Universidad ORT Uruguay
Facultad de Ingeniería

A tool for enhancing interoperability in automated LLM-assisted SQL migration

Entregado como requisito para la obtención del título de Ingeniería en Sistemas

Sebastián Daners - 255927

Tutores: Sergio Yovine y Franz Mayr

2025

Declaración de Autoría

Yo, Sebastián Daners declaro que el trabajo que se presenta en esta obra es de mi propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos el Proyecto Final de Ingeniería en Sistemas;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Sebastian Daners

16-10-2025

Agradecimientos

Dedicado a mi familia y amigos que me apoyaron y soportaron durante esta etapa.

Agradezco a mis tutores, Sergio Yovine y Franz Mayr por toda la ayuda que me dieron para completar este trabajo. Su experiencia fue invaluable.

Abstract Español

Este proyecto investiga el uso de grandes modelos de lenguaje (LLMs por sus siglas en inglés) como asistentes para la migración de código SQL de un proveedor a otro, una tarea donde la falta de portabilidad entre sistemas y semánticas y sintaxis específicas a los motores comúnmente limitan a los traductores basados en gramática. Se propone un bucle de retroalimantación guiado por errores para mejorar la respuesta de los LLMs utilizando un proceso de análisis de dos pasos. El primero se ocupa de identificar los errores sintácticos y el segundo los semánticos. Esta herramienta es probada utilizando diferentes modelos de OpenAI para demostrar su efectividad.

Abstract

This project investigates large language models (LLMs) as assistants for migrating SQL code from one vendor to another, a task where the lack of portability between systems and engine-specific syntax and semantics routinely hinder grammar-based automated translators. An error-driven feedback loop is proposed that improves the LLMs performance using a two-step analysis process. The first step identifies syntactic errors and the second one collects problems encountered during the execution of the generated code. This tool is then tested using different OpenAI models to prove its effectiveness.

Key Words

Artificial intelligence; Large language models; Prompt engineering; SQL migration; GoogleSQL; PostgreSQL; ZetaSQL; Dialect;

Index

1 Introduction

Large language models (LLM), which are key tools in tasks like code and text generation [1, 2], rely on the quality of the prompts to be able to accomplish those duties [3, 4, 5]. However, effective prompt design faces challenges such as the probabilistic interpretation of their outputs and the lack of standardized methods to evaluate their performance [6, 7]. This research work focuses on improvements in the efficiency and alignment by leveraging prompt engineering using feedback derived from an appropriate analysis of the errors incurred by the output of the LLM. In particular, it proposes a solution that addresses the problem of ensuring interoperability in LLM-aided SQL code migration between different databases.

1.1 Motivation and related work

Real-world professional work experience in code-migration projects revealed that current LLMs have a weak spot in handling SQL interoperability. The lack of portability between systems from different vendors can make LLMs struggle with generating code from or to a vendor's database they were not trained on. This was observed more specifically with GoogleSQL [8], the ANSI-compliant SQL implementation by Google that is used in their database service called Spanner [9]. Typically, migrating a schema from a different vendor to Spanner involves a multi-step process which requires manual analysis and refinement [10].

So far, no AI-powered tools have been created for SQL migration. For instance, Google's Spanner Migration Tool (SMT) is a standard yet incomplete grammar-based translator. As a matter of fact, SMT warns users that they should *'expect to tweak and enhance what the migration tool produces'* [11]. Indeed, extensive experimentation with SMT in the aforementioned real-world code-migration projects revealed a crude reality. Successfully generating correct code would require such an overwhelming amount of tweaks that using the tool would not pay off with respect to doing the entire migration manually from scratch.

To palliate this inconvenience, the idea is to get assisted by an LLM. Actually, the usage of LLMs to generate SQL code from natural language (NL) has risen over the past years, with state of the art methods rapidly improving. The reader is referred to [12] for a thorough review of the advances on LLM-based generation of SQL from NL. However, those approaches mainly focus on generating queries but do not deal with the generation of data definition language (DDL) statements [13], data manipulation language (DML) statements [14, 15], nor the translation between different implementations of the SQL standard. Moreover, available datasets like Spider [16], KaggleDBQA [17], and WikiSQL [18] only provide examples for converting NL to queries, while the schema is already provided for testing.

DDL statements are those which alter the structure of the database schema, meaning all statements starting with $CREATE$ or $DROP$. For example:

```
CREATE TABLE departments(...);
```

DML statements are those that modify the data, and in this project case they will always be $INSERT$ statements. For example:

```
INSERT INTO departments(...)
```

The DDL and DML statements just shown are only a small portion of what is available, but for the purposes of this document, the reader can just know of those.

1.2 Problem statement

In fewer words, the literature review carried out exposed that there were no AI-assisted tools that address the problem of migrating SQL between different database systems. Moreover, state-of-the-art approaches and benchmarks for NL to SQL translation do not handle DDL and DML statements, which are problematic in SQL migration. This observation opened an unexplored research question which has a relevant real-world motivation. Therefore, a new proof-of-concept tool had to be designed and implemented. Hence, the purpose of this thesis is to develop an AI-aided tool for SQL migration that supports DDL and DML, with focus on generating GoogleSQL.

1.3 Document outline

The document starts by proposing a solution in Chapter 2, explaining why a case study was created to validate the effectiveness of the solution. Since this research work was mainly experiment-driven, the document then continues by describing in Chapter 3 the quantitative and qualitative analyses of the experiments carried out. The quantitative analysis compares the effect of the tool for different state-of-the-art LLMs. The qualitative analysis contributed to engineering the prompt feedback in order to improve the quality of the generated code throughout the iterative process. Chapter 4 presents the components of the tool that was developed and explains their role in the overall research work. It also provides information about how to use it in a real-world project. Chapter 5 summarizes the contributions and discusses future research directions.

