Leveraging Large Language Models for Text-to-SQL on the IUPHAR/BPS Guide to Pharmacology Database

Ian Little

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2025

Abstract

In recent years, the advancement of Large Language Models (LLMs) has established a new and improved approach for converting natural language (NL) questions into Structured Query Language (SQL) statements (text-to-SQL). In this study, I researched the task of text-to-SQL for the IUPHAR/BPS Guide to Pharmacology database (GtoPdb) [1]. The motivation for my research was to improve the GtoPdb as a teaching resource, allowing staff and students to query the database through NL requests, making the GtoPdb more accessible for all users (students and researchers).

To conduct my research, I was given a small dataset of NL-SQL pairs suitable for use on the GtoPdb. I investigated several in-context learning methods found in relevant text-to-SQL literature. Through the combination of the best zero-shot methods explored on the training set I constructed an effective zero-shot text-to-SQL pipeline. Using this pipeline, I integrated several few-shot learning strategies to help pick NL-SQL examples, from the training set, when evaluating the held-out set.

In addition, I designed an evaluation metric, Partial Execution Accuracy (PEX), to better assess the accuracy of generated SQL queries. I discovered that choosing random few-shot examples, from the training set, had a significant impact on the accuracy of generated SQL queries (increasing from 3.33% to 16.67%). Moreover, I was able to show that choosing relevant few-shot examples, based on their similarity to a given NL question, resulted in a further increase in accuracy (reaching 30%).

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ian Little*)

Acknowledgements

This work would not have been possible without my fantastic supervisors *David Sterratt* and *Simon Harding*, who have both supported me throughout my research. I would also like to thank the NC-IUPHAR Database Executive Committee for curating a dataset for use on this project.

Over the last 4 years, my family and friends have been so incredibly supportive of me; I cannot thank them enough. In particular, this research is dedicated to my two sets of loving grandparents, *Margaret & Ian Dale* and *Christine & Bob Little*.

Table of Contents

1 Introduction **1**
- 1.1 Research Questions . . . 2
- 1.2 Project Outline . . . 2

2 Background **3**
- 2.1 Language Modelling . . . 3
- 2.2 Large Language Models (LLMs) . . . 4
  - 2.2.1 The Transformer . . . 4
  - 2.2.2 Training Large Language Models . . . 4
- 2.3 OpenAI's Large Language Models . . . 5
- 2.4 In-context Learning . . . 5
  - 2.4.1 OpenAI Prompt . . . 6
  - 2.4.2 Prompt Engineering . . . 6
  - 2.4.3 Few-shot Learning . . . 6
- 2.5 IUPHAR/BPS Guide to Pharmacology Database . . . 7
- 2.6 Text-to-SQL . . . 7

3 Methodology **9**
- 3.1 Dataset . . . 9
- 3.2 Evaluation Metrics . . . 10
  - 3.2.1 Robustness Metrics . . . 10
  - 3.2.2 Accuracy Metrics . . . 11
- 3.3 Selected Large Language Models . . . 13
- 3.4 Zero-Shot Methods . . . 13
  - 3.4.1 System Messages . . . 13
  - 3.4.2 Schema-linking . . . 15
  - 3.4.3 Self-correction . . . 16
  - 3.4.4 Self-validation . . . 18
- 3.5 Few-shot Learning Methods . . . 19
  - 3.5.1 Criteria For Picking Few-shot Examples . . . 19
  - 3.5.2 Ranking Similarity . . . 20

4 Text-to-SQL Pipeline **21**
- 4.1 System Message Exploration . . . 21
  - 4.1.1 Comparative Analysis . . . 21
  - 4.1.2 Refining LLM Selection . . . 23

4.2 Cost Efficiency and Token Optimisation . . . 23
4.2.1 Single-Column Schema-linking . . . 24
4.2.2 Multi-table Schema-linking . . . 24
4.2.3 Table-to-column Schema-linking . . . 25
4.2.4 Comparative Analysis . . . 25
4.2.5 Decrease in Token Count . . . 26
4.2.6 Time to Respond . . . 27
4.3 Improving Robustness . . . 28
4.3.1 Syntax Correction . . . 28
4.3.2 Empty Output Correction . . . 29
4.4 Self-validation . . . 30
4.5 Final Pipeline . . . 31

