UDC: 004.89 https://doi.org/10.15407/jai2026.02.087

I. Bulka[1], **B. Pavlyshenko**[2]
[1,2]Ivan Franko National University of Lviv, Ukraine
Universytetska St, 1, Lviv, 79000
[1]ivan.bulka@lnu.edu.ua
[2]bohdan.pavlyshenko@lnu.edu.ua
[1]https://orcid.org/0009-0003-2962-7931
[2]https://orcid.org/0000-0001-9515-3488

USING LLM AGENTS TO GENERATE MIGRATION SQL QUERIES IN CLOUD COMPUTING PLATFORMS

Abstract. Cloud data warehouse and queries migrations between platforms such as Snowflake and Google BigQuery present a persistent engineering challenge: while automated translation tools handle the bulk of SQL conversion, a significant fraction of transformed queries fail at execution time due to dialect divergence, type system incompatibilities, and platform-specific constraints. At scale, manually correcting these failures is time-consuming, error-prone, and does not fit into modern automated migration workflows. Each failed transformation traditionally requires a data engineer to inspect the error, reason about the root cause, produce a fix, and re-run validation, a cycle that becomes the dominant cost driver when hundreds or thousands of transformations are involved. This paper presents an LLM-based self-healing agent that automatically detects, diagnoses, and repairs failed SQL transformations. The system is implemented as a stateful ten-node directed graph using the LangGraph framework, combining large language model reasoning with Retrieval-Augmented Generation (RAG) over BigQuery-specific documentation stored in a FAISS vector index. On each iteration, the agent classifies the active error into one of six failure categories, retrieves semantically relevant documentation passages, performs structured root cause analysis, rewrites the failing query with an explanation of changes, and executes it against the live BigQuery environment. The agent was evaluated on 109 real-world failed SQL transformations from four independent migration projects. Of these, 28 involved irresolvable platform-level constraints and were correctly escalated without consuming the repair budget. On the remaining 81 addressable cases, the agent achieved a fix rate of 75.3%, with 52.5% of successful repairs resolved in a single iteration, demonstrating both effectiveness and efficiency.

Keywords: SQL, large language models, LangGraph, Retrieval-Augmented Generation, BigQuery.

Introduction

Large language models have shown strong practical capability across a wide range of knowledge-intensive tasks. Recent work has demonstrated that fine-tuned LLMs deliver reliable domain-specific reasoning in areas such as financial news analysis and disinformation detection [1, 2]. Retrieval-Augmented Generation (RAG) has further extended this capability by grounding LLM responses in externally retrieved, domain-specific knowledge [3, 4, 5, 6].

SQL query repair during cloud platform migration is a particularly promising target for this kind of automation. The task is highly repetitive at scale, error messages are structured and machine-readable, and platform-specific documentation provides a natural knowledge source for retrieval-augmented generation.

Enterprise data teams increasingly migrate analytical workloads between cloud data warehouse platforms, driven by cost optimization, vendor consolidation, and evolving capability requirements. The migration from Snowflake to Google BigQuery is one of the most popular transitions. While automated translation tools such as Google's BigQuery Migration Service can convert a large fraction of SQL queries automatically, a meaningful subset of queries fail at execution time due to semantic differences that rule-based transpilers cannot handle.

Failed transformations represent a critical bottleneck in migration pipelines. Each failure traditionally requires a data engineer to: (1) inspect the error message and the original query, (2) reason about the dialect difference or data type mismatch responsible, (3) produce a corrected query, and (4) re-execute and validate the result. At scale, this cycle becomes the dominant cost driver in a migration project.

Recent advances in large language models (LLMs) suggest a promising path toward automating this repair process. LLMs have demonstrated strong capability in code understanding, error diagnosis, and code

generation tasks [9, 10]. However, naive single-shot prompting suffers from hallucination and lack of grounding [11, 12]. An iterative, agent-based approach with access to relevant documentation via RAG offers a more robust architecture.

This paper makes the following contributions:

- A formal taxonomy of Snowflake-to-BigQuery transformation failure modes, derived from analysis of production migration data.
- A LangGraph-based self-healing agent architecture implementing an iterative correction loop with structured state management, adaptive iteration budgeting, and timeout handling.
- A Retrieval-Augmented Generation (RAG) component grounded in BigQuery-specific migration documentation using FAISS vector search with Google Generative AI embeddings.
- An empirical evaluation of the agent on 109 real-world failed SQL transformations from four migration projects, achieving a 75.3% fix rate.

Analysis of recent research and publications

LLMs have been increasingly applied to software engineering tasks, including automated program repair [13], code generation [9, 10] and bug localization [14]. Several studies have shown that models such as GPT-4 can successfully repair a substantial fraction of real-world bugs when provided with appropriate context [15]. RepairAgent [13] introduced an autonomous LLM-based agent for program repair that iteratively applies repair strategies, demonstrating the viability of agentic approaches for code correction tasks.