2 Proposed solution

The solution presented in this document to improve the quality of the generated SQL code is based on the introduction of a feedback loop where the LLM is provided with a more detailed context of the task by adding relevant properties extracted from the generated output (Fig. 2.1). More precisely, after an initial response, the tool assesses the quality of the generated GoogleSQL code and generates feedback based on the analysis of the errors found in the code. The feedback is then sent back to the LLM for it to generate another, hopefully better, response.

The core of the tool is a two-step evaluation process. First, the code is run through a parser that evaluates whether it is syntactically correct. If not, it returns all syntactical errors of the code. Second, all syntactically correct statements are run through a Spanner Docker image. Running the code on the engine itself allows to determine whether there are any semantic errors present in the code. Finally, both syntactic and semantic errors are bundled together with suggested fixes. This information is added to the prompt and sent back to the LLM. This process can be iterated.

Besides the tool, because of the lack of benchmarks for the migration task, an appropriate case study was necessary. It comprises a PostgreSQL code and a GoogleSQL equivalent. The case study was crafted to exhibit DDL and DML statements which typically require manual tweaks when using SMT.

It is worth to mention that this approach is similar to what other setups such as the ReAct framework [19] propose. Their approach allows the LLM to interface with and gather information from external sources such as knowledge bases or environments. In the case of this project, an external tool is being leveraged to let the chat model know how to improve the generated SQL code. The ReAct framework expects factual and grounded information from the tools used. In their paper [19] they show how giving a chat model access to a simplified Wikipedia API can improve the accuracy on question answering and fact verification tasks. This is not too different from how the proposed solution interacts with the chat

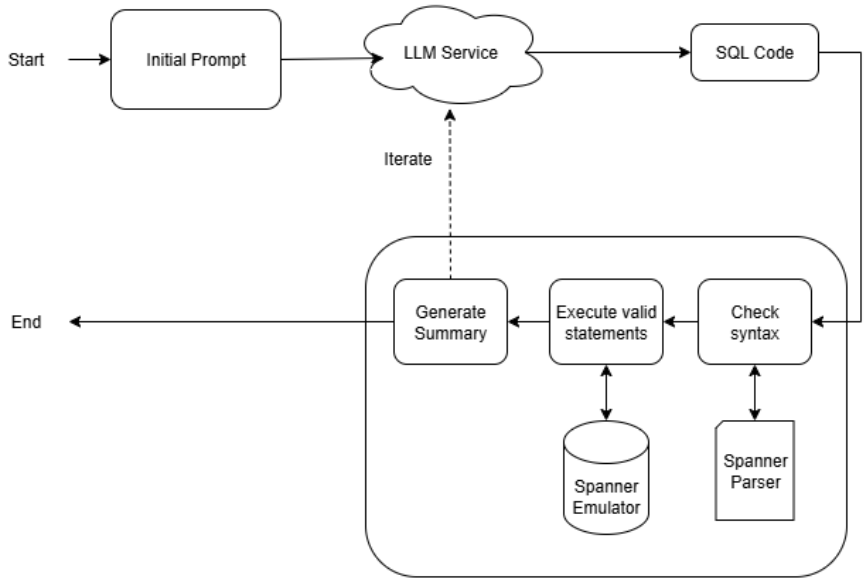


Figure 2.1: Main architecture

model. However, the proposed solution does not at any point reason that it needs to check how accurate the generated code is, it always acts, meaning that it will always check how accurate the generated code is until it has either no more errors to fix or the iteration limit has been reached.

3 Results

This chapter analyzes the quantitative and qualitative results obtained on the case study developed as part of this work to evaluate the performance of LLMs on the code migration task. The PostgreSQL and GoogleSQL code snippets are shown in Annex 7.1 and Annex 7.2, respectively. The code includes 23 statements with paradigmatic examples of data manipulation language (DML [14]) and data definition language (DDL [13]) that need to be properly translated.

The tool has been experimented with state-of-the-art language models GPT-4o and GPT-5, together with their respective mini variants. Section 3.1 discusses and compares the improvement of the language models in achieving the translation task for up to 3 iterations. Section 3.2 is devoted to a qualitative analysis of the code generated by each model. Section 3.3 describes the way the prompt is modified at each iteration by adding appropriate feedback from the tool that results from the analysis of the errors.

3.1 Quantitative Results

The initial prompt is simple: the LLM is tasked with translating from a snippet of PostgreSQL code (Annex 7.1) to GoogleSQL while preserving all semantic value. The hand-crafted GoogleSQL version (Annex 7.2) is not used for assessing the correctness of the generated code but only as a reference of the expected output. This aspect is discussed in Section 3.2. The main metrics used to evaluate the effectiveness of the tool are the parse and execution success rate:

