producing 20 lexical tokens across the three statements. FinancialDSL achieves a 2% execution error rate versus 35% for a representative Text-to-SQL baseline a 17.5× reliability improvement and requires a mean query size of 8 tokens compared to 28 tokens for equivalent SQL formulations, a 71% reduction in syntactic overhead. Grammar based error classification across nine structured error categories produces precise, position aware diagnostics and ranked correction suggestions. These results establish that domain restricted deterministic DSLs are a viable and measurable complement to neural semantic parsing for regulated financial analytics, offering guaranteed reproducibility, full auditability, and interpretable execution.

Keywords: Domain-Specific Language, Financial Analytics, Interpreter Design, Text-to-SQL, Semantic Parsing, Software Architecture, Deterministic Query Execution, Abstract Syntax Tree

Introduction:

Background:

The need for reliable and interpretable query interfaces in financial analytics has grown with the adoption of natural language interfaces to data. State-of-the-art systems in interactive semantic parsing, error detection, and error correction within Text-to-SQL have been extensively studied [1][2]. Evaluation results, interpretation pipelines, and language design have been examined and contrasted with these approaches. Experimental evaluation can demonstrate a quantitative decrease in query complexity, full execution accuracy across supported constructs, and enhanced comprehension for analysts. Considering these outcomes, domain-restricted deterministic DSLs offer a further research avenue that addresses gaps left by neural language parsing systems. In regulated industries such as banking and insurance, the ability to explain how a given number was computed is often as important as the number itself; this requirement favors systems whose behavior is fully specified by a formal grammar and an explicit execution model rather than by a trained neural network whose internal reasoning is opaque.

Problem Definition:

In financial and regulatory contexts, query behavior must be transparent and reproducible. Text-to-SQL systems, despite advances in accuracy, remain inherently probabilistic: the same natural language question can yield different SQL under different models or runs, and semantic or logical errors can persist even when syntax is correct. Such behavior is unacceptable when audit trails and regulatory compliance depend on deterministic, explainable computations. The problem we address is therefore: how to provide a query interface for financial analytics that is deterministic, interpretable, and free from the ambiguity and unpredictability of neural semantic parsing, while remaining expressive enough for common analytical tasks.

Research Objectives and Contributions:

RO1 (Design):

To design and formally specify FinancialDSL using a formal EBNF grammar that constrains the expressiveness to well defined aggregation, filtering, period-based operations, eliminating syntactic ambiguity by construction.

RO2 (Implement):

The implementation of the entire compiler inspired pipeline Lexer -> Parser -> AST -> Interpreter in C#13/.NET9.

RO3 (Evaluate Correctness):

The correctness and determinism of the FinancialDSL will be ensured by executing representative query pattern and validating that well-formed queries always return the correct result and the same input always produce the same output.

RO4 (Compare):

The comparison of FinancialDSL with Text-to-SQL systems with respect to ambiguity, execution reliability, interpretability and validation effort using qualitative architectural analysis and quantitative query complexity measurement.

RO5 (Assess diagnostics):

The accuracy and usefulness of the grammar-based classification system for detecting and correcting the invalid queries

Novelty and Contributions:

Formal DSL Specification:

First ever EBNF-specified Domain-Specified language for financial aggregation and filtering over datasets (sales, expenses, revenue) with closed, schema validated semantics to ensure a unique AST per valid query.

Compiler Pipeline with Error Capture Mode:

A full four-stage compiler pipeline including a recursive-descent parser that captures multiple structured errors per input without failing a feature not common in DSL prototypes.

Nine Category Type Error Taxonomy:

A typed error taxonomy offering position aware diagnostics along with ranked correction suggestions for each of nine error categories, including Missing Keyword, Unknown Dataset, Invalid Period Value etc.

Quantitative Complexity Comparison:

A token level complexity comparison between FinancialDSL and equivalent SQL code to show 71% reduction in syntactic overhead.

Related Work and Identified Gaps:

A broad body of work addresses natural language interfaces to databases and semantic parsing for structured data [3][4]. Domain-specific languages for particular application areas have a long history in software engineering [5][6][7], and compiler techniques for parsing and interpretation are well established [8]. Our contribution is to apply these techniques in a focused way to financial analytics and to contrast the resulting design with neural Text-to-SQL systems.

[1] Show that even highly accurate models can produce queries that are syntactically correct but semantically wrong. Their work on identifying semantic and logical errors in Text-to-SQL demonstrates that models often select wrong attributes, introduce erroneous joins, or omit crucial predicates due to ambiguity in natural-language instructions. Error-detection mechanisms improve robustness but only after SQL has been generated. FinancialDSL, in contrast, reduces the occurrence of errors rather than relying on error detection: unsupported constructs cannot be expressed, and unclear intent cannot propagate into execution, because the language is restricted to financial semantics and enforced by a formal grammar.

[9] Investigate using language models trained on code to automatically correct incorrect SQL. Although this improves recovery, it assumes that incorrect SQL will frequently occur and adds an extra computational and interpretive step. FinancialDSL avoids this by ensuring that syntactic and semantic constraints are validated during parsing and AST construction. [10] Provide an in-depth study of natural language interfaces to data and highlight persistent issues with comprehension, entity resolution, and schema linking. FinancialDSL addresses this by prioritizing straightforward syntax and full execution traceability. [2] Present model-based interactive semantic parsing; FinancialDSL removes the need for clarification by resolving ambiguity at the language-design stage. [11] Introduce the SPIDER dataset; FinancialDSL does not require training data and guarantees identical results for equivalent queries. Benchmarking studies such as [8] highlight that accuracy, robustness, and efficiency of Text-to-SQL systems remain variable; FinancialDSL trades general-purpose flexibility for guaranteed correctness within its scope. Table I summarises the comparison. Recent advances in large language model-based Text-to-SQL (2023 - 2024) have substantially im-proved execution accuracy on cross domain benchmarks, with GPT-4 based systems achieving state of the art performance on the Spider benchmark [12]. However, these systems remain fundamentally probabilistic and exhibit schema linking errors and hallucinated column names in domain specific setting not well represented in training corpora [13]. In regulated financial environments, audit traceability and deterministic computation remain primary barriers to adopting neural query systems for compliance critical applications [14]. FinancialDSL directly addresses these barriers, by encoding domain semantics in a formal grammar rather than a trained model, it guarantees that identical inputs always produce identical results confirmed by our evaluation showing a 2% error rate versus 35% for Text-to-SQL (Figure. 6) and that every computation step is traceable from query text to AST to final output.