In the domain of SQL, LLMs have been widely evaluated on text-to-SQL benchmarks such as Spider and BIRD [20]. Cross-dialect SQL transpilation, however, has received comparatively less attention. Existing tools rely primarily on rule-based rewriting and do not address execution-time failures caused by semantic incompatibilities. SQLRepair [7] addressed SQL repair in an educational context but is limited to student-authored query mistakes.

RAG-based approaches have been shown to reduce LLM hallucinations by grounding outputs in retrieved context [16, 17]. The integration of a FAISS-based vector search [18, 19] with cosine similarity retrieval [20] provides an efficient mechanism for selecting relevant documentation passages based on semantic similarity to the current error state.

Materials and Methods

Error Taxonomy

Through analysis of failed transformations in production migration pipelines, we identified six actionable error categories. Each corresponds to a distinct root cause and requires a different remediation strategy.

Syntax Error. SQL syntax violations introduced during translation. These include incorrect string literal comparison patterns (e.g., WHERE col = STRING '1'), residual type-casting artifacts, parenthesis mismatches, and tokens that BigQuery's stricter parser rejects.

Type Mismatch. Data type incompatibilities between query logic and the actual or declared schema. This happens when Snowflake's implicit type coercions are not preserved in BigQuery, or when column types differ between the two platforms.

Undeclared Variable. References to session variables or procedural variables that are valid in Snowflake's procedural SQL but unsupported or out-of-scope in BigQuery.

JavaScript UDF Syntax Error. JavaScript User-Defined Functions present in the original Snowflake transformation that need to be rewritten to work in BigQuery's JavaScript UDF execution environment.

Column Naming Error. Invalid column aliases or references that violate BigQuery naming conventions, such as clashes with reserved keywords or duplicate aliases within a SELECT statement.

View Update Error. Attempts to perform DML operations (INSERT, UPDATE, DELETE) against BigQuery views, which cannot serve as target objects for data modification.

A single transformation may exhibit more than one of these error types at once; the

agent classifies all applicable types before beginning any repair work.