$$ParseSuccessRate = \frac{SuccessfullyParsed}{TotalStatements}$$

$$ExecutionSuccessRate = \frac{SuccessfullyExecuted}{SuccessfullyParsed}$$

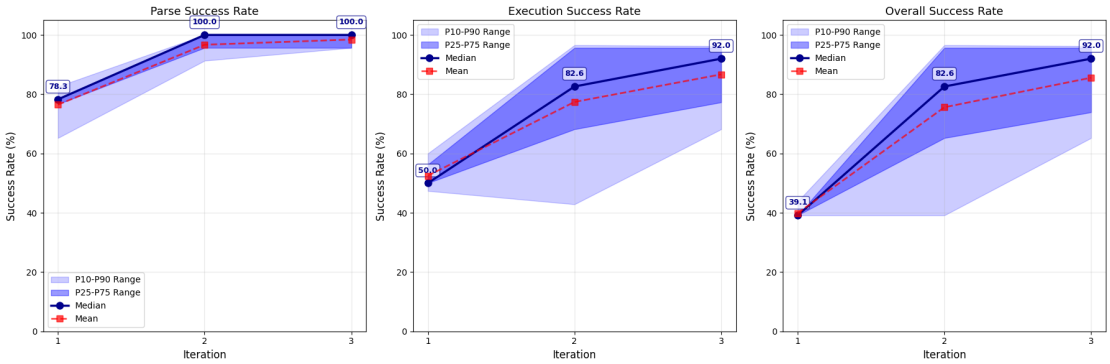


Figure 3.1: Success rate percentile ranges across iterations - GPT 4o

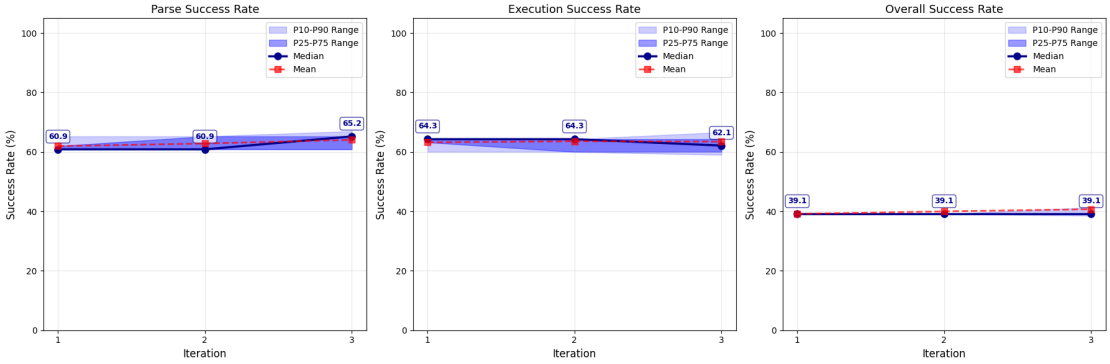


Figure 3.2: Success rate percentile ranges across iterations - GPT-4o Mini

Only statements with no parse errors get executed, so unless parse success rate is at 100 %, not every generated statement is executed and thus execution success rate is skewed based on parse success rate. Because of this, a third metric is introduced named overall success rate.

$$OverallSuccessRate = \frac{SucessfullyExecuted}{TotalStatements}$$

OpenAI's GPT 4o (Fig 3.1), 5 (Fig 3.3) and 5 mini (Fig 3.4) all improved their results after each tool iteration, with GPT 5's scoring 100 % median overall success rate. Notably GPT-4o Mini's results (Fig 3.2) were completely unaffected by the tools' feedback, leaving it as an outlier for this study and as such will not be referenced again.

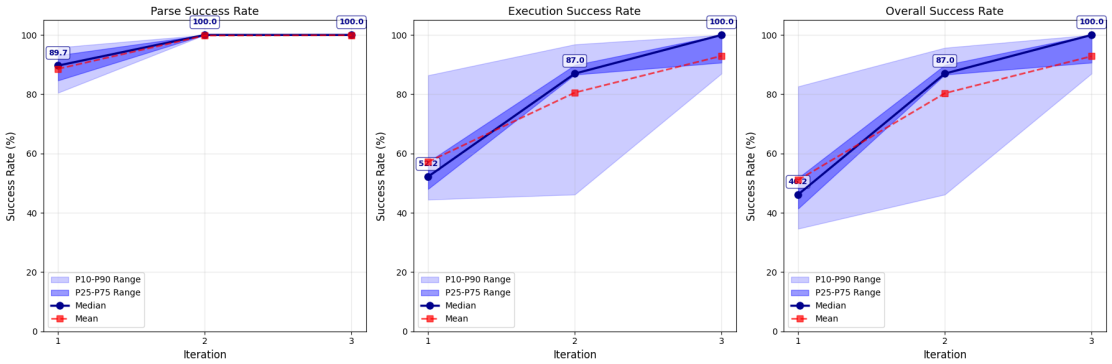


Figure 3.3: Success rate percentile ranges across iterations - GPT-5

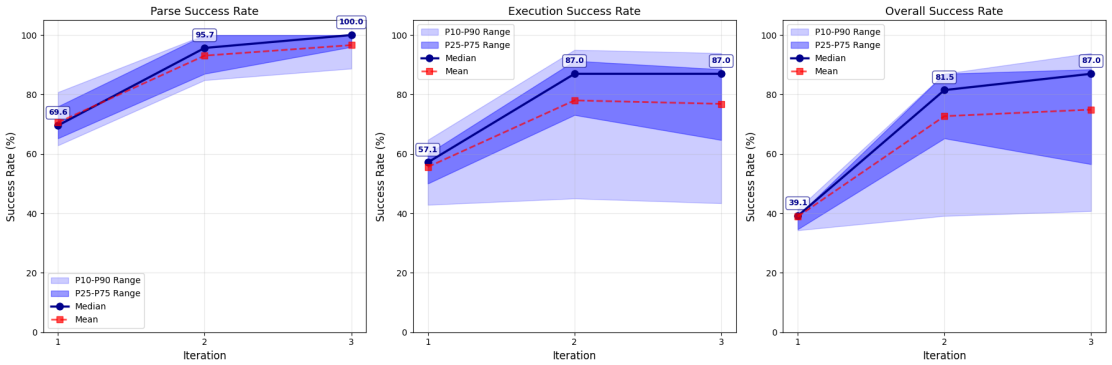


Figure 3.4: Success rate percentile ranges across iterations - GPT-5 Mini

3.2 Qualitative analysis

Although quantitative analysis gives us a good idea of how well the LLM is generating SQL code, not everything can be evaluated properly by automatic tools. For this, a manual study of the SQL outputs from our tool was conducted where a few interesting patterns emerged.

Particularly, the distribution of the number of statements (Fig 3.5) per response had more variation than expected considering that the task is always to translate the same 23-statement long PostgreSQL code (Annex 7.1).