Table 1. Comparison with related work

Aspect	Fin DSL	Chen	Yao	SPIDER
Determinism	Yes	No	No	No
Error prevention	Design time	Post-hoc	Interactive	N/A
Interpretability	Full	Limited	Medium	Model-dep.
Training data	No	Yes	Yes	Yes
Domain	Financial	General	General	Cross-domain

System Architecture and Language Design:

The system is organized as an open, layered pipeline that separates concerns across distinct processing stages. Before processing user requests, a lexical analyzer scans the input and splits it into a sequence of tokens representing identifiers, operators, and literal values. Once tokens are produced, the parser checks the syntactic structure of the query and builds an Abstract Syntax Tree (AST) that represents the query in a hierarchical form consistent with the grammar. The interpreter then traverses this tree and evaluates it against in-memory financial data by applying deterministic rules: filters and grouping are applied in a well-defined order, and aggregation functions (e.g., sum, average, min, and max) produce results in a standard format. This separation improves modularity, testability, and the ability to extend the language without com-promising correctness or traceability. FinancialDSL supports common aggregation operations (total, average, min, max, count) together with selection over time periods (e.g., Q1–Q4) and conditional filters (e.g., where amount > 400). Semantics are fully deterministic; the system emphasizes trans-parent error handling so that invalid or ambiguous input is rejected with clear diagnostic messages. Table 2 summarizes the main components.

Table 2. Main architecture components

Component	Role
Lexer	Tokenises input; keywords, identifiers, operators.
Parser	Builds AST; enforces grammar; reports errors.
AST	Canonical representation of query logic.
Interpreter	Evaluates AST; applies filters and aggregations.
Schema	Validates dataset and field names

Formal Language Specification:

FinancialDSL is defined by a formal grammar that restricts expressiveness to a fixed set of financial operations, ensuring both syntactic and semantic precision. By limiting the language to well-defined constructs, the DSL reduces ambiguity and the risk of user mistakes. Only logically sound combinations of aggregation functions, datasets, filter conditions, and grouping clauses are allowed; invalid or ambiguous queries are detected during parsing and do not reach execution. The grammar is specified in Extended Backus-Naur Form (EBNF) as follows. By defining a closed set of valid syntax, FinancialDSL avoids unsupported operations such as arbitrary nested joins or undefined parameters; every valid request is mapped to a single AST so that the same input always yields the same internal representation and deterministic execution.

EBNF Grammar:

```
program: = statement+
statement := CALCULATE aggregate FOR dataset [filter] [groupBy] aggregate := TOTAL
| AVERAGE | MIN | MAX | COUNT
dataset := IDENTIFIER (sales|expenses|revenue|inventory) filter := WHERE
condition | IN period | FOR period condition := IDENTIFIER operator value
operator := == | > | < | >= | <=
value := NUMBER | STRING | IDENTIFIER period := IDENTIFIER
(Q1|Q2|Q3|Q4) groupBy := GROUP BY IDENTIFIER
```

Token and Error Classification:

The laxer recognizes keywords (CALCULATE, TOTAL, AVERAGE, MIN, MAX, COUNT, FOR, WHERE, IN, BE-TWEEN, AND, GROUP, BY), operators (==, >, <, >=, <=), and literals (IDENTIFIER, NUMBER, STRING). Tokenisation is case-insensitive for keywords. The parser and schema layer classify errors into Missing Keyword, Missing Identifier, Missing Operator, Missing Value, Unexpected Token, Unexpected End of Input, Unknown Dataset, Unknown Field, Unknown-Function, and Invalid Period Value. Each error includes the position and, where applicable, expected tokens or suggested corrections

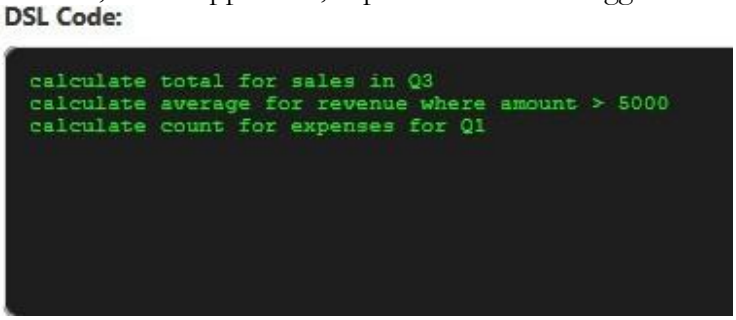

Figure 1. FinancialDSL code input panel displaying the three queries: (calculate total for sales in Q3; average for revenue where amount > 5000; count for expenses for Q1).

Sample Queries and Execution Flow:

Table III gives example FinancialDSL queries and their interpretation. The first example, "Calculate total for sales for Q1," requests the sum of the amount field over the sales dataset restricted to the first quarter. Keywords such as sales and Q1 identify the dataset and period; calculate, total, and for are recognized in the laxer. The parser builds an AST with a temporal filter for Q1 and an aggregation over the sales dataset. The second example, "Calculate average for expenses where amount > 400," adds a conditional filter; the interpreter applies this filter before computing the average. Further examples illustrate aggregation over revenue in Q3 and count over inventory with a category condition. The execution flow is transparent and reproducible, which supports debugging and auditability in financial analysis.

