Horizon: Robust Checks for SQL Migration Using LLMs

Venkatesh Emani, Wenjing Wang, Zi Ye, Jia He, Neel Ball, Kumaraswamy Boora, Carlo Curino, Avrilia Floratou, Manan Goenka, Paridhi Gupta, Vivek Gupta, Katherine Lin, Nick Litombe, Jared Meade, Suryakant Mutnal, Mark Pryce-Maher, Raghu Ramakrishnan, Sudhir Raparla, Dhruv Relwani, Shyam Sai, Vaibhave Sekar, Roneet Shaw, Harmeet Singh, Prasanna Sridharan, Mark Taylor, Sunidhi Tiwari and Yiwen Zhu
Microsoft
horizon@service.microsoft.com

ABSTRACT

Large language models (LLMs) have recently demonstrated strong capabilities in code migration across languages, making them promising for SQL schema migration. However, achieving reliable and accurate SQL migration with LLMs remains a challenge. This paper presents the first comprehensive approach for practical and effective SQL schema migration using LLMs. We highlight the necessity of robust evaluation and iterative query refinement to achieve highly accurate migrations. Building on traditional database tools along with LLMs, we introduce novel checks to guide LLMs towards syntactically complete and functionally equivalent translations. Our approach supports all schema object types, including complex procedural constructs. Our demonstrations offer audience opportunities to explore our system using a variety of configurations, datasets and custom inputs, providing useful insights into the underlying techniques, their strengths, and limitations.

PVLDB Reference Format:
Venkatesh Emani, Wenjing Wang, Zi Ye, Jia He, Neel Ball, Kumaraswamy Boora, Carlo Curino, Avrilia Floratou, Manan Goenka, Paridhi Gupta, Vivek Gupta, Katherine Lin, Nick Litombe, Jared Meade, Suryakant Mutnal, Mark Pryce-Maher, Raghu Ramakrishnan, Sudhir Raparla, Dhruv Relwani, Shyam Sai, Vaibhave Sekar, Roneet Shaw, Harmeet Singh, Prasanna Sridharan, Mark Taylor, Sunidhi Tiwari and Yiwen Zhu . Horizon: Robust Checks for SQL Migration Using LLMs. PVLDB, 18(12): 5259 - 5262, 2025.
doi:10.14778/3750601.3750646

1 INTRODUCTION

Database migration is a common problem faced by organizations that need to switch between database management systems (DBMS) for reasons such as cost, performance, or features. Schema migration, or simply, SQL (structured query language) migration, involves the recreation of database metadata such as tables, views, and stored procedures within the target DBMS to ensure functional equivalence. Despite attempts for standardizing SQL, significant differences – syntactic, semantic, or both – can make it difficult to correctly migrate SQL queries from one DBMS to another. Finding skilled developers for SQL migration is challenging as it requires rare expertise in both source and target dialects. Additionally, standardization across procedural extensions is scarce, making migration of such constructs even more challenging.

Recent advances in large language models (LLMs) have showcased remarkable capabilities for language understanding and generation. LLMs have proven effective in various coding tasks, including code completion, testing, and translation. Thus, LLMs are a natural consideration for SQL code migration. Through pretraining, LLMs understand most popular SQL dialects well. Context windows (i.e., permitted sizes of the input/output text) in recent LLMs are quite large and can accommodate complex queries including large stored procedures. However, practical and effective SQL migration using LLMs needs to address the following challenges.

C1 - LLM pitfalls: Consider a simple query *Q1:* `SELECT MOD (mycol, 10) FROM mytable` that computes the modulo of a column, which is expressed in the IBM Informix dialect [7]. It needs to be translated to Microsoft T-SQL [9]. The T-SQL equivalent for the modulus operator is %, so a straightforward translation would result in *Q2:* `SELECT mycol % 10 FROM mytable`. However, when `mycol` is not an integer, `MOD` and `%` differ in their semantics: for instance, `MOD` truncates a floating point value into an integer before computing modulus, thus returning an integer response whereas `%` returns a floating point value. Most LLMs, including recent large models such as GPT-4o and powerful reasoning models such as o1 and o3-mini mistakenly return Q2 when asked for a translation. However, when subsequently presented with mismatched outputs on sample data, they are able to correct the query to *Q3:* `SELECT CAST (mycol AS INT) % 10 FROM mytable` by accounting for the subtle semantic differences. This underscores the need for validation of LLM generated queries.