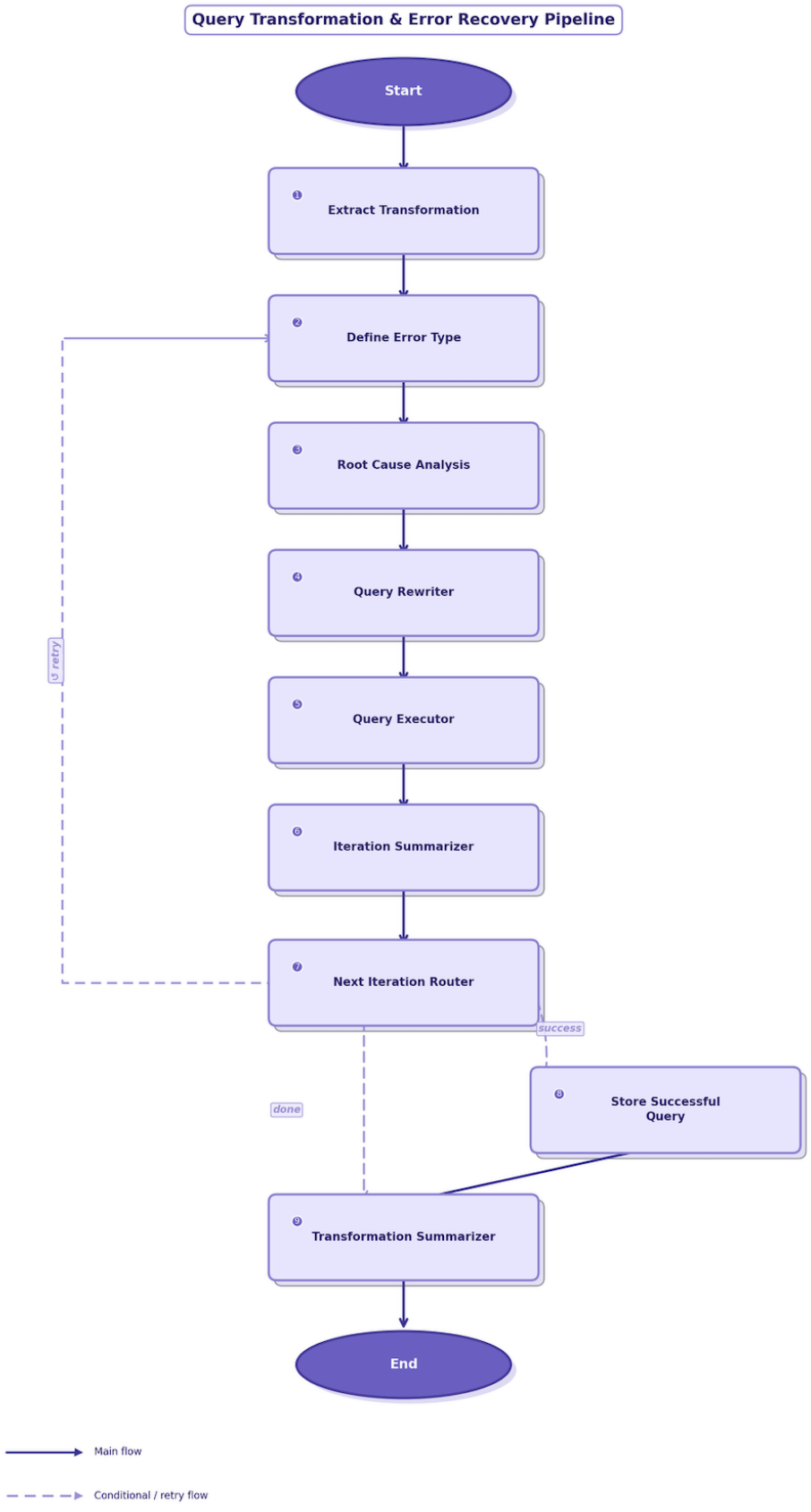

Fig. 1. LangGraph-based self-healing agent architecture

Agent Architecture

The agent is implemented as a stateful directed graph using LangGraph [9]. The agent state has two components: Agent Constant Values (fixed inputs loaded at entry: Snowflake source query, auto-translated BigQuery query, input and output table schemas, initial error message) and Agent Variables (mutable state updated across iterations: current query, iteration history, error classifications, RAG-retrieved context, flags for success/platform-issue/hanging).

The graph consists of ten nodes:

Extract Transformation is the entry node. It loads all context artifacts: the Snowflake source query, the auto-translated BigQuery query, input and output table schemas, and the initial error message. It initializes the agent state and creates the iteration storage directory.

Define Error Type invokes the LLM with a structured prompt to classify the failing query into one or more error categories from the taxonomy. It returns a typed Error Types object containing a list of error type labels and an explanatory rationale.

RAG Search generates a condensed retrieval query from the current error message using an LLM call, then performs similarity search against the FAISS knowledge base. It returns the top-5 most relevant documentation chunks ranked by cosine similarity, which are injected as additional context into the repair prompts.

Root Cause Analysis is the diagnostic core of the agent. Given the current query, error message, table schemas, iteration history, error type classifications, and RAG-retrieved documentation, the LLM produces a structured Error Analysis object containing a list of Error Root Cause entries. Each entry specifies a root cause description, the corresponding error type, and a concrete repair plan.

Query Rewriter consumes the Error Analysis from the previous node and generates a corrected BigQuery SQL query. The LLM is conditioned on the table schemas, the current error message, the root cause analysis, the iteration history, and error-type-specific prompt augmentations.

Query Executor submits the rewritten query to the target BigQuery execution environment. Execution is wrapped in a timeout mechanism with configurable retry logic to handle hanging queries. Execution time is recorded and used to adaptively reduce the remaining iteration budget.

Iteration Summarizer produces a structured Iteration Summary object after each execution attempt, capturing: the iteration number, a brief description of the issue found, the fix applied, the execution outcome, and a short error description if execution failed.

Next Iteration Router is a conditional routing node that determines the next state of the agent. It routes to Store Successful Query on success; to Transformation Summarizer on iteration budget exhaustion, detection of platform-level issues, or query hanging; and back to RAG Search for the next repair attempt otherwise.

Store Successful Query persists the successfully repaired query to disk. This file is consumed by downstream pipeline steps.

Transformation Summarizer produces a final Transformation Summary object describing the entire repair session: the initial problem, the resolution path, and the final outcome. The full agent state is persisted for auditing and diagnostics.

Termination Conditions

The agent terminates under one of four conditions:

Success. The rewritten query executes without error. The agent stores the fixed query and produces a summary.

Iteration budget exhausted. The number of attempted repairs reaches the configured maximum (default is 6). Execution time of individual attempts reduces the effective budget proportionally.

Platform issue detected. Error classification identifies a failure not addressable by SQL rewriting (e.g., DML on a view). The agent terminates and flags the transformation for human review.

Query hanging. The query executor detects a timeout across all retry attempts, setting the hanging flag and routing to the summarizer.

RAG Component

The RAG knowledge base is built from BigQuery-specific migration documentation, including the BigQuery SQL reference and Snowflake-to-BigQuery migration guide. Documents are split into fixed-size chunks with overlap and embedded using Google Generative AI text embeddings. The embeddings are stored in a FAISS index and persisted to disk for reuse across agent runs. At retrieval time, the agent generates a condensed query from the current error context and retrieves the top-5 chunks by cosine similarity.