Neither of the GPT-5 models ever generated code with fewer than the expected 23 statements, however GPT-4o would occasionally not generate all of the 4 insert statements. Here is a snippet of one of such responses:

```
-- Note: Spanner does not support RETURNING in INSERT statements.
-- Use a separate SELECT to retrieve the inserted row if needed.

-- Example insert:
-- INSERT INTO departments (dept_id, dept_name, location)
-- VALUES (GENERATE_UUID(), @dept_name, @location);

-- INSERT INTO employees (emp_id, first_name, last_name, email, hire_date, salary, dept_id)
-- VALUES (GENERATE_UUID(), @first_name, @last_name, @email, @hire_date, @salary, @dept_id);

...
```

While it is true that Spanner does not allow *RETURNING* under an insert statement, it does allow *THEN RETURN* which has the same function but renamed. This misunderstanding repeated itself with all models, however in most other cases they would limit themselves to just remove the *THEN RETURN* part of the statement and give a partially correct insert statement, instead of commenting the code out. For example, GPT-5 Mini generated this in one particular example:

```
-- Insert department (use parameters as @p1, @p2). Spanner does not support RETURNING;
-- retrieve the generated id with a follow-up query if needed.
INSERT INTO departments (dept_name, location)
VALUES (@p1, @p2);

SELECT dept_id
FROM departments
WHERE dept_name = @p1 AND location = @p2
ORDER BY dept_id DESC
LIMIT 1;
...
```

This previous example also illustrates how it can be possible for the models

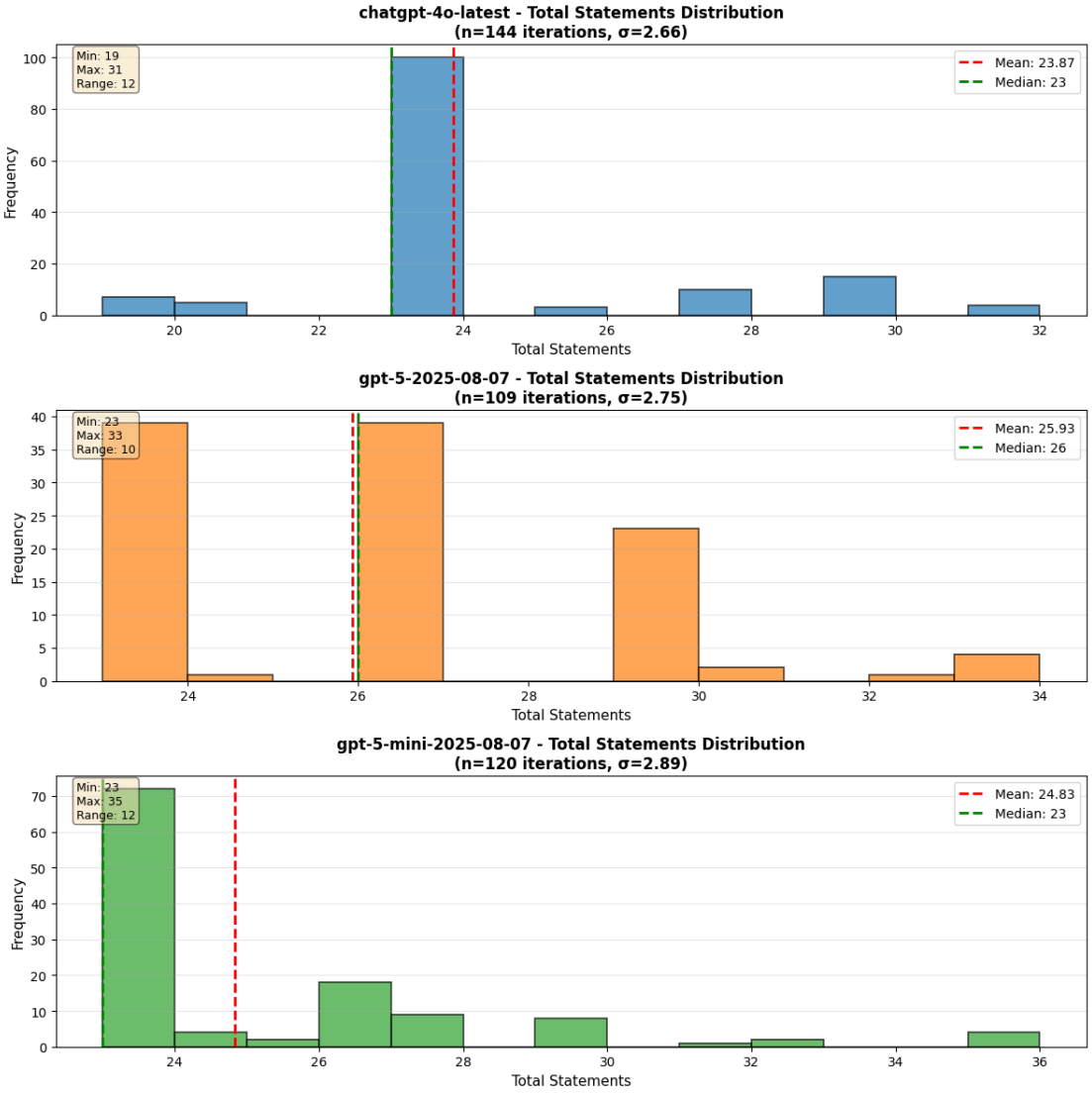


Figure 3.5: Number of statements histogram

to generate more than the expected 23 statements. The models trying to work around their self-imposed limitations generated some unexpected results. One such example was that GPT-5 exhibited a bi-modal statement distribution graph (Fig 3.5), having peaks at 23 and 26 statements.