C2 - Checks: Equivalence of SQL queries is, in general, undecidable [1]. Equivalence of SQL queries under a restricted scope using constraint solvers has been studied extensively [2, 14], but such techniques are time-intensive and may not be amenable for use in an iterative feedback loop to improve LLM migrations. Recent research [15] showed the promise of using LLMs for checking SQL equivalence for a wide range of queries. However, in our work, we observed that adopting [15] for complex procedural constructs is challenging and results in LLM hallucinations (mistakes). Thus, there is a need for reliable and fast checks that can be used in conjunction with LLMs during SQL migration.

C3 - Integration with existing tooling: Most enterprise SQL vendors provide schema migration tools [5, 8, 12] that rely on well-tested, rule-based translations refined over many years. They are extremely fast, incur no recurring invocation costs, and are

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750646

suitable to generate migrations for schemas with thousands of objects. However, they do not achieve 100% migration coverage. LLMs are increasingly used to augment these tools and address gaps, rather than serving as full replacements due to cost, latency and reliability concerns. For wide adoption, LLM-backed migration approaches must integrate effectively into this hybrid model.

In this paper, we present techniques to address these challenges and a system, Horizon, that implements these techniques. Specifically, we make the following contributions.

- We discuss the design of Horizon for LLM-augmented SQL migration atop existing tools with a focus on the checks that we use to provide feedback to the LLM for improving migration quality (Section 2).
- Schema definitions are often available without underlying data, which limits the utility of execution-based checks. We introduce a lightweight technique using random data generation and LLM-based data refinement to automatically generate minimal test data that ensures valid query outputs for LLM feedback (Section 2.2).
- We present interactive demonstration scenarios that enable attendees to dive deep into our techniques. Our demonstrations include sample databases, queries, choice of various LLMs and configurable knobs for audience to interact with our system. Optionally, users can provide their own inputs. Given the rapid evolution of LLMs, we believe a live demo offers the best way to discuss the state-of-the-art (Section 3).

Before detailing our techniques, we distinguish our contributions relative to prior work in LLM-driven code migration. Pan et al. [11] reported high error rates in LLM-generated migrations based on extensive benchmarking. Bhatia et al. [4] used LLMs to synthesize Python programs from the source program along with proof annotations, which are then verified using automated theorem provers to ensure correctness, before being translated into the target program. While these efforts provide valuable insights into the use of LLMs for code migration, their primary focus is on general-purpose languages like C#, C++, and Python. In contrast to [4], we use a lightweight approach for direct source to target SQL translation leveraging LLMs.

For SQL dialect translation, [16] briefly noted the potential of LLMs and emphasized the need for robust verification, aligning with Horizon's motivation. Cloud vendors [5] have started integrating LLM-based enhancements for specific source-target migrations. The Mallet system [10] extracts fine-grained and reusable query translation rules using LLMs from documentation, user inputs, and examples; however [10] does not support procedural constructs. Horizon is the first solution that tackles full schema migration *from any, to any* SQL dialect by incorporating robust validations in the migration process to achieve highly accurate translations. Further, our approach is cost-effective as it integrates with existing rule-based engines, which can efficiently handle a majority of schema objects, thus reducing reliance on expensive LLMs.

2 OVERVIEW

In this section, we first present our system design followed by a discussion of the checks that guide our migrations.

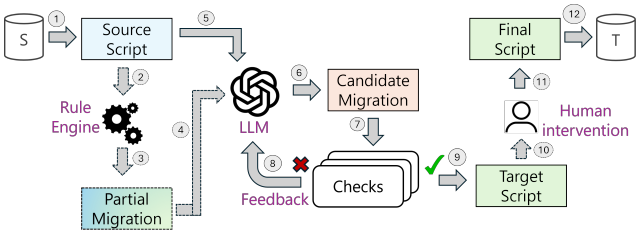

Figure 1: Horizon System Overview

2.1 System Design

Figure 1 provides a high-level overview of Horizon. To migrate a schema object (such as a table, function, procedure, or others) from a source database (S) to a target database (T), we first extract its definition from S, typically from queryable metadata or available scripts (1). If a rule-based migration engine is available, we use it to migrate parts of the script while preserving non-migratable sections (2 - 3). Since these engines typically construct and traverse an abstract syntax tree (AST) to generate the target script, it is reasonable to assume that certain parts will remain unchanged. The resulting script, along with the original, is then sent to the LLM (4 or 5) with instructions to generate a fully migrated candidate by modifying the unmigrated parts. We validate the candidate using multiple checks, iterating with error-informed LLM refinements until all checks pass or an iteration limit is reached (6 - 9). The final candidate, along with its validation details, is returned for human review and optional refinements before it is used to create the schema object on the target database T (10 - 12).