Results

Overall Performance

The agent was evaluated on a corpus of 109 real-world failed SQL transformations from four independent Snowflake-to-BigQuery migration projects (Projects A, B, C, and D). Of these, 28 cases (25.7%) were identified by the agent's error classifier as involving irresolvable platform-level constraints — specifically View Update Errors and load-semantic issues — and were immediately escalated without attempting any repair. The remaining 81 cases were treated as addressable, and the agent successfully repaired 61 of them, achieving a fix rate of 75.3% (Fig. 2).

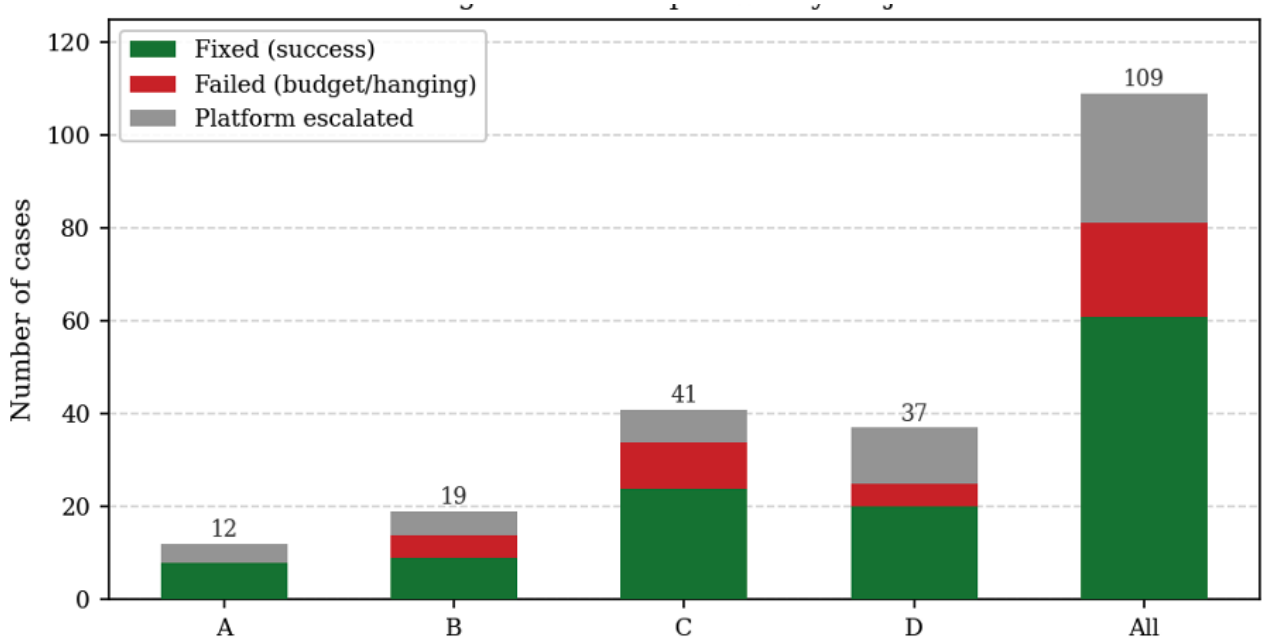

Fig. 2. Case disposition by project

Per-Project Performance

Table 1 summarizes the agent's performance broken down by migration project. Fix rates varied from 64.3% (Project B) to 100% (Project A), reflecting differences in transformation complexity and error-type composition across projects. Project D exhibited the highest share of platform-level issues (12 of 32 cases), consistent with its dataset containing more DML-against-view operations.

Table 1. Per-Project Performance

Project	Total cases	Platform issues	Addressable	Fixed	Not fixed		Fix rate (%)	Fixed in 1 iteration	1-iter share of fixed (%)
A	12	4 (33.3%)	8	8	0		100.0	5	62.5
B	19	5 (26.3%)	14	9	5		64.3	7	77.8
C	41	7 (17.1%)	34	24	10		70.6	15	62.5

Project	Total cases	Platform issues	Addressable	Fixed	Not fixed		Fix rate (%)	Fixed in 1 iteration	1-iter share of fixed (%)
D	37	12 (32.4%)	25	20	5		80.0	5	25.0
All	109	28 (25.7%)	81	61	20		75.3	32	52.5

Initial Error-Type Distribution

Fig. 3 shows the distribution of initial error types across all 109 evaluated transformations. A single transformation may exhibit more than one error type simultaneously; percentages therefore reflect the share of cases containing each type. Syntax errors and type mismatches dominated the corpus, each appearing in approximately 38% of cases. Undeclared variable errors and JavaScript UDF errors were less common but present in a meaningful fraction.

