Modernize Your Code Solution Accelerator

White Paper

Jessie Chen[1], Loren Henderson[1], Travis Hilbert[1], Solomon Pickett[1], Francia Riesco[1], Danielle Rubio[1], Mark Taylor[1], Bob Thrasher[1], and Venkatesh Emani[2]

[1]*Microsoft CSA OCTO*
[2]*Microsoft Gray Systems Lab (Azure Data OCTO)*

1 Introduction

Structured query language (SQL) is widely used for managing and querying relational databases, which are a common way to store and manage structured data. Despite attempts for standardizing SQL, there are significant differences in the dialects of SQL used by different database management systems (DBMSs). These differences can be syntactic, semantic, or both, and can make it difficult to migrate SQL queries from one DBMS to another. Further, most popular DBMSs also support procedural extensions to SQL that enable users to implement complex logic in the database itself. Standardization across these procedural extensions is scarce, and consequently, migrating queries that use these extensions is even more challenging.

Database migration is a common problem faced by organizations that need to switch DBMSs for reasons such as cost, performance, or features. A comprehensive migration encompasses several key phases, including the transfer of schemas, data, applications, and ultimately, user accounts and permissions. Schema migration, specifically, involves the recreation of database metadata – such as tables, views, and stored procedures – within the target DBMS to ensure functional equivalence. Stored procedures, which can encapsulate diverse and complex SQL queries within their definitions, introduce significant challenges to schema migration due to the need to accurately translate a wide range of nested SQL constructs across varying database dialects.

Traditional automation of SQL query migration encompassed the development of parser-based tools that could parse SQL queries in the source dialect, construct an intermediate representation (IR) and generate equivalent queries in the target dialect, optionally performing transformations on the IR if necessary. The scope of migrations possible in such tools may be limited in scope depending on the expressiveness of the IR, and finding an IR that can correctly express the semantics of constructs across different dialects is challenging. Further, generalizing a tool that is built for a specific pair of SQL dialects to work across multiple dialects is not straightforward, and requires significant investment; in the best case, it may require the development of an IR constructor and a query generator for each pair of SQL dialects, which in itself is a non-trivial task. For proprietary dialects, parsers may not be readily available leading to additional development and maintenance concerns for migrator tools.

Recent advances in large language models (LLMs) have showcased remarkable capabilities for language understanding and generation, extending beyond simple word prediction. LLMs have proven effective in various coding tasks, including code completion, summarization, and translation. Thus, LLMs are a natural consideration for SQL code migration as well, as they can significantly accelerate the development of SQL migration tools. A simple prompt to recent language models can accurately migrate many common SQL constructs from one dialect to another, and can handle even complex data and control flow in procedural SQL routines without additional effort. However, LLM responses can sometimes be misleading and incorrect.

Motivating Example

Consider the simple Informix query (Q1) in Figure 1 that computes the modulo of a column. The T-SQL equivalent for the modulus operator is `%`, so a straightforward translation would result in Q2. However, when `mycol` is not an integer type, `MOD` and `%` differ in their semantics: for instance, `MOD` truncates a floating point value into an

Q1: Informix SQL: `SELECT MOD(mycol, 10) FROM mytable`
Q2: Incorrect T-SQL: `SELECT mycol % 10 FROM mytable`
Q3: Correct T-SQL: `SELECT CAST(mycol AS INT) % 10 FROM mytable`

Figure 1: Subtle semantic differences between SQL queries of different dialects. Even advanced models such as GPT-4o, o1, and o3-mini provide the incorrect translation without appropriate context.

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, make this mistake when asked for a translation without additional context. However, when subsequently presented with different outputs on sample data, they are able to correct the query to Q3 by accounting for the subtle semantic differences.

This example reinforces the pitfalls of directly using LLMs for SQL migration and necessitates the need for multi-turn interactions to address potential mistakes in LLM translations. Agent-based architectures [10] leveraging LLMs and tools have emerged to tackle complex problems by decomposing them into smaller sub-problems and iteratively reasoning towards a solution.