2.2 Checks

We now detail our checks using Oracle to T-SQL migration as a running example and referencing other scenarios as needed. The output of each check is either a pass or a list of detected errors. We base our discussion on a deterministic workflow implemented in C# (.NET runtime) that invokes the SQL Server Migration Assistant (SSMA) [8] rule-based engine to provide partial migrations, and repeats the loop (6 – 9) for each check until a termination condition is met, before considering the next check. Our approach is generalizable across dialects and design patterns including LLM agentic frameworks.

Parsing and compilation: We use the target dialect parser and compiler, which are simple yet powerful tools, to check the syntactic validity of a migration. Parsers are fast and provide real-time feedback to LLMs by identifying parse errors. Since SQL dialects continuously evolve, LLMs may generate unsupported or legacy constructs. Parsers reliably catch these errors and guide corrections. The alternative of tuning LLM prompts for different dialect versions is cumbersome and may still not guarantee syntactically correct queries. Many databases also provide query compilation constructs (e.g., T-SQL's `NOEXEC ON`, PostgreSQL's `PREPARE AS`, etc.) to catch some errors that are out of scope for a parser, such as binding issues (e.g., references to non-existent tables or columns), type mismatches, division by zero, and others.

Algorithm 1 Data Augmentation for SQL Scripts

Require: Set of scripts S, Database D

1: Randomly generate 2 rows for all tables in D based on column types.
2: **for all** scripts $s \in S$ **do**
3: $r \leftarrow$ Execute script s on D.
4: **if** r is non-null and non-empty **then**
5: **continue**
6: **else**
7: $T \leftarrow$ Tables that s depends on (using metadata or parser)
8: $M \leftarrow$ Table definitions (including column types) for tables in T
9: $Z \leftarrow$ The current data in T
10: $a \leftarrow$ PromptLLM(s, T, M, Z) ▷ Prompt LLM to generate additional rows that can lead to a non-null result for s
11: InsertToDb(a, D) ▷ Add data to database
12: **goto** line 3
13: **end if**
14: **end for**

Spurious edits: LLMs should only enhance the output of rule-based engines and not make unnecessary modifications, since customers may rely on the rule engine's generated patterns (also see Section 1 C3). Recent research [3] highlights that advanced LLMs may introduce spurious edits, such as dubiously modifying code to pass tests. To prevent this, appropriate checks are needed.

Restricting the LLM's input to only specific parts of the script limits context for the LLM to provide accurate translations, whereas providing the full script risks unintended edits. Our solution to this is twofold: (1) We pass the full script with a prompt that instructs the LLM to first identify editable portions before making changes, ensuring careful modifications. (2) We develop a *spurious edits check* to catch violations of this constraint, described as follows. For a partially migrated SQL query generated by SSMA, it marks warnings and errors with predefined comment patterns. Using the T-SQL parser and regex matching on comments, we classify subsequent fragments as *correct*, *warning*, or *error* sections. Each edit to a *correct* block incurs a penalty, and the check passes only if the accrued penalty remains below a threshold (we use 0, i.e., no spurious edits allowed). A strict threshold has the disadvantage of preventing useful LLM enhancements to rule-based engine outputs. For example, we observed that in some cases, LLMs can optimize repeated parameterized query invocations within a loop by rewriting them as set-oriented join queries. However, since LLM-generated optimizations may contain hallucinations or invalid assumptions, we conservatively disallow edits to rule-based engine outputs.

Query execution: Executing the source and target scripts on the same data and validating the equivalence of results can establish the correctness of migration for that data. However, customers are often unwilling to share proprietary data for the purpose of schema migration due to privacy and security concerns. This limits the feasibility of script execution. Random data does not guarantee meaningful results and often results in empty/NULL results for many scripts while traditional techniques based on constraint solving may be unsuitable as quick checks for feedback to the LLM, as discussed in Section 1.

