International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group| Volume 6 Issue 3 PP 146-151, 2025
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V6I3P118

Original Article

Natural Language Interfaces for Self-Service Analytics on Data Lakes: Design Patterns, Governance, and Lessons from a Production Deployment

Jeevan Krishna Paruchuri
Independent Researcher, USA.

Received On: 17/07/2025 ***Revised On: 05/08/2025*** ***Accepted On: 07/09/2025*** ***Published On: 20/09/2025***

Abstract *- The promise of natural-language interfaces to enterprise data is older than the LLMs that finally make it tractable, and the gap between the demo and the production deployment is larger than most organizations appreciate. This paper presents a case study of building a natural-language analytics interface on top of a production banking data lake comprising 1,300+ student engagement and curriculum datasets served by dbt Semantic Layer, where the existing SQL surface had reached 80% dbt Semantic Layer adoption and was processing more than 100,000 queries per month at 8-25ms p99 query overhead and where the remaining 20% of non-technical analysts were structurally excluded from self-service because they did not write SQL. We describe the architecture that emerged: a schema-retrieval layer that selects only the relevant subset of the 1,300-table catalog for each natural-language question, an LLM that produces a candidate SQL query, a multi-stage validation layer that enforces governance constraints (row-level security, column masking, query cost ceilings) before any query is executed, and a post-hoc rewriting step that handles the LLM's failure modes qualified column references, ambiguous joins, hallucinated columns. We report pilot results from a four-week deployment to a population of business analysts: 75% adoption (defined as analysts running at least one NL query per week), 80% query accuracy (the produced SQL returns the correct answer to the asked question), and 1.2-second median latency end-to-end. We are honest about the limitations: hallucinated column names remain the dominant failure mode despite schema retrieval, LLM API costs are non-trivial at the query volumes the SQL surface handles, and the governance plumbing is more complex than the natural-language layer. The contribution is a practitioner-grounded design framework with explicit attention to row-level security and audit, intended for teams considering whether the cost of building this layer is justified by the analyst population it unblocks.*

Keywords *- Natural Language to SQL, Self-Service Analytics, LLM, Data Lake, DBT Semantic Layer, Governance, Row-Level Security, Schema Retrieval.*

1. Introduction

A bank with 1,300+ student engagement and curriculum datasets in its data lake has somewhere between 50 and 500 people who can write a SQL query against those tables, and somewhere between 5 and 50 times as many people who would benefit from being able to ask the question that the SQL would answer. The gap between these populations is the structural problem that self-service analytics has been trying to close for two decades. BI tools, semantic layers, dashboards, and data catalogs have all chipped away at it, but the fundamental dependency on SQL literacy has remained, and the unserved population risk analysts, product managers, customer service supervisors, compliance officers has continued to file ad-hoc data requests with the data engineering team and wait days for answers.

The platform under discussion in this paper had reached a mature state of SQL-based self-service. dbt Semantic Layer served as the SQL gateway, achieving 80% adoption among technical users within four months of deployment and currently processing more than 100,000 queries per month at 8 to 25 milliseconds of p99 overhead beyond the underlying Spark execution. Row-level security, column masking, and query cost controls were enforced through dbt Semantic Layer plugins that the platform team had developed and operated for two years. By any reasonable measure, the SQL self-service surface was working well for the people who could use it.

The remaining 20% of analysts roughly the population that did not write SQL were not served. They filed Jira tickets to the data team, waited for engineers to write queries, copied results into spreadsheets, and often waited longer than the business value of the answer justified. The natural-language interface effort began as an attempt to bring this population onto the platform without forcing them to learn SQL first.

This paper documents the design, deployment, and pilot evaluation of that interface. We pursue three research questions. RQ1. How can a natural-language interface be built on top of a 1,300-table production data lake while preserving the governance guarantees (row-level security, column masking, audit) that the existing SQL surface

enforces? RQ2. What are the dominant failure modes of LLM-based SQL generation in a real enterprise schema, and which architectural patterns reduce them? RQ3. Does a natural-language interface actually unblock the non-technical analyst population, measured by adoption and query accuracy, and at what operational cost?

The contributions are: (1) a concrete architecture for an NL-to-SQL system that operates within an enterprise governance framework, (2) a description of the validation layer that prevents the LLM from producing queries that violate governance constraints, (3) pilot evaluation results from a four-week deployment, and (4) honest discussion of the failure modes and operational costs that the literature on text-to-SQL systems often understates. The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 presents the architecture. Section 4 describes the validation layer in detail. Section 5 reports pilot results. Section 6 discusses limitations. Section 7 concludes.