In this report, we discuss our agentic framework for SQL migration that leverages commonly available SQL tools along with LLMs to provide reliable SQL translations. We focus on IBM Informix to T-SQL translations in our Python-based tool, but the techniques are generic and can be applied for any-to-any SQL migration and implemented in any language. We highlight the challenges in building such a framework including managing agentic interactions, the variability of SQL tooling availability for different SQL dialects and different implementation languages, and benchmarking. We note that our tool is not intended to be a final and ready solution for SQL migration; rather we provide an extensible framework that can be used as a basis and reference for building such a solution. Next, we discuss our system design and implementation choices followed by our evaluations.

2 System Design

This accelerator is designed as an agent-based system that leverages the *semantic kernel* framework [2]. Semantic kernel allows AI models to interact with external systems and APIs though plugins and skills. It supports multi-turn interactions between agents to accomplish complex tasks and automatically manages message passing between agents, and invocation of the appropriate agent based on the user input or the latest agent's response. Further, semantic kernel also allows integration with common LLM patterns such as retrieval augmented generation (RAG) through simplified memory management leveraging vector stores under the hood.

In our work, we use the Agent Group Chat [3] and Plugin [4] abstractions provided by semantic kernel. Figure 2 provides an overview of our architecture. The agent group chat pattern allows multiple agents to participate in a group chat. Given an input, each *agent*, which is assigned a specific task, provides its response. Agents are equipped with *plugins* to help with their tasks. Simply, a plugin is a function that can be invoked by

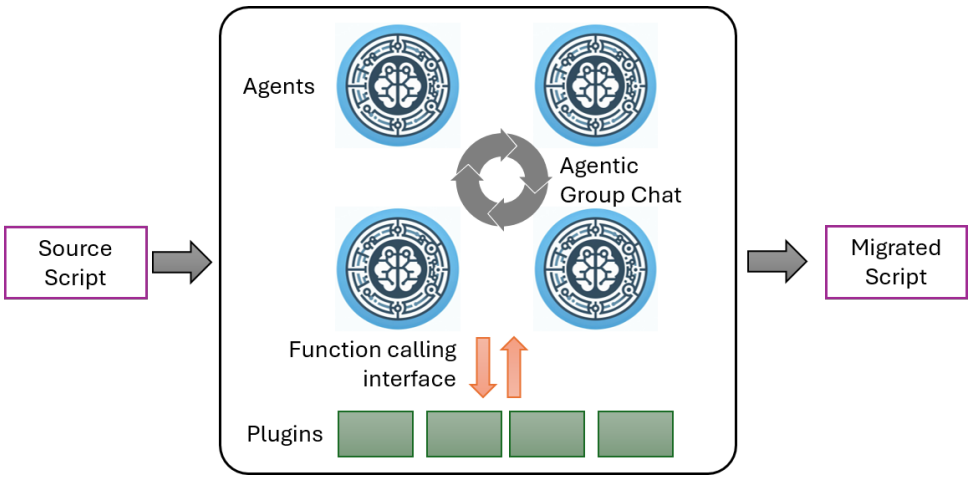

Figure 2: System Architecture

the semantic kernel framework with specific inputs and the response is fed back to an agent for next steps. Plugins can contain arbitrary code. Semantic kernel takes care of function invocation, marshaling the appropriate function inputs and responses between the agent and the plugin using the *function calling* interface [9]. A *selection function* and *termination function* (backed by LLM) determine which agent to invoke next and when to terminate the group chat. Note that while we use semantic kernel and OpenAI chat models in our work, our techniques are generic and can be implemented using other orchestration frameworks such as LangChain [6], OpenAI Agents SDK [8], etc. and can use other models.

Below, we list the agents in our system. Note that each agent can use a different model depending on the task complexity; for instance, in our implementation, we have used OpenAI GPT-4o as the default model for agents, but Syntax Checker agent (details below) uses the smaller GPT-4o-mini since it only needs to invoke the associated parser plugin and summarize the result.