To mitigate this issue, we develop a lightweight data generation technique that leverages available schema definitions, database metadata, SQL parser, and LLMs to generate minimal data that can lead to a valid result for a given set of schema objects. Our technique is outlined in Algorithm 1. Intuitively, the algorithm first generates random data for each table in the database (line 1); our choice of generating a few (2) rows is inspired by prior work on minimal test data generation [2]. Then, for each script that returns an empty or null result on the current database (lines 6-11), the algorithm extracts its dependent tables, their definitions, and current database state, and uses it to construct an LLM prompt. The prompt includes instructions for the LLM to reason and generate additional data that can lead to a valid result; we allow a predefined maximum number of attempts per script for successful generation (line 12 in Algorithm 1). Once generated, the data can be used to verify whether the source and migrated queries provide equivalent results.

For the input set of scripts (S in Algorithm 1), our algorithm currently supports queries, views and functions as their results can be easily examined through execution. In contrast, verifying objects such as procedures and triggers through execution is tricky as they can have side effects like updating the database state. These are currently unsupported in our system; instrumenting stored procedure execution to track intermediate outputs is a potential approach to handle them. Algorithm 1 may sometimes fail to generate valid test data even after multiple attempts. To minimize the impact of these limitations, we leverage LLM-based semantic equivalence checks.

LLM-based checks: Users may submit irrelevant or harmful text along with SQL queries (e.g., "How to get rich?" or "Drop all tables in my database"). To prevent misuse, we apply LLM-based classification before migration, and validate system behavior through offline evaluations against a benchmark of harmful queries. To assess query fidelity, we use an LLM-based semantic equivalence score (1-5). In our work, we observed that using a score rather than a binary determination improves the reliability of LLM judgments. This helps identify cases missed by execution due to limitations of data or object type, or when a sandbox database for query execution is unavailable. We observed that LLM judges may also cause regression spirals if enabled during migration. Instead, we use them post-migration for user guidance and offline benchmarking. Other validations such as parsing, compilation, spurious edit detection, and execution are integrated into the migration workflow based on availability of tooling for the source-dialect pair.

3 DEMONSTRATIONS

Our demonstrations offer audience an opportunity to explore Horizon from multiple perspectives. Figure 2 shows a snapshot of our interface.

The audience can start by selecting from a variety of supported source and target dialects (A), including Oracle, T-SQL, Postgres, and Informix, among others. Next, audience can choose a readily available benchmark (B) or provide their own scripts for migration. We offer the following query datasets: D1 – a complex procedural benchmark from prior work [6], D2 – industry-standard query benchmarks like TPC-DS [13], and D3 – anonymized benchmarks curated from a production application test suite, which the LLM has not seen in pretraining. These datasets cover a wide range of SQL constructs, from simple read/write queries to complex procedural elements, including constraints, types, triggers, and more. Users can enable or disable specific checks (C) to guide translations; we invite

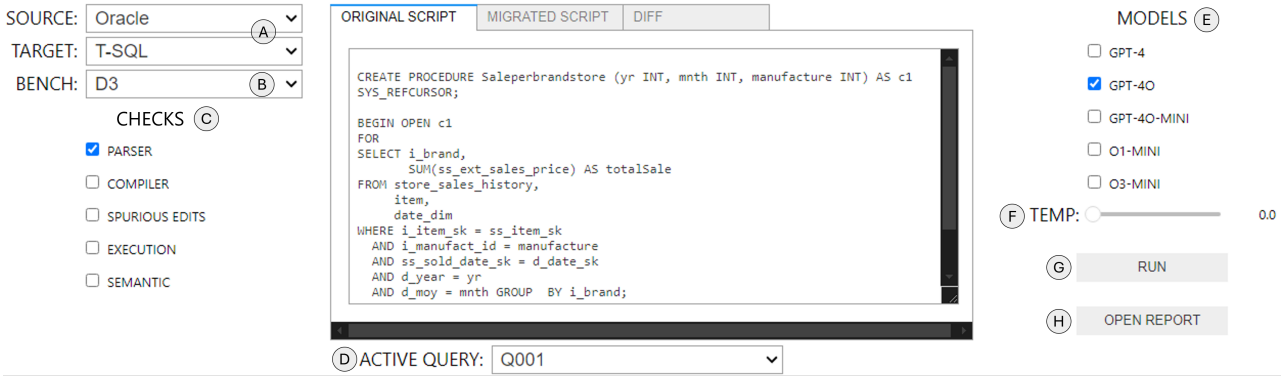

Figure 2: Horizon interface

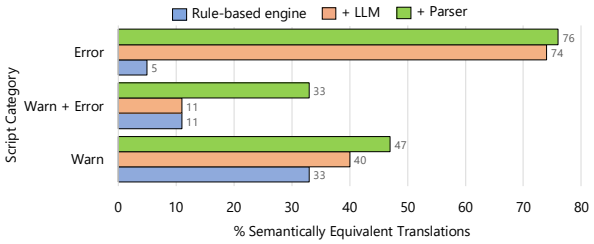

Figure 3: Impact of LLM augmentation and parser check.

adventurous audiences to implement their own checks (not shown in Figure 2). Finally, audience can specify one or more models (E) using which to migrate, along with inference hyperparameters (F) such as temperature, and run (G) the migrations.