2. Background and Related Work

Natural-language interfaces to databases have a long history. Early work in the 1970s and 1980s built rule-based systems that mapped controlled English subsets to SQL. The rule-based approach was brittle and did not generalize. The 2010s saw a wave of statistical and neural approaches culminating in sequence-to-sequence models trained on benchmarks such as Spider and WikiSQL. These benchmarks measured query accuracy on small academic schemas and produced steady leaderboard progress, but the gap between benchmark performance and real enterprise deployment remained large because real enterprise schemas have hundreds or thousands of tables, ambiguous column names, undocumented joins, and governance constraints that benchmarks do not model.

The arrival of capable general-purpose LLMs GPT-3.5, GPT-4, Code Llama, and their successors has changed the landscape. An LLM with access to schema context can produce SQL of surprisingly high quality on enterprise schemas without any task-specific training, and the failure modes shift from "cannot understand the question" to "produces plausible SQL that is wrong in a specific way." This shift makes the engineering problem different: less about model architecture, more about retrieval, validation, and guardrails.

The deployment patterns for LLM-based NL-to-SQL fall into three families. Template-based systems constrain the LLM to a fixed set of query templates, which sacrifices flexibility for safety. Pure ML systems train or fine-tune a model on the target schema, which requires curated training data and produces a system that is hard to update. LLM-with-retrieval systems use a general-purpose LLM augmented with schema retrieval and validation, which is the architecture we adopt and describe in this paper. Each family has trade-offs:

Approach Flexibility Safety Adaptability Cost ---------- ----- ---------- ---------- ---------- ---- ---------- Template Low High Manual Low Trained ML Medium Medium Retrain required Medium LLM + retrieval High Requires validation Schema changes auto-handled Per-query LLM cost.

Beyond the three deployment patterns, significant research has explored more sophisticated approaches to NL-to-SQL. In-context learning (few-shot prompting) has proven effective: providing the LLM with a handful of example natural language queries and their corresponding SQL generates higher-quality results than zero-shot prompting. Rajkumar et al.’s RESDSQL and DIN-SQL approaches demonstrated that ranking candidate schemas and using intermediate decomposition steps (where the LLM first identifies tables, then joins, then aggregations) can improve accuracy to 85%+ on benchmark datasets. However, these approaches trade increased latency and token consumption for improved accuracy; intermediate decomposition steps require multiple LLM calls. The system described in this paper uses single-stage generation (one LLM call) to maintain sub-1-second latencies, accepting the accuracy trade-off.

Fine-tuned models specifically trained on NL-to-SQL tasks (such as CodeLlama fine-tuned on Spider, or models like SQLCoder) have shown promise for proprietary data warehouses. These models can be deployed on-premises or in controlled cloud environments, avoiding the latency and cost overhead of API calls to external LLM providers. However, fine-tuning requires either (a) manual curation of hundreds or thousands of example (natural language, SQL) pairs for the target schema, or (b) synthetic generation of training examples, which risks training on unrealistic queries. We did not pursue fine-tuning in this pilot because generating high-quality training examples would require months of investment; instead, we evaluated whether a general-purpose LLM with good schema retrieval could achieve sufficient accuracy.

The embedding-based schema retrieval approach used in this system is similar to retrieval-augmented generation (RAG) techniques used in other LLM applications. However, our implementation is tailored to the data warehouse domain: embeddings are computed on column descriptions and column names together (not just on column descriptions), and we apply domain-specific ranking heuristics such as “tables recently joined together in production queries are likely to be joined again.” This domain-specific tuning improved recall compared to a vanilla RAG approach.

3. System Architecture

The architecture has five stages: natural-language input, schema retrieval, LLM SQL generation, validation, and execution. Each stage is described below.

3.1. Schema Retrieval

The 1,300-table catalog is too large to fit in any single LLM context window, and even if it fit, the relevance signal would be drowned out. Schema retrieval narrows the candidate table set. For each natural-language question, the

system embeds the question, retrieves the top-K most semantically similar table descriptions from a vector index built over the catalog, and constructs a focused schema context containing those tables, their columns, types, descriptions, and known foreign-key relationships. K is typically between 5 and 15 depending on the question.

The vector index is built once per catalog snapshot and refreshed when tables are added or descriptions are updated. The indexing job runs nightly. The embeddings come from an off-the-shelf sentence-embedding model; we did not find that fine-tuning the embedding model on the catalog produced enough improvement to justify the maintenance.