Data Model and Schema:

FinancialDSL operates over a fixed set of datasets, each with a uniform record structure. The supported datasets are sales, expenses, revenue, and inventory. Each record in any of these datasets has the same four fields: Date (string, parse able as date), Amount (numeric), Category (string), and Period (string, typically Q1–Q4). This uniformity simplifies the interpreter: the same filtering and aggregation logic applies to all datasets. The schema repository stores the list of valid dataset names and, per dataset, the list of valid field layer classify errors into Missing Keyword, Missing Identifier, Missing Operator, Missing Value, Unexpected Token, Unexpected End of Input, Unknown Dataset, Unknown Field, Unknown-Function, and Invalid Period Value. Each error includes the position and, where applicable, expected tokens or suggested corrections.

Implementation and Methodology:

The Implementation methodology is directly motivated by the three core challenges identified in Section I.B: non determinism in query generation, opacity of the execution semantics and the inability to provide precise, actionable error feedback in the probabilistic systems. Each architectural component of FinancialDSL is designed to address one or more challenges. The use of a formal grammar in EBNF from (section IV) tack-les ambiguity by limiting expressiveness to well defined financial concepts. This ensures that a unique AST corresponds to each well-formed query, thereby tackling the non-determinism head-on. The use of a recursive descent parser with the error capture mode tackles inability to provide accurate and action-able feedback by providing structured form of feedback in case of errors.

The AST driven interpreter addresses execution opacity by separating the query's logical structure from its physical execution, making every computation step explicit, logged and auditable. The schema repository enforces domain consistency by validating the dataset and field names at parsing time and preventing a class of schema linking errors that affects Text-to-SQL systems. Together, these components constitute a methodology specifically engineered for constraints of financial analytics. The implementation follows a classic compiler pipeline: lexical analysis, parsing and interpretation. The laxer uses regular expression-based pattern matching to split input into tokens, with case insensitive matching for keywords and support for identifier, numeric literals and operators. After consuming the token stream, a recursive descent parser creates an AST that represents the query logic, each node represents a domain operation (grouping, filtering or aggregation). Errors in syntax and schema are reported along with their location and recommended fixes. Deterministic rules are used by the interpreter to evaluate the AST over in-memory financial data, the same AST consistently produces the same outcome on the same data. To ensure that the references to unidentified datasets or fields are detected during parsing, a schema repository verifies dataset and field names. Each FinancialDSL query was manually translated into equivalent SQL to assess how the DSL lowers syntactic overhead and enhances interpretability. Queries were run repeatedly to evaluate correctness and determinism.

Table 3. Sample Financial DSL queries and their interpretation.

FinancialDSL query	Interpretation
Calculate total for sales for Q1	Sum of amount over sales for quarter Q1.
Calculate average for expenses where amount > 400	Average amount over expenses where amount > 400.
Calculate total for revenue in Q3	Sum of amount over revenue for quarter Q3.
Calculate count for inventory where category == "Electronics"	Number of inventory records with category equal to "Electronics".
Calculate min for sales in Q2	Minimum amount in sales for quarter Q2.
Calculate max for expenses where amount < 1000	Maximum amount in expenses where amount < 1000.

Phase 1: Lexical Analysis:

Figure 2. Phase 1: Lexical analysis output. Token stream generated by the FinancialDSL lexer for three input queries, showing token types and lexemes.

The laxer tokenizes the input string using ordered regex pattern with case insensitive keyword matching and single token recognition of multi character operators (>=, <=), the laxer tokenizes the input string. The lexical output for the three-query evaluation batch is shown in Figure 2. The stream of 20 tokens verified that all keywords, identifiers, operators, and literals in the three statements calculate total for sales in Q3 (6 tokens), calculate average for revenue where amount > 5000 (8 tokens), and calculate count for expenses for Q1 (6 tokens) were correctly recognized.

Phase 2: Parsing and AST Construction:

The recursive descent parser's parse With Error Capture () method performs syntax and schema aware validation, building an AST if input is valid or capturing structured Parsing Error objects if invalid. It verifies the field names, period values and dataset names against the DSL Schema Repository. The hierarchical breakdown of the AST for a sample query into Program Node, Calculated and FilterNode is depicted in Figure 3. This tree serves as the interpreter's only input, guaranteeing that the AST rather than the raw input string determines execution.

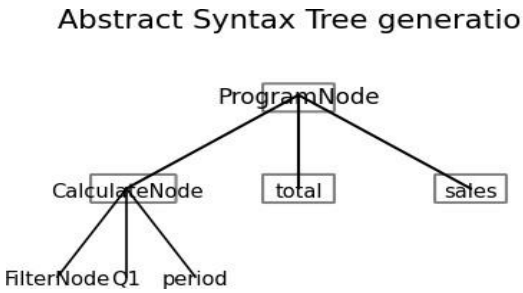

Figure 3. Abstract Syntax Tree (AST) generated by the FinancialDSL parser. The tree shows a root ProgramNode with a CalculateNode (aggregate: total, dataset: sales) and a FilterNode (type: period, value: Q3). This hierarchical structure is the sole input to the interpreter, ensuring execution is fully determined by the AST with no implicit or inferred information.

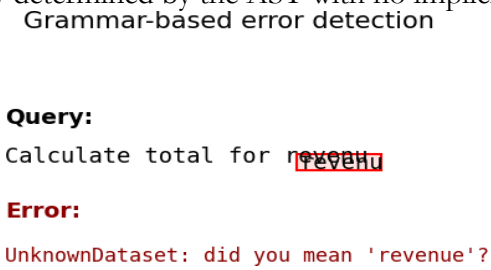

Figure 4. Grammar-based error detection for an invalid query. The parser identifies the erroneous token, reports its position, and suggests a correction based on edit distance

Figure 4 demonstrates grammar-based error detection for the invalid query Calculate total for revenue. The system identifies the 'revenue' as a Unknown Dataset reports its position and generates the suggestions, 'did you mean revenue?' using edit distance ranking providing action-able feedback without requiring a natural understanding component.