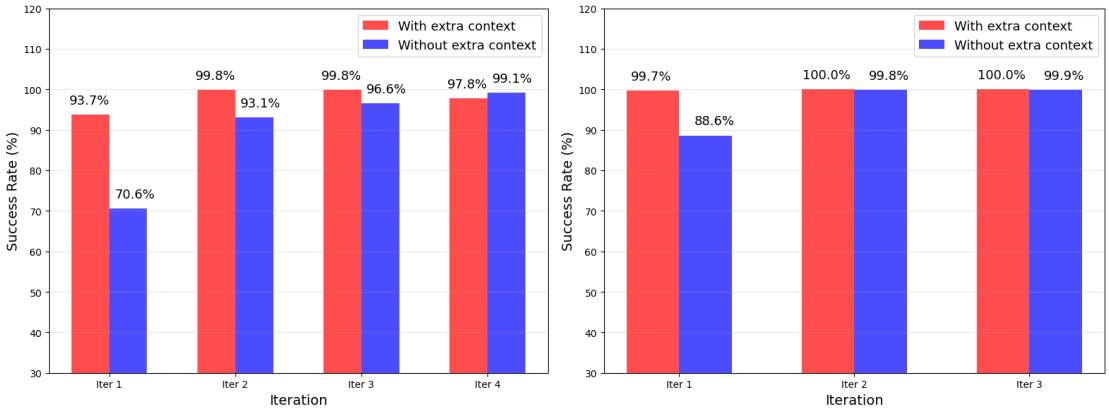
These 26-statement responses were due to Spanner handling the *SERIAL* property differently than PostgreSQL. To have an auto-incrementing numbered value in Spanner you must either set the default sequence kind in the database options or create a new sequence for each table where you want an auto-incrementing value [20]. This second option was chosen by GPT-5 in approximately 40 % of the iterations, and because three of the tables used *SERIAL* in the original PostgreSQL code would mean that three more statements would be needed to generate those sequences. The other cases where GPT-5 did not choose this option were when the primary key of the tables was changed to a string type using the Spanner defined method *GENERATE_UUID()* as the default value generator. Both of these methods are equally encouraged by the official documentation [21].

Here is a snippet of how a sequence based table needs to be created:

```
CREATE SEQUENCE departments_seq OPTIONS (sequence_kind = 'bit_reversed_positive');

CREATE SEQUENCE employees_seq OPTIONS (sequence_kind = 'bit_reversed_positive');

CREATE SEQUENCE projects_seq OPTIONS (sequence_kind = 'bit_reversed_positive');

CREATE TABLE departments (
  dept_id INT64 NOT NULL DEFAULT (GET_NEXT_SEQUENCE_VALUE(SEQUENCE departments_seq)),
  dept_name STRING(50) NOT NULL,
  location STRING(100),
  created_at TIMESTAMP DEFAULT (CURRENT_TIMESTAMP())
) PRIMARY KEY (dept_id);

CREATE TABLE employees (
  emp_id INT64 NOT NULL DEFAULT (GET_NEXT_SEQUENCE_VALUE(SEQUENCE employees_seq)),
  first_name STRING(50) NOT NULL,
  last_name STRING(50) NOT NULL,
...
```

And when using UUIDs the table creation code snippet is as follows:

```
CREATE TABLE departments (
  dept_id STRING(36) NOT NULL DEFAULT (GENERATE_UUID()),
  dept_name STRING(50) NOT NULL,
  location STRING(100),
  created_at TIMESTAMP DEFAULT (CURRENT_TIMESTAMP())
) PRIMARY KEY (dept_id);

CREATE TABLE employees (
  emp_id STRING(36) NOT NULL DEFAULT (GENERATE_UUID()),
  first_name STRING(50) NOT NULL,
...
```



(a) GPT-5 Mini (b) GPT-5

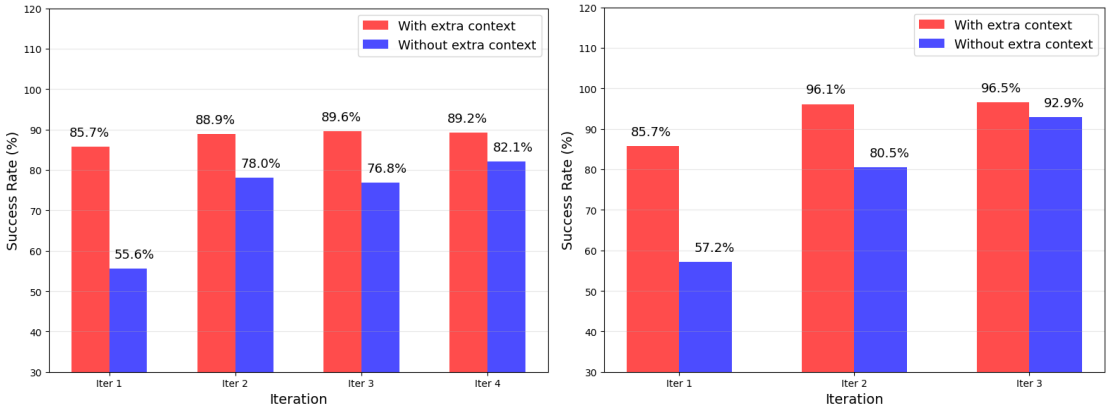
Figure 3.6: Parse success rate

3.3 Improving the initial context

The thorough experimental evaluation with the case study performed in Section 3.1, together with the qualitative analysis discussed in Section 3.2, has been paramount to engineer a more detailed prompt than the initial one used in the experiments so far.

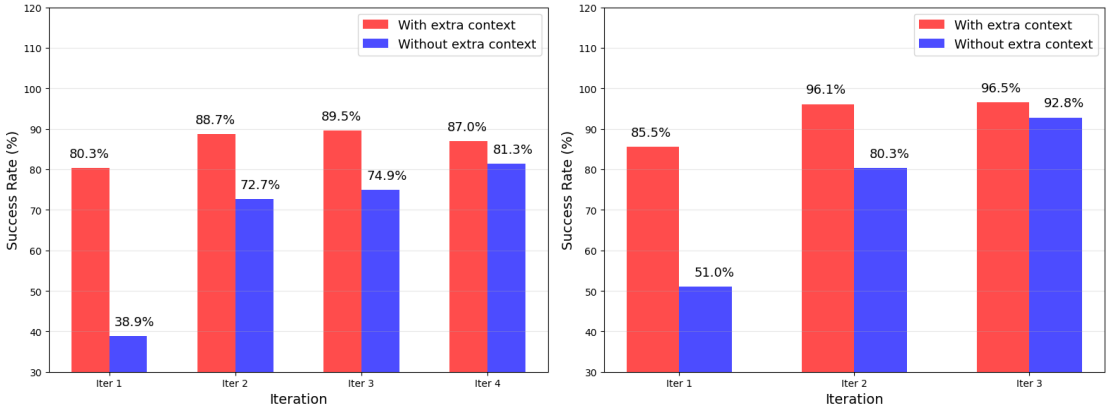
With the aim of boosting performance improvements at each iteration of our tool, we customized a new prompt containing code generation guidelines extracted from a careful analysis of the feedback responses observed during the previous experimental phase. It is important to notice that these guidelines contain only the most common errors encountered when parsing and executing generated code, including tips about how to solve them.