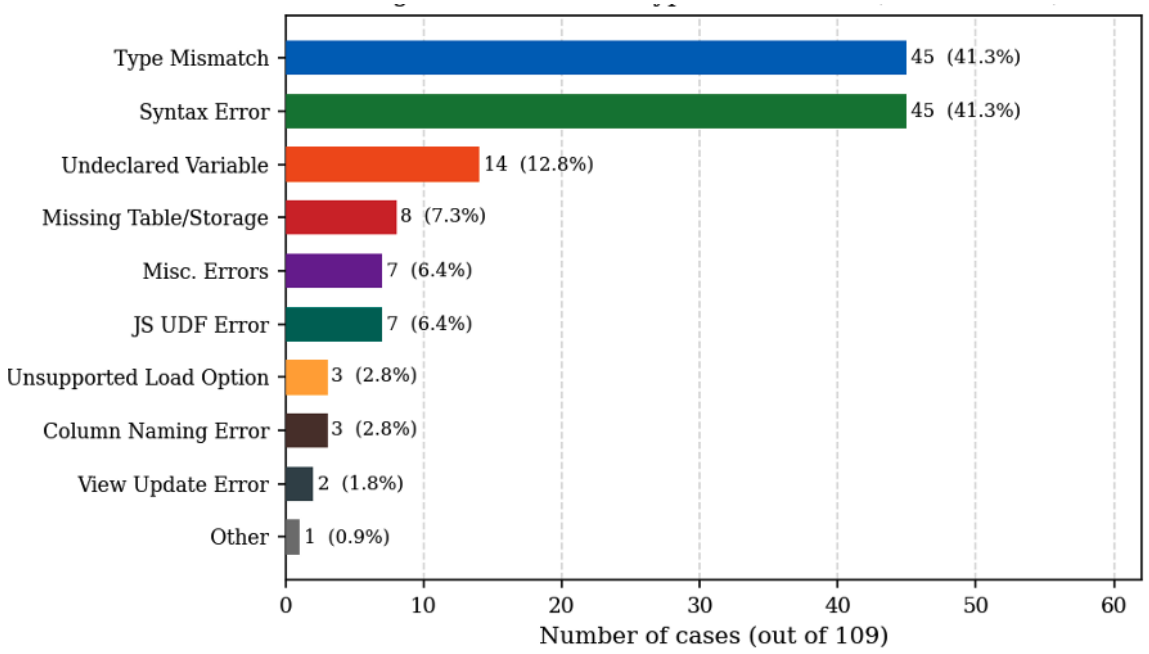

Fig. 3. Initial error-type distribution across all 109 cases

Fix Rate by Category

Table 2 presents the fix rate decomposed by initial error category, computed over addressable cases only. The agent achieved perfect fix rates on undeclared variable errors (9/9, 100%) and column naming errors (3/3, 100%), reflecting strong capability for declarative SQL restructuring. Type mismatch and syntax errors were resolved at 80.5% and 78.1%, respectively. JavaScript UDF errors achieved only a 50% fix rate, and view update errors were never resolved by rewriting (0%), consistent with their classification as platform-level issues.

Table 2. Fix Rate by Initial Error Category

Error category	Addressable cases		Fixed	Not fixed	Fix rate (%)
Type Mismatch	41		33	8	80.5
Syntax Error	32		25	7	78.1
Undeclared Variable	9		9	0	100.0

Error category	Addressable cases		Fixed	Not fixed	Fix rate (%)
JavaScript UDF Error	6		3	3	50.0
Miscellaneous	5		2	3	40.0
Column Naming Error	3		3	0	100.0
View Update Error	2		0	2	0.0

Iteration Efficiency

Fig. 4 shows the distribution of iterations required to achieve a successful fix across all 61 repaired cases. The majority, 32 cases (52.5%), were resolved in a single iteration, demonstrating that the agent's first-attempt reasoning is highly effective for straightforward dialect-translation errors. The remaining cases required between two and five iterations, with a long tail of harder cases requiring more correction cycles.