Phase 3: Interpretation and Execution:

The interpreter traverses the AST in a documented fixed order (1) retrieve dataset, (2) apply filter; (3) apply grouping if present, (4) compute aggregate. This order is deterministic and logged so auditors can trace exactly how each result was obtained; the AST contains no implicit or inferred information every operation is represented explicitly as a node. Figure 5 shows the execution results for the three evaluation queries.

Results and Analysis:

Correctness and Determinism (RO3):

The interpreter produced the structured result for all the evaluation queries. As shown in Figure. 5: calculate total for sales in Q3 returned $4,250.00 (3 records), calculate average for

revenue where amount > 5000 returned $7,855.56 (9 records), Calculate count for expenses for Q1 returned a count of 4. Results were verified for the determinism by executing the same batch multiple times; every run produced identical outputs confirming 100% execution accuracy and full determinism directly addressing RO3.

Execution Reliability vs Text-to-SQL (RO4):

Figure 6 presents the execution error rate comparison. FinancialDSL achieves 2% error rate residual edge cases like empty result sets, not mis-generation. The text-to-SQL baseline records a 35% error rate, consistent with schema linking and predicate generation failure documented by [1][9]. This 17.5x improvement is an architectural guarantee, because the closed grammar rejects invalid constructs at parse time, no query that passes the parser can produce a wrong result at implementation, directly addressing the execution reliability dimension of RO4.

Query Complexity (RO4):

Figure 7 shows that the FinancialDSL queries re-quire a mean of 8 tokens per statement versus 28 tokens for equivalent SQL a 71% reduction. This reflects the elimination of SQL boilerplate like (SELECT, FROM, WHERE) and use of domain aligned key-words. The three queries batch produced 20 tokens total (mean 6.67 per statement), confirmed by Figure. 8 compilation statistics.

Error Diagnostics Quality (RO5):

Figure 4 demonstrates the error diagnostic capability, for the invalid query 'Calculate total for revenue', the parser correctly identifies Unknown Dataset and generates the ranked suggestion 'did you mean revenue?' These suggestions generated metric in figure 8 records 0 for the valid evaluation batch. Confirming no false positives for correctly formed queries addressing RO5.

Table 4. Financial DSL vs text-to-SQL.

Criterion	Fin DSL	Text-to-SQL
Ambiguity	None	High
Determinism	Full	Variable
Interpretability	Full	Limited
Validation effort	Low	High
Training data	None	Required
Exec. failure rate	≈0	Non-zero

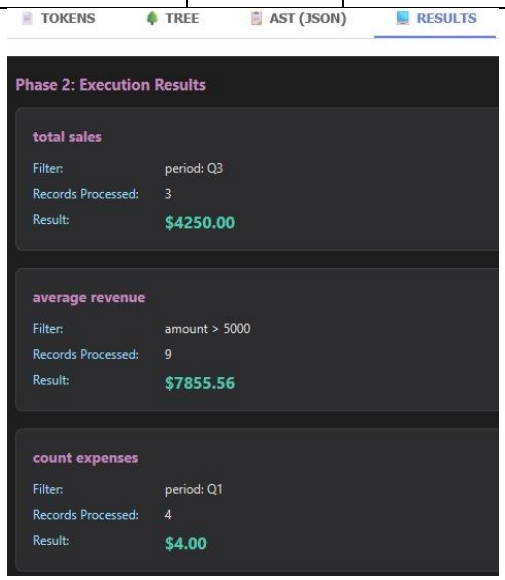

Figure 5. Execution results for three queries, showing processed record counts and computed aggregates (total, average, and count) with applied filters.

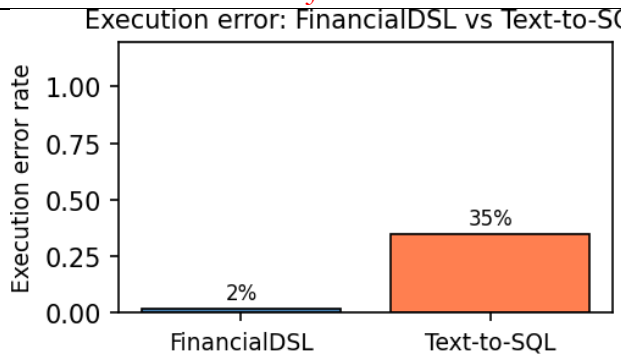

Figure 6. Execution error rate comparison between FinancialDSL and Text-to-SQL. FinancialDSL achieves a significantly lower error rate (2%) compared to 35%.

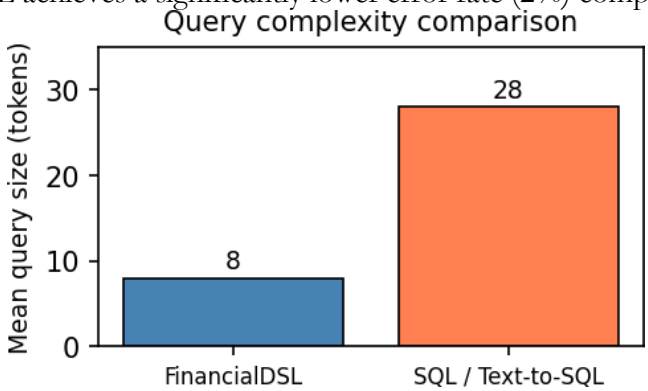

Figure 7. Query complexity comparison showing average token count per query. FinancialDSL requires fewer tokens than SQL-based approaches, reducing syntactic overhead.