The experimental results obtained with this new initial prompt are shown in Figures 3.6, 3.7 and 3.8. It is evident that the performance from the first iteration onward improved drastically for GPT-5 and GPT-5 Mini in comparison with the non-enhanced prompt used for all previous runs in Section 3.1. This upswing is quite noticeable in the first iteration, where it reaches about 35 % and 40 % increase in the overall success rate, respectively (Fig. 3.8). In the case of GPT-5 Mini, it takes only one iteration with the augmented prompt to achieve almost the same overall success rate reached after 4 iterations with the non-augmented one.



(a) GPT-5 Mini (b) GPT-5

Figure 3.7: Execution success rate



(a) GPT-5 Mini (b) GPT-5

Figure 3.8: Overall success rate

4 Codebase

The codebase for this project is available in *https://github.com/sDaners/tesis*. The primary language used was Go [22] mainly because of my familiarity with it. Python [23] was also used in notebook format [24] to generate the plots [25] presented in this document.

4.1 The mostly manual tool

The first evaluator tool developed for this project was extremely basic and was only done to verify that an LLM could struggle with translating from PostgreSQL [26] to GoogleSQL [8]. This tool did not yet incorporate the parser as a prerequisite to executing statements, so every statement would be executed, and the parse errors would be returned by the Spanner engine itself. The prompts were manually sent using OpenAI's chat GUI, and each statement had to be *separated by their corresponding type* (create, insert, select or drop). It also did not generate any feedback, since its only purpose was to run the GoogleSQL code on Spanner to check for errors.

The code in *postgres_test* handled the actual execution of the tool, running it as if it were a unit test. While developing this, to verify that the tool was really working as intended, the PostgreSQL code was manually translated to Spanner's GoogleSQL. That code can be found in *repo/valid_repo/valid_spanner_sql.go*.

4.2 The semi-automatic tool

Once the proof of concept of the first tool looked promising, the next step was to avoid having to manually convert the LLM's response into something the

tool could use. The best way to achieve this was to let the tool read the SQL code directly from files, instead of having to write the SQL statements into their corresponding string slices.

For this, a new tool was written from scratch which could read every *.sql* file inside of a named folder and evaluate them. The tool would then generate a markdown file containing a summary of the parsing and execution errors with their corresponding metrics and a detailed section showing every error that occured on every file. The original code for this can be found in *tests/spanner/generated_spanner_test.go*. This was later migrated to *cmd/folder-eval/main.go* for ease of use and to remove the testing framework. The generated markdown file containing the evaluation can be found in the same folder.

4.3 The iterative tool

Using the knowledge gathered from the markdown summary, a new tool was created which would evaluate a single file and compile a list of suggestions for the LLM to hopefully improve the quality of its response. The purpose of this feedback was to enable the LLM to understand any issues their generated code might have, and even provide it with possible solutions. At this point then was when the proper feedback loop was implemented, which would prove itself quite successful. The feedback underwent a lot of fine-tuning for a while until the main bottleneck for testing became the sample size, as running the tool against only one file was not enough to derive any conclusions. The code for this tool can be found in *cmd/sql-eval/main.go*

4.4 The complete tool

At this point, the only missing piece was automating the entire flow so that a relevant sample size could be generated. The feedback pipeline was fully functional, so the only remaining step to automate was interacting with the LLM.

For this, OpenAI's conversations API [27] was used. The pipeline was also adjusted to allow multiple concurrent executions to decrease the total time taken to gather data. All executions record their results into a new *JSON* file, which would accumulate the results for every run.

Using flags, the behavior of the tool can be changed:

Iterations - Set the maximum number of prompts to send per pipeline. Setting it as 2 is the minimum required for the feedback loop to engage.

Concurrent - Set the number of concurrent pipelines to run.

Model - Set the OpenAI model to be used for all pipelines

Save output - Save the final SQL code to a new file

Save results - Save the metrics for the runs into the JSON file.

More context - Add extra context on the first prompt on how to generate better GoogleSQL code

Some of the generated SQL code can be found in the folder *generated_sql*. How the files were named changed over time, with the latest being fully self generated names using the date and time to avoid file name conflicts. In total, the PostgreSQL code was translated more than 250 times to GoogleSQL, but only a small percentage of those files were actually preserved.

5 Conclusion

A tool was presented which improves the quality of an LLM's output when translating from PostgreSQL to GoogleSQL. This tool leverages an iterative process that feeds off parsing and execution errors to give feedback to the LLM so that it can improve the generated code. The tool was shown to improve the quality of the generated code for all models tested, except GPT-4o Mini.

State of the art models such as GPT-5 were able to improve their overall success rate 10th percentile score from 34.6 % to 86.9 % in just 3 iterations, with even better results if additional initial guidelines are provided (Section 3.3). As good as that 86.9 % final score for the 10th percentile of generated code is, doing it on a state of the art model shows that there is still work to be done in this niche.

All in all, a lot of work can still be done to further increase the quality of the generated DDL and DML code. Future work should include testing more case studies, and the case studies should include more languages than PostgreSQL and GoogleSQL. Future work could also include adapting this tool so that AI agents can use it, for example, by converting it to a model context protocol (MCP) server [28].