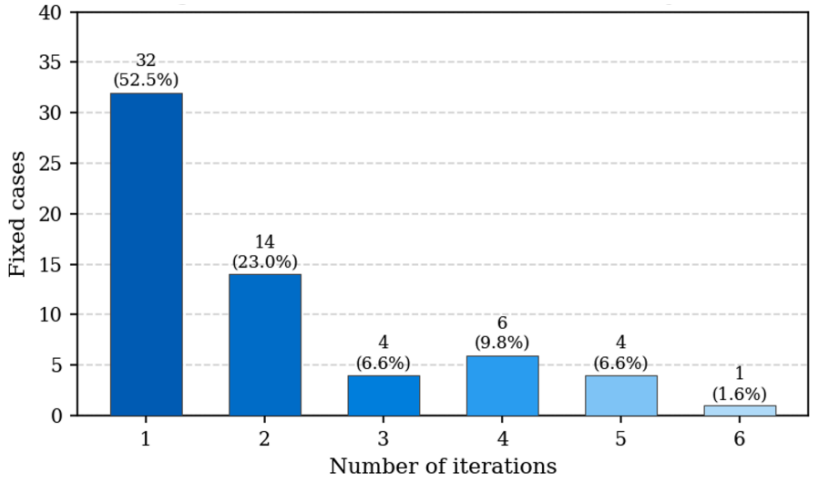

Fig. 4. Distribution of iterations to successful fix

Fig. 5 breaks down iteration counts by project. Project B shows the highest concentration at one iteration (77.8% of its fixed cases), reflecting a workload dominated by well-defined type mismatch and syntax patterns. Project D, by contrast, shows a flatter distribution extending to five iterations, consistent with more complex, multi-error transformations.

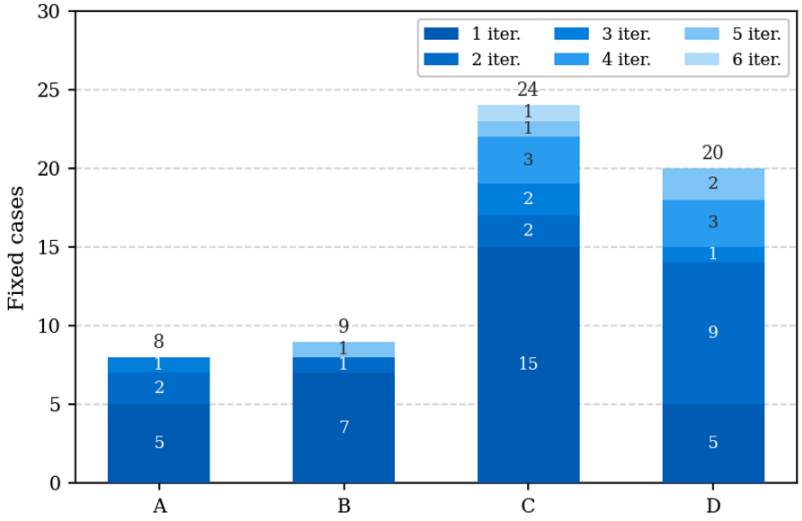

Fig. 5. Iterations to successful fix per project

Case Outcome Decomposition

Fig. 6 presents the overall outcome decomposition across all 109 evaluated cases. Successfully fixed cases account for 56.0% of the full evaluation corpus. Platform-level escalations (25.7%) represent cases correctly identified and forwarded for manual engineering review. Cases where the iteration budget was exhausted without a successful fix account for 18.3% of the total, representing the remaining gap for future improvement.

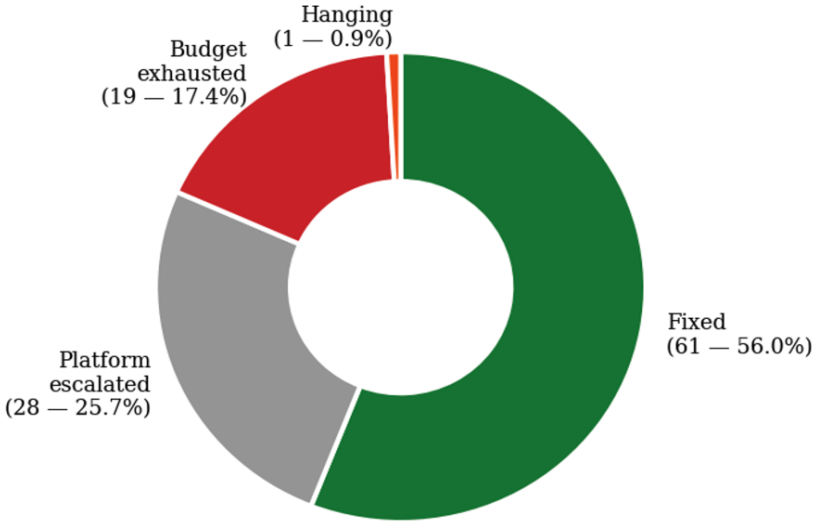

Fig. 6. Overall case outcome decomposition

Discussion

The results show that an LLM agent with iterative repair and access to platform documentation can automatically fix a large share of failed SQL migrations. A 75.3% fix rate on addressable cases, with more than half resolved in a single iteration, confirms that the approach works well for the most common failure modes.