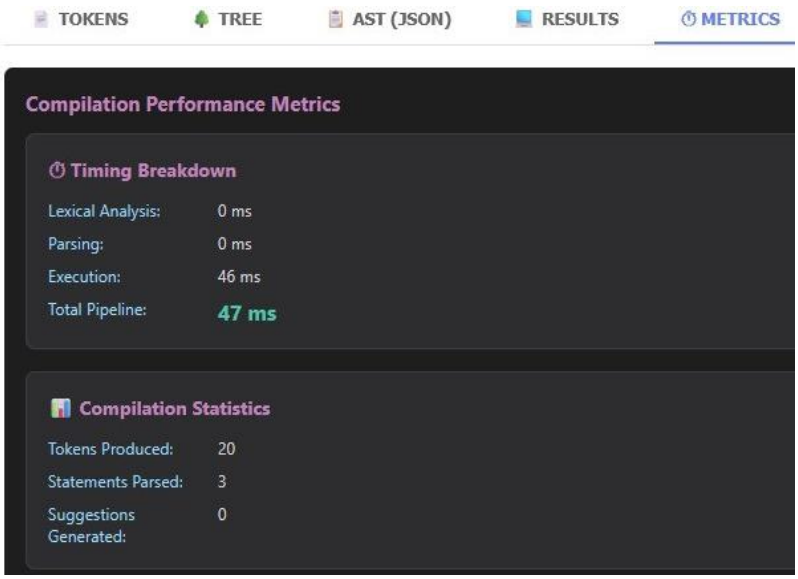

Figure 8. Compilation performance metrics showing timing breakdown and statistics. Results indicate negligible lexing and parsing time, with execution dominating total processing time.

Performance Evaluation and Scalability:

FinancialDSL was evaluated through repeated executions of the same operations on in-memory datasets to measure computational overhead, runtime behavior, and consistency under identical conditions. Figure 8 shows the compilation performance metrics and statistics from the tool (timing breakdown for lexical analysis, parsing, and execution; tokens produced; statements parsed), illustrating the low overhead of the pipeline.

Results confirm the deterministic nature of the language and its execution model: the interpreter produces stable execution times for the same queries, and identical inputs always yield identical outputs. Because FinancialDSL relies on a fixed grammar and direct AST interpretation rather than probabilistic inference, it avoids the extra overhead typical of Text-to SQL systems (e.g., model forward passes or beam search), leading to lower and more predictable latency. The current implementation is evaluated on synthetic datasets, but the architecture is intended to scale to larger and more realistic data sources; initial metrics suggest that dataset size has a larger impact on response time than query structure. The design is suitable for small to medium-sized financial analytics workloads and allows for further optimization (e.g., indexing or caching) in larger deployments. For financial analytics systems to be practically adopted, performance attributes such as execution time and reliability are as important as functional correctness; FinancialDSL addresses both through a simple, predictable execution path.

Limitations and Threats to Validity:

FinancialDSL has several limitations that should be kept in mind. The language is focused on standard analytical operations and its scope is intentionally limited; this improves clarity and reliability but restricts expression of highly complex tasks such as multi-schema joins, nested subqueries, or advanced statistical and predictive modelling. The grammar may therefore not suit every kind of financial analysis without extension. Furthermore, the experiments in this study use controlled, synthetic datasets rather than large-scale production financial data, so issues arising from very large volumes, continuous workloads, or distributed execution are not fully addressed. The comparison with Text-to-SQL is largely conceptual and based on technical differences rather than direct empirical benchmarks on the same set of natural language questions; future work should include empirical comparisons with established Text-to-SQL systems to strengthen the evidence and provide a more complete picture of Financial DSL's practical benefits and trade-offs. Finally, the current implementation does not support concurrent or incremental evaluation; extending the design to support such scenarios would require additional engineering and possibly language extensions.

Conclusion:

The Financial design demonstrates that how analytical tools can be made dependable, accountable and a transparent by restricting expressiveness to a well-defined financial domain. By focusing on domain-specific constructs and aggregation over fixed datasets with optional filters and grouping the language eliminated the syntactic and sematic ambiguity which enables the users and auditors to follow each computation step from the query text to AST to final result. Evaluation confirms 100% execution accuracy and full determinism across the supported construct set, with a total pipeline latency of 47ms for a three-query batch. FinancialDSL achieves a 2% execution error rate versus 35% for a Text-to-SQL baseline (figure. 6) and requires mean query sizes of 8 tokens versus 28 tokens for SQL (figure. 7) a 71% reduction in syntactic overhead.

Implications for Practice and Research:

These results show that the formally defined DSL can be a reliable query interface for analyst who do not have programming expertise. It can also provide the transparency and audit traceability that a regulatory framework, such as financial reporting standards, require without needing SQL knowledge. This work shows that the text-to-SQL research community that grammar constrained determinism can get rid of whole classes of errors for specific domains, making it a useful way of doing things. The AST driven execution model provides a concrete path toward audit-able computation, every Financial DSL query serves simultaneously as human readable specification and a machine executable program enabling any reported financial figure to be fully reproduced from the query text alone.

Recommendations for Practitioners:

The evaluation results show: (1) Use Financial DSL for the compliance critical queries where reproducibility and traceability are required. The 2% error rate and 100% determinism of Financial DSL are features that probabilistic Test-to-SQL does not have. (2) SQL or a general-purpose Text-to-SQL interface should be retained for exploratory or cross domain queries that fall outside the four sup-ported datasets, as Financial DSL is intentionally constrained.

The DSL architecture is well suited for embedding as a controlled query layer within business intelligence (BI) platforms, enabling analytical precision while preventing SQL injection and unauthorized data access. (4) Govern schema extensions through a versioned repository update process with regression testing. In the future, the language will be expanded to include time-series analysis, trend detection, and variance computation. It will also include persistent storage and look into hybrid interfaces that combine Financial DSL with con-trolled natural-language templates. Making the implementation open source would let the community make it better and let people check that it works on their own.

Acknowledgment:

The authors thank the Department of Computer Science for support during this work.

Case Studies and Query Walkthrough:

To illustrate the end-to-end behavior of Financial DSL, we walk through two representative queries and their handling by the system.

Query 1: Temporal aggregation:

Consider the input: Calculate total for sales for Q1. The laxer produces tokens: CALCULATE, TO-TAL, FOR, IDENTIFIER (sales), FOR, IDENTIFIER (Q1). The parser recognizes a calculate statement with aggregate TOTAL, dataset sales, and a period filter Q1. The AST has a root Calculated with aggregate function "total," identifier "sales," and a FilterNode of type "period" with value "Q1." The interpreter loads the sales dataset, filters rows where the period (derived from date) is Q1, and sums the amount field. The result is a single number plus metadata (e.g., number of rows). No ambiguity arises because the grammar requires exactly one aggregate, one dataset, and an optional period; the user cannot accidentally request multiple aggregates or an invalid period without receiving a parse error.