6 References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[3] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," 2021. [Online]. Available: https://arxiv.org/abs/2101.00190

[4] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," 2022. [Online]. Available: https://arxiv.org/abs/2109.01652

[5] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023. [Online]. Available: https://arxiv.org/abs/2201.11903

[6] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, "A survey on evaluation of large language models," *ACM transactions on intelligent systems and technology*, vol. 15, no. 3, pp. 1–45, 2024.

[7] Z. Chen, C. Wang, W. Sun, G. Yang, X. Liu, J. M. Zhang, and Y. Liu, "Promptware engineering: Software engineering for llm prompt development," 2025. [Online]. Available: https://arxiv.org/abs/2503.02400

[8] Google, “The googlesql language in spanner,” [Oct 16, 2025]. [Online]. Available: https://cloud.google.com/spanner/docs/reference/standard-sql/overview

[9] ——, “Spanner relational database,” [Oct 16, 2025]. [Online]. Available: https://cloud.google.com/spanner/docs

[10] ——, “Migrate your schema,” [Oct 16, 2025]. [Online]. Available: https://cloud.google.com/spanner/docs/schema-migration

[11] ——, “Spanner migration tool,” [Oct 16, 2025]. [Online]. Available: https://googlecloudplatform.github.io/spanner-migration-tool/

[12] A. Mohammadjafari, A. S. Maida, and R. Gottumukkala, “From natural language to sql: Review of llm-based text-to-sql systems,” 2025. [Online]. Available: https://arxiv.org/abs/2410.01066

[13] Google, “Googlesql dml syntax,” [Oct 16, 2025]. [Online]. Available: https://cloud.google.com/spanner/docs/reference/standard-sql/data-definition-language

[14] ——, “Googlesql dml syntax,” [Oct 16, 2025]. [Online]. Available: https://cloud.google.com/spanner/docs/reference/standard-sql/dml-syntax

[15] W. Y. Y. Bernard Chang, “Getting started with nl2sql (natural language to sql) with gemini and bigquery,” [Nov 7, 2024]. [Online]. Available: https://cloud.google.com/blog/products/data-analytics/nl2sql-with-bigquery-and-gemini

[16] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev, “Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task,” *CoRR*, vol. abs/1809.08887, 2018. [Online]. Available: http://arxiv.org/abs/1809.08887

[17] C. Lee, O. Polozov, and M. Richardson, “Kaggledbqa: Realistic evaluation of text-to-sql parsers,” *CoRR*, vol. abs/2106.11455, 2021. [Online]. Available: https://arxiv.org/abs/2106.11455

[18] V. Zhong, C. Xiong, and R. Socher, “Seq2sql: Generating structured queries from natural language using reinforcement learning,” *CoRR*, vol. abs/1709.00103, 2017. [Online]. Available: http://arxiv.org/abs/1709.00103

[19] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," 2023. [Online]. Available: https://arxiv.org/abs/2210.03629

[20] Google, "Create and manage sequences," [Oct 16, 2025]. [Online]. Available: https://cloud.google.com/spanner/docs/sequence-tasks

[21] ——, "Primary key default values management," [Oct 16, 2025]. [Online]. Available: https://cloud.google.com/spanner/docs/primary-key-default-value

[22] ——, "The go programming language," [Oct 16, 2025]. [Online]. Available: https://go.dev

[23] P. S. Foundation, "Python programming language," [Oct 16, 2025]. [Online]. Available: https://www.python.org

[24] J. Foundation, "Jupyter notebooks," [Oct 16, 2025]. [Online]. Available: https://jupyter-notebook.readthedocs.io/en/stable/

[25] T. M. development team, "Matplotlib: Visualization with python," [Oct 16, 2025]. [Online]. Available: https://matplotlib.org/

[26] P. G. D. Group, "Postgresql," [Oct 16, 2025]. [Online]. Available: https://www.postgresql.org/

[27] OpenAI, "Convesations api reference," [Oct 16, 2025]. [Online]. Available: https://platform.openai.com/docs/api-reference/conversations

[28] P. Anthropic, "What is the model context protocol (mcp)?" [Oct 16, 2025]. [Online]. Available: https://modelcontextprotocol.io/docs/getting-started/intro