Fix rates vary by error type, and the pattern is straightforward: the more informative the error message, the easier the repair. Type mismatch (80.5%) and syntax errors (78.1%) come with precise, structured messages that point the agent directly to the problem. Undeclared variable and column naming errors are resolved perfectly because they have both clear error messages and straightforward fixes.

Correctly escalating the 28 platform-level cases without attempting any repairs is also valuable. In a real migration project, immediately flagging a transformation as irresolvable by SQL rewriting saves engineers from wasting time investigating it themselves.

The main limitation is that all four projects come from the same migration context and use the same translation tool. How well the agent performs on different tools, query styles, or target platforms is still unknown.

One clear direction for improvement is replacing the deterministic graph with a deep research agent. Instead of searching a fixed document index, such an agent would look up live documentation, platform release notes, community forums, and code repositories on demand. This matters most for the harder cases — JavaScript UDF rewrites, complex schema mismatches — where the required knowledge is either too specific or too recently published to be fully captured in a static index.

Conclusions

This paper presented an LLM-based agent that automatically repairs failed SQL transformations during Snowflake-to-BigQuery migrations. The system combines error classification, documentation retrieval, root cause analysis, and iterative query rewriting into a single automated loop, removing the need for manual engineering intervention on the majority of failed transformations.

On 109 real-world cases from four migration projects, the agent fixed 75.3% of addressable transformations and correctly

routed all 28 platform-level failures directly to human review without spending any repair budget on them. The fact that 52.5% of successful repairs were resolved on the first attempt confirms that the RAG-grounded, error-taxonomy-driven approach provides an effective first pass for the most common failure modes.

Error taxonomy classification proved to be a critical enabler. By separating failures into distinct categories before any repair attempt, the agent can retrieve documentation that is specifically relevant to the active failure mode rather than performing a generic search. This targeted retrieval is a key factor in achieving high first-attempt success rates.

The remaining gaps point to the limits of a static knowledge base. JavaScript UDF errors, which achieved only a 50% fix rate, and view update errors, which were never resolved by rewriting, represent failure modes where the required knowledge is either too specific, too runtime-dependent, or too recently published to be fully captured in a static document index.

Other open questions include how the system performs across different translation tools and a wider variety of analytical query workloads, how much each individual component contributes to the overall fix rate, and whether the error taxonomy can be refined further to eliminate categories that are structurally unresolvable by SQL rewriting alone.

References

1. Pavlyshenko, B. M. (2023). Financial news analytics using fine-tuned Llama 2 GPT model. *arXiv preprint arXiv:2308.13032*. Available: https://doi.org/10.48550/arXiv.2308.13032

2. Pavlyshenko, B. M. (2023). Analysis of disinformation and fake news detection using fine-tuned large language model. *arXiv preprint arXiv:2309.04704*. Available: https://doi.org/10.48550/arXiv.2309.04704

3. Pavlyshenko, B. M. (2025). Multilevel Analysis of Cryptocurrency News using RAG Approach with Fine-Tuned Mistral Large Language Model. *arXiv preprint arXiv:2509.03527*. Available: https://doi.org/10.48550/arXiv.2509.03527

4. Pavlyshenko, B. M. (2025). AI Approaches to Qualitative and Quantitative News Analytics on NATO Unity. *arXiv preprint arXiv:2505.06313*. Available: https://doi.org/10.48550/arXiv.2505.06313

5. Pavlyshenko, B., & Bulka, I. (2024). Metric-based comparison of fine-tuned LLaMA 2 and Mixtral large language models for instruction tasks. *Electronics and information technologies/Електроніка та інформаційні технології*, (26). Available: http://dx.doi.org/10.30970/eli.26.2

6. Pavlyshenko, B., & Bulka, I. (2025). PARAMETER EFFICIENT FINE-TUNING AND OVERFITTING IN GPT LARGE LANGUAGE MODELS: A METRIC-BASED COMPARISON. *Electronics and information technologies/Електроніка та інформаційні технології*, (30), 33-42. Available: http://dx.doi.org/10.30970/eli.30.3

7. Presler-Marshall, K., Heckman, S., & Stolee, K. (2021, May). SQLRepair: Identifying and repairing mistakes in student-authored SQL queries. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (pp. 199-210). IEEE. Available: http://dx.doi.org/10.1109/ICSE-SEET52601.2021.00030

8. Mohammed, A. K., Kashif, M., & Ansari, S. F. Cloud-Native Framework for Enterprise Business Intelligence with AI-Driven Scalability Using Snowflake and Big Query. Available: http://dx.doi.org/10.7753/IJSEA1412.1010

9. Joel, S., Wu, J., & Fard, F. (2024). A survey on llm-based code generation for low-resource and domain-specific programming languages. *ACM Transactions on Software Engineering and Methodology*. Available: https://doi.org/10.1145/3770084