- **Migrator:** This agent instructs a specialized SQL migration assistant to convert SQL queries accurately from one dialect (source) to another (target), emphasizing syntactic correctness, semantic equivalence, and careful handling of potentially harmful inputs. The agent generates multiple distinct candidate queries that could be target migrations for the given source query.

- **Picker:** This agent's task is to examine multiple candidate target queries, focusing on semantic equivalence rather than syntactic correctness. The agent must select the candidate query that best matches the original query's logic, carefully considering semantic alignment and avoiding unnecessary or additional logic.

- **Syntax Checker:** Given a query, this agent validates the syntax of the query and reports errors, if any. Instead of an LLM prompt, this agent leverages a database query parser, such as the T-SQL ScriptDom parser [1].

  We would like to highlight here that the quality of the parser can directly affect the quality of LLM migrations. This is because the errors identified by the parser are fed back to the agents to be fixed; so the better the parser is at catching errors, the more accurate the query would be. Availability of reliable parsers for all dialects in any one language is a challenge. For instance, the T-SQL ScriptDom parser is

available in the .NET runtime, and there are no equivalent Python parsers that can parse complex T-SQL queries including procedural constructs. To mitigate this issue, in our work, we illustrate how to include any tool that is an executable file (.exe on Windows or an executable binary on Linux) as a semantic kernel plugin. Users who wish to onboard other tools can adopt a similar pattern.

- **Fixer:** Given an input (erroneous) query and the errors identified by a parser, the task of the fixer agent is to emit a correct query where the specified errors have been fixed. Note that we separate identification of errors using the parser and fixing them using the fixer agent into separate steps rather than have the LLM identify as well as fix the errors; in our experience, the latter leads to unnecessary and often erroneous edits and using a deterministic tool like a parser to identify errors improves the accuracy of translations.

- **Semantic Verifier:** Syntactic correctness does not necessarily mean that the target query performs the same operations as the source query. To address this, we include a semantic verifier agent that can act as an LLM-judge to examine the logic in both the source and potential target queries and raise any semantic issues. Here we note that it is tricky to obtain the right balance between being too easy or too hard on the queries. Determining the equivalence of SQL queries over all database states is undecidable in general [5]. So the LLM-based judgement is only an approximation and can sometimes cause regressions. It is better substituted with equivalence checking by executing the queries on a sample database and comparing the results, when such data is available. To minimize variability in the final responses and avoid unnecessary changes, we include the semantic verification analysis as an additional component in the response so that users can make necessary edits based on their discretion rather than in the agent group chat.

3 Evaluations

In this section, we discuss preliminary evaluations of our tool on synthetic and benchmark queries. We use the below datasets. The benchmark queries, code for accessing relevant tools and LLM prompts are included as part of the release.

- **Simple:** This is a synthetic benchmark of seven queries that cover various SQL constructs including limit, string functions, NULL handling, procedure creation, type creation, join and procedure execution. This benchmark, although consisting of small queries, is intended to demonstrate that our tool can successfully migrate a wide range of SQL constructs.

- **Functions:** This dataset consists of ten function definitions adapted from the SQL-ProcBench [7] benchmark. SQLProcBench is a comprehensive benchmark of over a hundred procedural query definitions including scalar and table-valued user defined functions (UDFs) and stored procedures. This benchmark readily provides definitions for three popular dialects namely, T-SQL, Oracle PL/SQL and Postgresql. To demonstrate that our tool can handle complex query migrations as well, we created equivalents in Informix for ten functions from this benchmark with sizes up to around 100 lines of indented SQL code. The functions contain a mix of various