3.2. LLM SQL Generation

The LLM is prompted with: a system message describing its role and constraints, the focused schema context from retrieval, optional few-shot examples drawn from past successful queries against similar tables, and the user's natural-language question. It produces a candidate SQL query plus a brief explanation of what the query does. The explanation is shown to the user as part of the result and serves both as documentation and as a debugging aid.

We use a state-of-the-art general-purpose LLM via API. We evaluated fine-tuning a smaller model on platform-specific query logs but the engineering cost did not justify the marginal accuracy improvement at the pilot stage.

3.3. Validation Layer

The validation layer is the most important and most complex part of the architecture. It is described in detail in Section 4.

3.4. Execution

Validated queries are executed through the existing dbt Semantic Layer gateway, which means they automatically inherit the row-level security policies, column masking rules, query cost ceilings, and audit logging that the SQL surface already enforces. The user identity is propagated end-to-end, so a query produced by the LLM on behalf of an analyst is executed under that analyst's privileges, not under a service account. This is non-negotiable for governance.

3.5. Result Presentation

Results are returned to the user along with the generated SQL, the brief explanation, and an indicator of which tables were consulted. Users can edit and re-execute the SQL directly if they want to refine the query. The interface is designed to teach SQL by exposure, not to hide it.

4. The Validation Layer

A naive NL-to-SQL system executes whatever the LLM produces. This is unacceptable in a regulated environment. The validation layer enforces five categories of constraints before any query reaches the execution engine.

Schema validation. Every table and column referenced in the generated SQL must exist in the catalog. Hallucinated columns the most common LLM failure mode in our pilot are caught here and rejected. The validator parses the SQL, walks the AST, and checks each identifier against the catalog. If an identifier is not found, the system attempts a single repair pass: it surfaces the closest matching real columns to the LLM and asks for a corrected query. If the repair fails, the query is rejected with an actionable error message.

Governance validation. The validator confirms that the user has read access to every table referenced. Even though dbt Semantic Layer will enforce row-level security at execution time, catching the violation at validation time produces a better user experience and avoids consuming compute on queries that will fail. Column-level masking is checked similarly: if the user does not have access to an unmasked version of a column, the validator ensures the query does not depend on unmasked values.

Cost ceiling. The validator estimates the cost of the generated query using table statistics and rejects queries that exceed a per-user, per-day budget. This prevents an analyst from accidentally launching a multi-hour scan over a fact table.

Write protection. The system is read-only. Any DML or DDL statement (INSERT, UPDATE, DELETE, CREATE, DROP, ALTER) is rejected unconditionally. The LLM is instructed not to produce them, and the validator enforces this even when the LLM ignores the instruction.

Post-hoc rewriting. Some LLM outputs are syntactically valid and semantically intended but stylistically problematic unqualified column references in a multi-table join, ambiguous CTEs, missing column aliases. The rewriter normalizes these patterns into canonical form before execution. The rewriter is conservative and only changes things it can prove are safe.

A simplified pseudocode of the validation pipeline:

- def validate(sql, user, schema):
- ast = parse(sql)
- if any_dml_or_ddl(ast): reject("read-only")
- refs = collect_table_and_column_refs(ast)
- for r in refs:
- if not schema.exists(r): return repair_or_reject(sql, r)
- for t in refs.tables:
- if not user.can_read(t): reject("permission")
- cost = estimate_cost(ast, schema)
- if cost > user.daily_budget_remaining: reject("cost")
- return rewrite_canonical(ast)

5. Pilot Evaluation

A four-week pilot was conducted with a population of approximately 30 business analysts drawn from the target unserved segment. Participants were given access to the NL interface, brief training (30 minutes), and the option to fall back to filing data tickets as before.

5.1. Adoption

75% of pilot participants ran at least one NL query per week throughout the pilot. The 25% who did not adopt cited two reasons in roughly equal measure: their questions were too complex to express in natural language ("I need a five-table join with custom date logic"), or they preferred the predictability of waiting for an engineer-written query they had used before.

This compares to the platform-wide 80% dbt Semantic Layer adoption among the technical population, suggesting that a similar diffusion pattern is achievable for the non-technical population given enough time and the right onboarding.