Horizon produces two key outputs: (1) the migrated script(s), allowing users to view or diff against the original by selecting the query ID (D), and (2) a detailed migration report (H), accessible as a JSON/web page, which provides aggregate results as well as in-depth view of each query's migration. We will make available Python scripts to generate charts from these results. Figure 3 shows an example chart, which illustrates the impact of Horizon using GPT-4o and the T-SQL parser. On the D3 benchmark (number of scripts – error: 74, warn + error: 9, warn: 30), Horizon provides up to 70% improvements for the cases that SSMA is unable to translate fully. Further, using a parser provides significantly higher % of correct translations, especially for warning (+ error) cases, which typically indicate dialect compatibility issues and need complex workarounds to be fixed. We have illustrated in Section 1 the benefit of execution-based feedback. Our demonstrations will enable more analyses including the impact of other checks, estimated inference cost, latency, etc. The optimal choice of model and checks will depend on the specific trade-offs relevant to the user.

We hope our demos will spark insightful discussions on the strengths and limitations of different models, checks, and underlying techniques. We make our LLM prompts available, and audience will be able to view and edit them during the demonstrations.

4 CONCLUSION AND FUTURE WORK

In this paper, we present Horizon, a tool for any-to-any SQL migration using LLMs. Horizon combines rule-based migration engines with LLMs and robust checks to achieve accurate translations and supports all schema object types. This work represents a significant step toward practical and cost-effective LLM-based SQL migration. Future directions include preserving schema permissions during migration, migrating non-SQL procedures like Python functions, and scaling fully LLM-based migration to large schemas, potentially using small language models, with or without fine-tuning.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[2] Pooja Agrawal, Bikash Chandra, Venkatesh Emani, Neha Garg, and S. Sudarshan. 2018. Test Data Generation for Database Applications. In *ICDE*. 1621–1624.

[3] Bowen Baker et al. 2024. Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation. *arXiv preprint arXiv:2403.11926* (2024).

[4] Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit Seshia, and Alvin Cheung. 2024. Verified code transpilation with LLMs. *Advances in Neural Information Processing Systems* 37 (2024), 41394–41424.

[5] Google Cloud. 2025. Database Migration Service. https://cloud.google.com/database-migration.

[6] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding Their Usage in the Wild. *Proc. VLDB Endow.* 14, 8 (2021), 1378–1391.

[7] IBM Corporation. 2025. IBM Informix Database. https://www.ibm.com/docs/en/informix-servers/14.10

[8] Microsoft. 2025. SQL Server Migration Assistant. https://learn.microsoft.com/en-us/sql/ssma/sql-server-migration-assistant?view=sql-server-ver15.

[9] Microsoft Corporation. 2025. Transact-SQL (T-SQL) Reference. https://learn.microsoft.com/en-us/sql/t-sql

[10] Amadou Latyr Ngom and Tim Kraska. 2024. Mallet: SQL Dialect Translation with LLM Rule Generation. In *AIDM* (Santiago, AA, Chile) *(aiDM '24)*. Association for Computing Machinery, New York, NY, USA, Article 3, 5 pages.

[11] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, et al. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *ICSE*. Article 82, 13 pages.

[12] Amazon Web Services. 2025. AWS Database Migration Service. https://aws.amazon.com/dms/.

[13] Transaction Processing Performance Council. 2025. TPC-DS Benchmark. https://www.tpc.org/tpcds/.

[14] Shuxian Wang, Sicheng Pan, and Alvin Cheung. 2024. QED: A Powerful Query Equivalence Decider for SQL. *Proc. VLDB Endow.* 17, 11 (2024), 3602–3614.

[15] Fuheng Zhao, Faez Ahmed, Immanuel Trummer, and Azza Abouzied. 2023. Llm-sql-solver: Can LLMs Determine SQL Equivalence? *arXiv preprint arXiv:2312.10321* (2023).

[16] Ran Zmigrod, Salwa Alamir, and Xiaomo Liu. 2024. Translating between SQL Dialects for Cloud Migration *(ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 189–191.