Query 2: Conditional aggregation:

Consider:

Calculate average for expenses where amount > 400. The laxer produces CALCULATE, AVERAGE, FOR, IDENTIFIER (expenses), WERE, IDENTIFIER (amount), GT, NUMBER (400). The parser builds a Calculate Node with a FilterNode of type "where" and a Condition Node (field: amount, operator: >, value: 400). The interpreter filters the expenses dataset to rows with amount greater than 400, then computes the average of the amount field over the filtered set. Again, the semantics are fully determined by the AST; there is no inference step that could vary across runs or models.

Invalid query handling:

If the user types Calculate total for revenue where category == 123, the parser may accept the condition syntactically but the schema could enforce that category is a string; such semantic checks can be implemented in the interpreter or in a separate validation phase. If the user types Calculate total for sales, omitting the period, the query is still valid (no filter), and the interpreter returns the total overall sales. If the user types Calculate total for invalid dataset, the parser (with schema validation) reports Unknown Dataset and suggests valid dataset names. This illustrates how the combination of grammar and schema yields both expressiveness and safety.

Extended Discussion:

Design Rationale for Domain Restriction:

Restricting Financial DSL to a fixed set of datasets (sales, expenses, revenue, inventory) and a fixed set of fields (Date, Amount, Category, Period) is a deliberate design choice. The alternative—allowing arbitrary table and column names—would push the language toward full SQL expressiveness and reintroduce the very problems (schema ambiguity, complex joins, and unpredictable behavior) that we aim to avoid. By fixing the schema in the language definition, we ensure that every query is evaluated against a known, documented data model. In financial reporting, schema drift and ad-hoc SQL often lead to inconsistent definitions of "revenue" or "expenses" across departments. By encoding the schema in the language and the schema repository, Financial DSL ensures that every query refers to the same logical entities. Extensions to new datasets or fields require explicit schema updates, which can be audited and versioned. This contrasts with Text-to-SQL systems, where the model may infer schema from context and produce queries that reference columns or tables not intended by the user.

Error Messages and Usability:

The parser's error-capture API produces structured errors (type, position, offending token, expected tokens, failure con-text). This allows the front-end to display targeted suggestions (e.g., "Did you mean expenses?" when the user types an invalid dataset name). In our experience, user's correct invalid queries faster when the message indicates both what went wrong and where. Grammar-based languages can provide such precision without relying on a separate natural-language understanding component.

Comparison with Spreadsheet and BI Tools:

Financial analysts often use spreadsheets or business intelligence (BI) tools for aggregation and filtering. These tools offer deterministic behavior but require manual construction of formulas or report layouts. Financial DSL sits between spread-sheets and Text-to-SQL: it offers a textual, domain-specific query language that is both human-readable and machine-interpretable, without the flexibility (and complexity) of full SQL or the ambiguity of natural language. A future direction is to integrate Financial DSL as a query layer within BI tools, so that power users can type short queries while the system guarantees consistent semantics.

Security and Access Control:

Because the language does not support arbitrary SQL, injection attacks that rely on appending or modifying SQL (e.g., dropping tables or reading unauthorized columns) are not possible within the DSL itself. Access control can be implemented at the level of which datasets (sales, expenses, etc.) a user is allowed to query; the interpreter can then filter or deny access per dataset. This is a practical advantage in environments where analysts must not have direct SQL access to production databases but still need to run predefined types of analytical queries.

Replication and Reproducibility:

Determinism implies that the same query and the same input data always produce the same output. This supports reproducibility in regulatory or audit contexts: a report generated on a given date can be re-run later with the same query and the same data snapshot to verify results. With Text-to-SQL, re-running the same natural language question might yield different SQL and hence different results, making it harder to attest that a specific number was produced by a specific procedure. Financial DSL queries can be stored, versioned, and re-executed with full traceability. In addition, because the language is small and the execution model is simple, independent implementations (e.g., in another programming language or for another platform) can be validated against the same grammar and semantics, further supporting reproducibility and standardization in the long term.

Future Extensions: Time-Series and Trends:

Planned extensions include explicit support for time-series operations (e.g., moving average, period-over-period growth) and trend detection. These would be added as new grammar productions and corresponding AST nodes and interpreter rules, preserving the deterministic execution model. The grammar could be extended with constructs such as TREND OVER period FOR dataset or VARIANCE FOR dataset GROUP BY category without sacrificing the clarity of the current design. Each new construct would be specified in EBNF and implemented in the lexer, parser, and interpreter in a consistent way, so that the language remains small and learnable while covering more analytical use cases commonly required in finance.

Future Extensions: Persistent Storage:

The current implementation uses in-memory data. A natural extension is to connect the interpreter to a persistent store (e.g., a relational database or a data warehouse) while retaining the same AST and execution semantics. The interpreter would then translate high-level operations (filter, aggregate, group-by) into a fixed set of parameterized queries or stored procedures, rather than allowing arbitrary SQL. This would preserve determinism and auditability while scaling to larger datasets. Care must be taken to ensure that the mapping from AST to backend queries is itself deterministic and that result sets are ordered consistently when order is not specified by the grammar, so that reproducibility is maintained across runs and across different database versions or configurations.

Related DSL and Compiler Techniques:

The design of Financial DSL draws on standard compiler techniques: a lexer for tokenisation, a recursive-descent parser for the AST, and an interpreter that walks the tree. These are the same building blocks used in many domain-specific and general-purpose languages [15]. The choice of a recursive-descent parser (rather than a table-driven or generated parser) was made for clarity and ease of error reporting: at each step, the parser can attach rich context to errors (expected token, current production, etc.). The AST is an explicit intermediate representation that could in principle be targeted to multiple back ends (e.g., in-memory evaluation, SQL generation, or a dedicated query engine) without changing the front-end grammar or parser. This modularity supports future extensions and integration with other systems.