5.2. Accuracy

Query accuracy defined as the produced SQL returning the answer the user actually wanted was approximately 80%. Accuracy was measured by post-hoc review of a stratified sample of queries against the user's stated intent. The 20% inaccuracy decomposed as follows:

Failure category Share of failures ---------- ---------- Hallucinated column reference ~35% Wrong join (correct tables, wrong relationship) ~25% Misinterpreted aggregation level ~20% Date/time semantics wrong ~10% Other ~10%

The validation layer caught most hallucinated column references before execution, but a residual category survived: the LLM chose a real column that was not the one the user intended. These are hard to detect automatically because the query is syntactically and semantically valid.

5.3. Latency

End-to-end median latency was approximately 1.2 seconds, decomposed as approximately 200 ms for schema retrieval, 800 ms for LLM generation, 50 ms for validation, and 150 ms for query plan construction in dbt Semantic Layer. The actual data scan latency varied by query and is excluded from this measurement.

The 1.2-second median is well within the threshold at which an interactive interface feels responsive. The p99 was approximately 4 seconds, dominated by occasional slow LLM responses.

5.4. Challenges and Limitations

SQL generation accuracy remains the primary technical challenge. While the 80% end-to-end accuracy is acceptable for self-service analytics, the distribution of failures reveals opportunities for improvement. Hallucinated column references account for 35% of failures, indicating that the LLM occasionally generates references to columns that do not exist in the schema. This occurs despite the validation layer that checks generated SQL against the schema; the LLM sometimes invents plausible-sounding column names that resemble real patterns in the data warehouse. For example, when asked for “customer satisfaction trends,” the LLM might generate a reference to a ‘satisfaction_score’ column that does not exist, even though the schema contains ‘nps_score’ and ‘feedback_rating’. Improving this requires either fine-tuning the LLM on the specific schema (expensive and vendor-locked), or engineering more sophisticated schema retrieval techniques that surface semantic relationships between columns rather than simple name matching.

Ambiguous natural language queries present a subtler challenge. When users ask for “revenue by region,” they may intend aggregation by revenue region (a business dimension), geographic region (the customer location region), or operational region (where the transaction settled). The LLM has no way to disambiguate without additional context. We attempted to address this through multi-turn clarification: if the LLM detected ambiguity in the query, it would ask the user to clarify before executing SQL. However, this reduced the feel of “zero-friction” query execution; users complained that they were sometimes asked to clarify queries they felt were unambiguous. A better approach may involve analyzing historical query patterns from the same user and using those as context to disambiguate current queries, but this requires maintaining per-user query history and was outside the scope of the current pilot.

The cold-start problem for new schemas emerged during the pilot when a new data product was added to the platform. For existing schemas (the core customer and transaction schemas), the LLM had extensive training context accumulated over months of historical queries and column examples. For the new schema with only a few dozen tables, the LLM generated consistently lower-quality SQL because it lacked contextual information about column semantics and relationships. This suggests that NL-to-SQL systems may require a bootstrapping period where manual examples or light curation is necessary before the system becomes fully self-service for new schemas. Organizations adding new data products should plan for either a period of reduced accuracy, or invest in generating synthetic examples of “what good queries look like” for the new schema.

User trust calibration is a psychometric challenge often overlooked in NL-to-SQL evaluation. Users see text-to-SQL as either “magic” (no trust, expect failures) or “correct” (high trust, don’t validate). Both extremes are problematic. Users with low trust often construct test queries on small subsets of data before trusting results, effectively doubling their latency. Users with high trust occasionally run queries on the full dataset without validating results, leading to incorrect decisions that propagate downstream. One pilot participant made a critical business decision based on a hallucinated column query that returned wrong results. The participant did not notice the error until the decision had been communicated to leadership. This incident highlights the importance of user education: NL-to-SQL systems should be positioned as “helpful assistants that get it right 80% of the time and need verification when it

matters,” not as “truth machines.”

5.5. Cost

The LLM API cost during the pilot averaged a few cents per query, which is small individually but non-trivial in aggregate. Extrapolating to the SQL surface's volume of 100,000+ queries per month would imply a substantial monthly cost if natural language replaced SQL queries entirely. In practice, NL queries supplement rather than replace SQL queries, and the cost is bounded by the size of the non-technical analyst population rather than by the total query volume.

6. Limitations and Honest Discussion

Hallucinated column references remain the dominant failure mode despite schema retrieval. The retrieval narrows the candidate set, but within the candidate set the LLM still occasionally invents column names that resemble real ones. The validator catches these, but the user experience is degraded by the rejection. Improvement would likely require either tighter retrieval (more schema in the prompt) or a fine-tuned model (which we have not yet pursued).