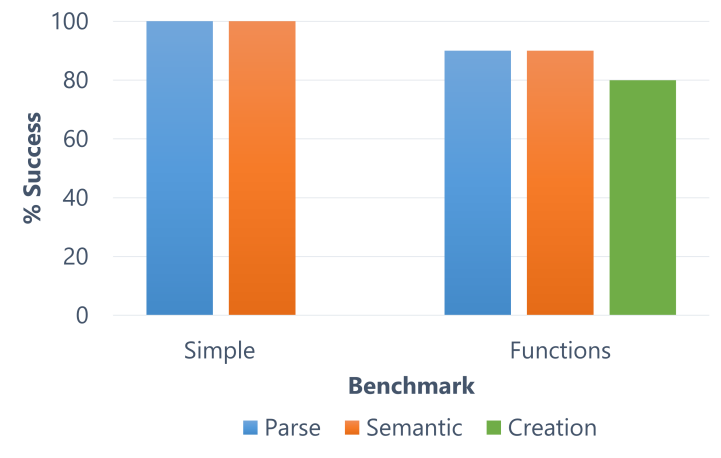

Figure 3: Percentage of successful conversions across different benchmarks. Each bar indicates the success percentage based on a metric; see description for details.

SQL constructs including complex subqueries, control and data flow, variable declaration, assignment, branching, loops, calls to other functions, use of temporary tables, etc.

Figure 3 shows the results of our tool on these benchmarks. The x-axis indicates the benchmark, and y-axis indicates the percentage of successful migrations as determined by various metrics: a successful *parse* means that the migrated query could be parsed using the T-SQL ScriptDom parser [1] without any errors, a successful *semantic* means that the migrated query and the original query are determined by an LLM judge to not have any semantic mismatches, a successful *creation* means that the migrated query could be successfully executed on the target database to create the corresponding schema object (function in this case). For each metric, we report the aggregate percentage over the entire benchmark. We limit the use of the *creation* metric to only the *functions* benchmark as *simple* contains a mix of different kinds of queries, some of which may not result in the creation of a new schema object.

As we see from Figure 3, our tool provides a high percentage of correct migrations for simple and complex queries alike. Diving into the details of failure cases in the *functions* benchmark, the system fails to provide a translation for one of the functions (f7) due to an unknown bug. For another function (f10), the system provides a migration that is syntactically correct and deemed to be semantically equivalent by the LLM judge. However, at creation time, this function fails due to the use of temporary tables (#table) inside a function, which is disallowed in T-SQL. The workaround to this issue is to explicitly create and drop the table within the function to achieve the same semantics.

The results indicate three key takeaways: (1) LLMs can be used to provide highly accurate migrations for a wide variety of simple and complex SQL queries from one dialect to another, (2) debugging agentic interactions is complex and (2) beyond syntactic and semantic correctness, other advanced checks during the migration can further improve migration quality. The release contains sample code to integrate tools as plugins into our system enabling users to implement their own checks as appropriate.

Bibliography

[1] Sql scriptdom. https://github.com/microsoft/SqlScriptDOM, 2023. Accessed: 2025-03-12.

[2] Semantic kernel: Open-source sdk for ai orchestration. https://github.com/microsoft/semantic-kernel, 2024. Accessed: 2025-03-12.

[3] How-to: Coordinate agent collaboration using agent group chat. https://learn.microsoft.com/en-us/semantic-kernel/frameworks/agent/examples/example-agent-collaboration?pivots=programming-language-python, 2025. Accessed: 2025-03-12.

[4] What is a plugin? https://learn.microsoft.com/en-us/semantic-kernel/concepts/plugins/?pivots=programming-language-python, 2025. Accessed: 2025-03-12.

[5] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. 01 1995. ISBN 0-201-53771-0.

[6] Harrison Chase and LangChain Contributors. Langchain: Building applications with llms. https://github.com/hwchase17/langchain, 2024. Accessed: 2025-03-12.

[7] Surabhi Gupta and Karthik Ramachandra. Procedural extensions of sql: Understanding their usage in the wild. *Proceedings of the VLDB Endowment*, 14(8):1378–1391, 2021.

[8] OpenAI. Openai agents sdk. https://platform.openai.com/docs/guides/agents-sdk, 2024. Accessed: 2025-03-12.

[9] OpenAI. Function calling. https://platform.openai.com/docs/guides/function-calling, 2024. Accessed: 2025-03-12.

[10] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), March 2024. ISSN 2095-2236. doi: 10.1007/s11704-024-40231-1. URL http://dx.doi.org/10.1007/s11704-024-40231-1.