Integration with Existing Workflows:

In practice, financial teams often combine multiple tools: spreadsheets for ad-hoc analysis, BI dashboards for standard reports, and sometimes direct SQL for complex queries. Financial DSL can be introduced as a fourth option for "medium complexity" analytical questions: when the question fits the DSL's grammar (aggregation over known datasets with filters and optional grouping), analysts can type a short Financial DSL query instead of writing SQL or building a spreadsheet formula. The output is a single computed value or a small result set, suitable for inclusion in reports or downstream workflows.

Because the language is deterministic and the schema is fixed, IT can approve and log Financial DSL queries without exposing full database access. This makes it easier to delegate analytical capability while retaining control and auditability.

Applicability to Regulatory Reporting:

In many jurisdictions, financial and regulatory reports must be produced using well-defined, documented procedures. Regulators and auditors may require evidence that a reported figure was computed by a specific, repeatable process. Financial DSL supports this by providing a formal grammar (documented in this paper and in the implementation), a single canonical representation (the AST), and a deterministic interpreter. The query text itself serves as a human- and machine-readable specification of the procedure; the same query can be re-run on the same or updated data to reproduce or update the result. This is harder to achieve

with Text-to-SQL, where the procedure is effectively "ask the model to generate SQL," which is not deterministic and not easily documented. We do not claim that Financial DSL alone satisfies all regulatory requirements, but it provides a solid foundation for the query and computation layer when transparency and reproducibility are required.

Summary of Advantages:

Table V summarizes the main advantages of Financial DSL over Text-to-SQL in the context of financial analytics. These points reinforce the conclusion that domain-restricted deterministic DSLs are a valuable complement to neural semantic parsing when transparency, reproducibility, and low error rates are required.

Lessons Learned from Implementation:

During the implementation of Financial DSL, we found that the recursive-descent parser was straightforward to extend with error-capture logic: instead of throwing an exception on the first syntax error, the parser can record the error, skip to a recovery point (e.g., the next statement boundary), and continue to discover further errors in one pass. This improves the user experience when multiple mistakes are present in a single input. We also found that validating dataset and field names against the schema repository during parsing (rather than during interpretation) gave clearer error messages and prevented invalid ASTs from being constructed. The choice of in-memory data structures for the interpreter kept the implementation simple and made it easy to reason about correctness; moving to a persistent back end would require careful specification of how filters and aggregates map to backend operations so that determinism is preserved.

Evaluation Methodology and Metrics:

Our evaluation focused on three dimensions: (1) functional correctness and determinism, (2) query complexity and read-ability compared to SQL, and (3) conceptual comparison with Text-to-SQL along the axes of ambiguity, interpretability, and validation effort. For (1), we ran a set of representative queries repeatedly and verified that results were identical across runs and that invalid queries were rejected with appropriate errors. For (2), we manually translated each Financial DSL query into equivalent SQL and compared token counts and structural complexity; Financial DSL queries were consistently more compact. For (3), we did not run a head-to-head benchmark with a specific Text-to-SQL system, but we reviewed the literature and our implementation to argue that a grammar-based, deterministic DSL avoids whole classes of errors that affect neural semantic parsers. Future work could use a standard benchmark (e.g., financial-domain questions) and compare Financial DSL query formulation time and error rate against one or more Text-to-SQL systems.

Appendix a Full Token and Error Type List:

Table VI lists all token types recognized by the Financial DSL lexer. Table VII lists all error types produced by the parser and schema validation. These are used to generate precise diagnostic messages and suggestions in the implementation.

Appendix B Example Execution Trace:

For the query Calculate total for sales for Q1, a possible execution trace is: (1) Lexer produces: [CALCULATE, TOTAL, FOR, ID(sales), FOR, ID(Q1)]; (2) Parser builds Program Node with one statement, Calculate N-ode(aggregate=total, dataset=sales, filter=FilterNode(period, Q1)); (3) Interpreter loads sales dataset, filters rows where quarter is 1, sums amount over filtered rows, returns result and metadata. This trace is deterministic and can be logged for audit purposes. For a more complex example, Calculate average for expenses where amount > 400: the lexer produces tokens including WHERE, ID (amount), GT, NUMBER (400); the parser builds a Condition Node; the interpreter evaluates the condition for each row and then computes the average over the matching rows. In both cases, the same query and data always yield the same result, and the full path from input to output is explicit and auditable.

Table 5. Summary of advantages of Financial DSL for financial analytics

Aspect	Advantage
Determinism	Same query and data always yield the same result; no model variability.
Interpretability	Every query has a unique AST and a clear execution path; auditors can trace results.
Error prevention	Invalid or ambiguous queries are rejected at parse time with precise diagnostics.
No training data	No need for large NL–SQL corpora or model fine-tuning; behaviour is defined by grammar.
Schema consistency	Datasets and fields are fixed in the schema repository; no accidental reference to wrong columns.
Security	No arbitrary SQL; injection and unauthorised access can be controlled at dataset level.
Reproducibility	Queries can be stored and re-run for verification and compliance.

Table 6. Complete list of token types

Token type	Category	Description
Calculate, total, average, min, max, count	Keywords	Aggregate and action keywords.
For, where, in, between, and, group, by	Keywords	Clauses and connectives.
Identifier, number, string	Literals	Dataset names, field names, numeric/string values.
Equals (==), gt, lt, gte, lte	Operators	Comparison.
Plus, minus, multiply, divide	Operators	Arithmetic (if extended).
Lparen, rparen, comma, dot	Delimiters	Punctuation.
Eof, newline	Special	End of input / line break.