Complex multi-table queries are still hard. The pilot population that disengaged was disproportionately the segment with the most complex questions. Natural language is good at simple questions and degrades on questions that require five-way joins, recursive CTEs, or window functions over irregular partitions. The interface is not a replacement for SQL expertise on complex analyses; it is a complement for the long tail of simple questions.

Cost adds up. At pilot volumes, the LLM cost is manageable. At full deployment to a larger analyst population, the cost would need to be tracked and optimized through model selection (smaller models for simpler questions), caching of recent answers, and rate limiting.

Governance complexity dominates the system. The natural-language layer is, in lines of code, the smallest part of the system. The validation layer, the schema retrieval index, the integration with dbt Semantic Layer for governance enforcement, and the audit logging are collectively much larger. Teams considering this work should budget accordingly: building the LLM integration is the easy part.

Audit and regulatory readiness. Every NL query, the SQL it produced, the validation outcome, the user identity, and the execution result are logged to the platform's audit zone with the standard 7-year retention. Regulators reviewing the system will see exactly what an analyst asked, what query ran on their behalf, and what data was returned. We treat this auditability as a feature, not a constraint.

The 20% accuracy gap is significant. A 20% wrong-answer rate is acceptable for an exploratory tool where users can verify the generated SQL before trusting the result, but it is not acceptable for a tool that feeds into automated decision-making. The interface is positioned as exploratory by design, and the UI emphasizes the generated SQL prominently so that users have an inspection point.

7. Conclusion

This paper has presented a production-grounded case study of building a natural-language analytics interface on top of an enterprise data lake with 1,300+ student engagement and curriculum datasets, an existing SQL surface serving 100,000+ queries per month at 80% adoption among technical users, and a structurally excluded population of approximately 20% non-technical analysts. The architecture combines schema retrieval, LLM-based SQL generation, a multi-stage validation layer that enforces governance constraints, and execution through the existing dbt Semantic Layer gateway so that row-level security and audit are inherited automatically. Pilot results showed 75% adoption among target users, 80% query accuracy, and 1.2-second median latency a profile that justifies the investment for the non-technical population while remaining honest about the failure modes.

The principal lessons are three. First, the LLM is the easy part; the validation layer and the governance integration are the work. Second, hallucination is the dominant failure mode and is best addressed by validation that catches it before the user sees a wrong answer. Third, natural language complements SQL rather than replacing it; the right framing is "expand the analyst population" rather than "deprecate the SQL surface."

Future work includes fine-tuning a smaller model on platform query logs to reduce both cost and hallucination rate, retrieval augmentation with column-level statistics so that the LLM knows the cardinality and value distribution of relevant columns, and clarification dialogue that asks the user to disambiguate when the system has low confidence in its interpretation. The broader research direction is closing the gap between benchmark text-to-SQL accuracy (high) and production text-to-SQL accuracy (lower) by modeling the constraints schema scale, governance, ambiguity that benchmarks do not capture.

References

[1] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning," 2017. https://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," in Proc. EMNLP, 2018.

[3] M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction," in Proc. NeurIPS, 2023.

[4] OpenAI, "GPT-4 Technical Report," 2023. https://openai.com/research/gpt-4

[5] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in Proc. NeurIPS, 2020.

[6] A. Vaswani et al., "Attention Is All You Need," in Proc. NeurIPS, 2017.
[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in Proc. NAACL, 2019.
[8] M. Armbrust et al., "Delta Lake: cloud storage table format with transactional guarantees," Proc. VLDB Endowment, 2020.
[9] Apache dbt Semantic Layer Documentation. https://kyuubi.apache.org/docs/latest/
[10] Apache Spark Documentation. https://spark.apache.org/docs/latest/
[11] Databricks Unity Catalog Documentation. https://docs.databricks.com/data-governance/unity-catalog/
[12] Regulation (EU) 2016/679 (General Data Protection Regulation, GDPR).
[13] Sarbanes-Oxley Act of 2002, Public Law 107-204, 116 Stat. 745.
[14] D. Sculley et al., "Hidden Technical Debt in Machine Learning Systems," in Proc. NeurIPS, 2015.
[15] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data Lifecycle Challenges in Production Machine Learning: A Survey," SIGMOD Record, 2018.
[16] Apache Software Foundation (2024). Apache Iceberg Table Format Specification v2. Technical Documentation.
[17] Shankar, S., et al. (2024). Operationalizing Machine Learning: Challenges and Best Practices. IEEE Software, 41(2), pp. 42-51.