7 Annex

7.1 PostgreSQL code

```
CREATE TABLE departments (
    dept_id SERIAL PRIMARY KEY,
    dept_name VARCHAR(50) NOT NULL,
    location VARCHAR(100),
    created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE employees (
    emp_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(150),
    hire_date DATE NOT NULL,
    salary NUMERIC,
    dept_id INTEGER REFERENCES departments(dept_id),
    manager_id INTEGER REFERENCES employees(emp_id),
    phone_number VARCHAR(20)
);

CREATE UNIQUE INDEX idx_emp_email ON employees(email);

CREATE TABLE projects (
    project_id SERIAL PRIMARY KEY,
    project_name VARCHAR(100) NOT NULL,
    start_date DATE,
    end_date DATE,
    budget NUMERIC,
    status VARCHAR(20) DEFAULT 'ACTIVE',
```

```
    CONSTRAINT check_dates CHECK (end_date > start_date),
    CONSTRAINT check_status CHECK (status IN ('ACTIVE', 'COMPLETED', 'ON_HOLD', 'CA
);

CREATE TABLE project_assignments (
    emp_id INTEGER,
    project_id INTEGER,
    role VARCHAR(50),
    hours_allocated INTEGER,
    PRIMARY KEY (emp_id, project_id),
    FOREIGN KEY (emp_id) REFERENCES employees(emp_id),
    FOREIGN KEY (project_id) REFERENCES projects(project_id)
);

CREATE INDEX idx_emp_name ON employees(last_name, first_name);
CREATE INDEX idx_dept_location ON departments(location);
CREATE INDEX idx_project_status ON projects(status);

CREATE OR REPLACE VIEW employee_details AS
SELECT
    e.emp_id,
    e.first_name,
    e.last_name,
    e.email,
    d.dept_name,
    m.first_name as manager_first_name,
    m.last_name as manager_last_name
FROM employees e
LEFT JOIN departments d ON e.dept_id = d.dept_id
LEFT JOIN employees m ON e.manager_id = m.emp_id;

INSERT INTO departments (dept_name, location)
VALUES ($1, $2)
RETURNING dept_id;

INSERT INTO employees (first_name, last_name, email, hire_date, salary, dept_id)
VALUES ($1, $2, $3, $4, $5, $6)
RETURNING emp_id;

INSERT INTO projects (project_name, start_date, end_date, budget, status)
VALUES ($1, $2, $3, $4, $5)
```

```
RETURNING project_id;

INSERT INTO project_assignments (emp_id, project_id, role, hours_allocated)
VALUES ($1, $2, $3, $4);

SELECT e.emp_id, e.first_name, e.last_name, e.email, d.dept_name,
       m.first_name as manager_first_name, m.last_name as manager_last_name,
       p.project_name
FROM employees e
LEFT JOIN departments d ON e.dept_id = d.dept_id
LEFT JOIN employees m ON e.manager_id = m.emp_id
LEFT JOIN project_assignments pa ON e.emp_id = pa.emp_id
LEFT JOIN projects p ON pa.project_id = p.project_id;

DROP VIEW  employee_details;
DROP INDEX  idx_project_status;
DROP INDEX  idx_dept_location;
DROP INDEX  idx_emp_name;
DROP INDEX  idx_emp_email;
DROP TABLE  project_assignments;
DROP TABLE  projects;
DROP TABLE  employees;
DROP TABLE  departments;
```

7.2 Valid GoogleSQL code

```
CREATE TABLE departments (
    dept_id STRING(36) DEFAULT (GENERATE_UUID()),
    dept_name STRING(50) NOT NULL,
    location STRING(100),
    created_at TIMESTAMP DEFAULT (CURRENT_TIMESTAMP())
) PRIMARY KEY (dept_id);

CREATE TABLE employees (
    emp_id STRING(36) DEFAULT (GENERATE_UUID()),
    first_name STRING(50) NOT NULL,
    last_name STRING(50) NOT NULL,
    email STRING(150),
    hire_date TIMESTAMP NOT NULL,
    salary FLOAT64,
    dept_id STRING(36),
    manager_id STRING(36),
    phone_number STRING(20),
    CONSTRAINT fk_dept FOREIGN KEY (dept_id) REFERENCES departments(dept_id),
    CONSTRAINT fk_manager FOREIGN KEY (manager_id) REFERENCES employees(emp_id)
) PRIMARY KEY (emp_id);

CREATE TABLE projects (
    project_id STRING(36) DEFAULT (GENERATE_UUID()),
    project_name STRING(100) NOT NULL,
    start_date TIMESTAMP,
    end_date TIMESTAMP,
    budget FLOAT64,
    status STRING(20) DEFAULT ('ACTIVE'),
    CONSTRAINT check_dates CHECK (end_date > start_date),
    CONSTRAINT check_status CHECK (status IN ('ACTIVE', 'COMPLETED', 'ON_HOLD', 'CA
) PRIMARY KEY (project_id);

CREATE TABLE project_assignments (
    emp_id STRING(36) NOT NULL,
    project_id STRING(36) NOT NULL,
    role STRING(50),
    hours_allocated INT64,
    CONSTRAINT fk_emp FOREIGN KEY (emp_id) REFERENCES employees(emp_id),
    CONSTRAINT fk_project FOREIGN KEY (project_id) REFERENCES projects(project_id)
```

```
) PRIMARY KEY (emp_id, project_id);

CREATE UNIQUE INDEX idx_emp_email ON employees(email);
CREATE INDEX idx_emp_name ON employees(last_name, first_name);
CREATE INDEX idx_dept_location ON departments(location);
CREATE INDEX idx_project_status ON projects(status);

CREATE OR REPLACE VIEW employee_details
    SQL SECURITY INVOKER
    AS SELECT
        e.emp_id,
        e.first_name,
        e.last_name,
        e.email,
        d.dept_name,
        m.first_name as manager_first_name,
        m.last_name as manager_last_name
    FROM employees e
    LEFT JOIN departments d ON e.dept_id = d.dept_id
    LEFT JOIN employees m ON e.manager_id = m.emp_id;

INSERT INTO departments (dept_name, location)
VALUES (@dept_name, @location)
THEN RETURN dept_id;

INSERT INTO employees (first_name, last_name, email, hire_date, salary, dept_id)
VALUES (@first_name, @last_name, @email, @hire_date, @salary, @dept_id)
THEN RETURN emp_id;

INSERT INTO projects (project_name, start_date, end_date, budget, status)
VALUES (@project_name, @start_date, @end_date, @budget, @status)
THEN RETURN project_id;

INSERT INTO project_assignments (emp_id, project_id, role, hours_allocated)
VALUES (@emp_id, @project_id, @role, @hours);

SELECT e.emp_id, e.first_name, e.last_name, e.email, d.dept_name,
       m.first_name as manager_first_name, m.last_name as manager_last_name,
       p.project_name
FROM employees e
LEFT JOIN departments d ON e.dept_id = d.dept_id
```

```
LEFT JOIN employees m ON e.manager_id = m.emp_id
LEFT JOIN project_assignments pa ON e.emp_id = pa.emp_id
LEFT JOIN projects p ON pa.project_id = p.project_id;

DROP VIEW employee_details;
DROP INDEX  idx_project_status;
DROP INDEX  idx_dept_location;
DROP INDEX  idx_emp_name;
DROP INDEX  idx_emp_email;
DROP TABLE  project_assignments;
DROP TABLE  projects;
DROP TABLE  employees;
DROP TABLE  departments;
```