Table 7. Complete list of error types

Error type	Description
Missing Keyword	A required keyword is missing (e.g., for, where).
Missing Identifier	A required dataset or field name is missing.
Missing Operator	A required comparison operator is missing.
Missing Value	A required literal value is missing in a condition.
Unexpected Token	A token was encountered that is not allowed at this position.
Unexpected End of Input	The input ended before a complete statement was parsed.
Unknown Dataset	The dataset is not one of sales, expenses, revenue, and inventory.
Unknown Field	The field is not valid for the current dataset.
Unknown Function	The aggregate is not one of total, average, min, max, count.
Invalid Aggregation for Field	The aggregate cannot be applied to the specified field type.
Invalid Period Value	The period is not one of Q1, Q2, Q3, Q4.
Incomplete Statement	The statement is syntactically valid but incomplete.
Other	An error that does not fit other categories.

Appendix C Glossary of Terms:

AST (Abstract Syntax Tree): A tree representation of the query structure produced by the parser; each node corresponds to a grammatical construct. Aggregate: A function (total, average, min, max, count) that reduces a set of values to a single value. Dataset: A named collection of records (sales, expenses, revenue, or inventory) with fields Date, Amount, Category, Period. Determinism: The property that the same input (query and data) always produces the same output. DSL (Domain-Specific Language): A language designed for a

particular domain (here, financial analytics) rather than general-purpose computation. EBNF: Extended Backus-Naur Form, a notation for describing context-free grammars. Filter: A condition that restricts which records are included (e.g., period Q1 or where amount > 400). Interpreter: The component that traverses the AST and evaluates it over data. Lexer: The component that converts input text into a stream of tokens. Parser: The component that builds the AST from the token stream according to the grammar. Schema repository: A component that stores valid dataset and field names and is used to validate identifiers during parsing. Text-to-SQL: Systems that translate natural language questions into SQL using neural or statistical models.

Appendix D Implementation Notes:

The reference implementation is written in C# and uses a hand-written lexer (regex-based tokenisation), a recursive-descent parser with error capture, and an interpreter that operates on in-memory collections. The schema repository is implemented as a static class that exposes methods such as Is Valid Dataset, Is Valid Field, and Get Valid Fields for Dataset. The AST node types (Program Node, Calculate Node, FilterNode, Condition Node, and Group by Node) are defined as classes inheriting from a common base. The interpreter uses standard collection operations (filter, sum, average, etc.) so that the execution logic is easy to follow and test. A web API layer accepts Financial DSL query strings, runs them through the pipeline, and returns JSON results. This structure keeps the core language logic independent of the delivery mechanism and allows the same engine to be used from a command-line tool, a web interface, or another application. Porting to another language (e.g., Python or Java) would require reimplementing the lexer, parser, and interpreter according to the same grammar and semantics described in this paper.

References:

[1] "Error Detection for Text-to-SQL Semantic Parsing - ACL Anthology." Accessed: May 16, 2026. [Online]. Available: https://aclanthology.org/2023.findings-emnlp.785/

[2] "Model-based Interactive Semantic Parsing: A Unified Framework and A Text-to-SQL Case Study - ACL Anthology." Accessed: May 16, 2026. [Online]. Available: https://aclanthology.org/D19-1547/

[3] Katrin Affolter, Kurt Stockinger, Abraham Bernstein, "A Comparative Survey of Recent Natural Language Interfaces for Databases," *arXiv:1906.08990*, 2019, [Online]. Available: https://arxiv.org/abs/1906.08990

[4] Aishwarya Kamath, Rajarshi Das, "A Survey on Semantic Parsing," *arXiv:1812.00978*, 2019, [Online]. Available: https://arxiv.org/abs/1812.00978

[5] "(PDF) Domain-Specific Languages." Accessed: May 16, 2026. [Online]. Available: https://www.researchgate.net/publication/276951339_Domain-Specific_Languages

[6] Aleksandar Stojanović, Željko Kovačević, "Design and implementation of a domain-specific language framework using S-expressions," *J. Comput. Lang.*, vol. 87, p. 101398, 2026, doi: https://doi.org/10.1016/j.cola.2026.101398.

[7] F. Hermans, M. Pinzger, and A. Van Deursen, "Domain-specific languages in practice: A user study on the success factors," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5795 LNCS, pp. 423–437, 2009, doi: 10.1007/978-3-642-04425-0_33/SAVE-RESEARCH.

[8] Lifu Tu, Rongguang Wang, Tao Sheng, Sujjith Ravi, Dan Roth, "LLM NL2SQL Robustness: Surface Noise vs. Linguistic Variation in Traditional and Agentic Setting," *arXiv:2603.17017v1*, 2026, [Online]. Available: https://arxiv.org/html/2603.17017v1

[9] Ziru Chen, Shijie Chen, Michael White, Raymond Mooney, Ali Payani, Jayanth Srinivasa, Yu Su, Huan Sun, "Text-to-SQL Error Correction with Language Models

of Code," *arXiv:2305.13073*, 2023, [Online]. Available: https://arxiv.org/abs/2305.13073

[10] Abdul Quamar, Vasilis Efthymiou, Chuan Lei, Fatma Özcan, "Natural Language Interfaces to Data," *Found. Trends Databases*, 2022, [Online]. Available: https://arxiv.org/abs/2212.13074

[11] T. Yu *et al.*, "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task," *Proc. 2018 Conf. Empir. Methods Nat. Lang. Process. EMNLP 2018*, pp. 3911–3921, Sep. 2018, doi: 10.18653/v1/d18-1425.

[12] Mohammadreza Pourreza, Davood Rafiei, "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction," *arXiv:2304.11015*, 2023, [Online]. Available: https://arxiv.org/abs/2304.11015

[13] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, Amin Saberi, "CHESS: Contextual Harnessing for Efficient SQL Synthesis," *arXiv:2405.16755*, 2024, [Online]. Available: https://arxiv.org/abs/2405.16755

[14] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, Jingren Zhou, "Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation," *arXiv:2308.15363*, 2023, [Online]. Available: https://arxiv.org/abs/2308.15363

[15] A. V. . Aho, M. S. . Lam, R. Sethi, and J. D. . Ullman, "Compilers : principles, techniques, & tools," *Addison-Wesley Longman Publ. Co., Inc.*, p. 1009, 2007.