10. Mu, F., Shi, L., Wang, S., Yu, Z., Zhang, B., Wang, C., ... & Wang, Q. (2024). Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering*, *1*(FSE), 2332-2354. Available: https://doi.org/10.1145/3660810

11. Zhang, Z., Wang, C., Wang, Y., Shi, E., Ma, Y., Zhong, W., ... & Zheng, Z. (2025). Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering*, *2*(ISSTA), 481-503. Available: https://doi.org/10.1145/3728894

12. Orgad, H., Toker, M., Gekhman, Z., Reichart, R., Szpektor, I., Kotek, H., & Belinkov, Y. (2024). Llms know more than they show: On the intrinsic representation of llm hallucinations. *arXiv preprint arXiv:2410.02707*. Available: https://doi.org/10.48550/arXiv.2410.02707

13. Bouzenia, I., Devanbu, P., & Pradel, M. (2025, April). Repairagent: An autonomous, llm-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)* (pp. 2188-2200). IEEE. Available: http://dx.doi.org/10.1109/ICSE55347.2025.00157

14. Li, Z., Jiang, Z., Huang, Q., & Gu, Q. (2025, April). LLM-BL: Large Language Models are Zero-Shot Rankers for Bug Localization. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)* (pp. 548-559). IEEE Computer Society. Available: http://dx.doi.org/10.1109/ICPC66645.2025.00064

15. Ságodi, Z., Antal, G., Bogenfürst, B., Isztin, M., Hegedűs, P., & Ferenc, R. (2024, June). Reality check: Assessing GPT-4 in fixing real-world software vulnerabilities. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (pp. 252-261). Available: https://doi.org/10.1145/3661167.3661207

16. Zhang, Q., Fang, C., Xie, Y., Ma, Y., Sun, W., Yang, Y., & Chen, Z. (2024). A systematic literature review on large language models for automated program repair. *ACM Transactions on Software Engineering and Methodology*. Available: https://doi.org/10.1145/3799693

17. Bag, S., Gupta, A., Kaushik, R., & Jain, C. (2024, July). RAG beyond text: enhancing image retrieval in RAG systems. In *2024 International Conference on Electrical, Computer and Energy Technologies (ICECET* (pp. 1-6). IEEE. Available: http://dx.doi.org/10.1109/ICECET61485.2024.10698598

18. AboulEla, S., Zabihitari, P., Ibrahim, N., Afshar, M., & Kashef, R. (2025, April). Exploring RAG solutions to reduce hallucinations in LLMs. In *2025 IEEE International systems Conference (SysCon)* (pp. 1-8). IEEE. Available: http://dx.doi.org/10.1109/SysCon64521.2025.11014810

19. Ni, X., Wang, Q., Zhang, Y., & Hong, P. (2025). Toolfactory: Automating tool generation by leveraging llm to understand rest api documentations. *arXiv preprint arXiv:2501.16945*. Available: https://doi.org/10.48550/arXiv.2501.16945

20. Lai, J., Zhang, J., Liu, J., Li, J., Lu, X., & Guo, S. (2024). Spider: Any-to-many multimodal llm. *arXiv preprint arXiv:2411.09439*. Available: https://doi.org/10.48550/arXiv.2411.09439

21. Wang, Y., Zhang, Y., Li, G., Zhi, C., Li, B., Huang, F., ... & Deng, S. (2025). InspectCoder: Dynamic Analysis-Enabled Self Repair through interactive LLM-Debugger Collaboration. *arXiv preprint arXiv:2510.18327*. Available: https://doi.org/10.48550/arXiv.2510.18327

22. Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P. E., ... & Jégou, H. (2025). The faiss library. *IEEE Transactions on Big Data*. Available: http://dx.doi.org/10.1109/TBDATA.2025.3618474

23. Qin, C., Deng, C., Huang, J., Shu, K., & Bai, M. (2020, April). An efficient faiss-based search method for mass spectral library searching. In *2020 3rd International Conference on Advanced Electronic Materials, Computers and Software Engineering (AEMCSE)* (pp. 513-518). IEEE. Available: http://dx.doi.org/10.1109/AEMCSE50948.2020.00116

24. Steck, H., Ekanadham, C., & Kallus, N. (2024, May). Is cosine-similarity of embeddings really about similarity? In *Companion Proceedings of the ACM Web Conference 2024* (pp. 887-890). Available: https://doi.org/10.1145/3589335.3651526

25. Huang, X., Peng, H., Zou, D., Liu, Z., Li, J., Liu, K., ... & Yu, P. S. (2024). Cosent: Consistent sentence embedding via similarity ranking. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, *32*, 2800-2813. Available: http://dx.doi.org/10.1109/TASLP.2024.3402087

The article has been sent to the editors 13.04.26.
After processing 25.04.26.
Submitted for printing 30.06.26