5 Results **32**
5.1 Establishing Baseline Performances . . . 33
5.1.1 Zero-shot . . . 33
5.1.2 Few-shot . . . 33
5.2 Choosing Few-shot Examples . . . 34
5.2.1 Similar SQL Examples . . . 34
5.2.2 Similar NL Question Examples . . . 35
5.3 Summary . . . 36

6 Conclusions **38**
6.1 Discussion . . . 38
6.1.1 Contributions . . . 38
6.1.2 Related Work Comparison . . . 39
6.2 Limitations and Future Work . . . 40

Bibliography **41**

A Example SQL for Understanding Evaluation Metrics **47**
A.1 Execution Accuracy (EX) . . . 47
A.2 Exact Matching (EM) . . . 47
A.3 Partial Execution Accuracy (PEX) . . . 48

B SQL Used to Output System Messages **49**
B.1 SQL For Text Representation . . . 49
B.2 SQL For Basic Representation . . . 49

C Schema-Linking Prompts/Visualisations **50**
C.1 Schema Linking Visual For Each System Message . . . 50
C.2 Single-column Schema-Linking Prompt . . . 53
C.3 Multi-table Schema-Linking Prompt . . . 53
C.4 Table-to-column Schema-Linking Prompt . . . 54
C.5 System Messages in JSON format . . . 55
C.6 Token Count Reduction . . . 57

D Self-Validation Workflow **58**

Chapter 1

Introduction

Large Language Models (LLMs), such as OpenAI's ChatGPT, have recently emerged as a groundbreaking technology within the field of computer science. They have demonstrated unprecedented capabilities in responding to natural language (NL) requests, through prompts, by generating logically coherent outputs. In light of this, LLMs have been identified as a solution for the specialised task of translating NL questions into Structured Query Language (SQL) [2, 3, 4] statements (text-to-SQL) [5]. Notably, LLM-based methods achieve pinnacle performance on benchmark datasets [6, 7], surpassing all prior non-LLM approaches .

Within this research, I investigated the performance of text-to-SQL translation on the IUPHAR/BPS Guide to Pharmacology database (GtoPdb) [1], a resource which stores succinct summaries, key references, and experimentally recommended ligands for pharmacological targets. The goal of my research was to enhance the GtoPdb's usability as a teaching tool to allow users (particularly educators and students) to query the database using NL questions, bypassing the technical barrier of SQL expertise.

I used several LLMs from OpenAI, allowing for a solid exploration of several published text-to-SQL in-context learning methodologies. To evaluate my methods, I used a small dataset of NL-SQL pairs commonly queried on the GtoPdb, which was split into a training and held-out set. I evaluated zero-shot methods on the training set, whereby the LLM produced SQL queries when given no NL-SQL example pairs. I also refined my exploration, focusing on a subset of these methods and LLMs based on the following key factors:

- **Cost** – The financial feasibility of the method.
- **Response Time** – The speed at which SQL response is generated.
- **Performance** – The overall quality of the generated SQL queries.
- **Usability** – The supported features needed for certain methods.

Having discovered the best methods for generating SQL queries, I constructed a zero-shot text-to-SQL pipeline. Finally, I tested my text-to-SQL pipeline on the held-out set by providing the LLM with a minimal number of NL-SQL example pairs, chosen from

the training set, a technique known as few-shot learning [8].

1.1 Research Questions

The research questions regarding text-to-SQL for the GtoPdb that my research will answer are as follows:

- Is it possible to design an evaluation metric that effectively measures the accuracy of the text-to-SQL approaches?
- Which database schema representation is the best?
- Does the proposed self-validation have any effect on producing SQL queries?
- Which LLM is the most suitable?
- How important is few-shot learning compared to zero-shot learning?
- What is the most effective way of choosing examples for few-shot learning?

1.2 Project Outline

Background – This begins with a brief introduction to LLMs, followed by a focus on specific OpenAI LLMs. Next, an explanation of the GtoPdb [1] and finally, a discussion of benchmark datasets for the task of text-to-SQL.

Methodology – Here, I present a detailed description of the dataset used for the project and formally define my evaluation metrics. I then outline and define the experiments conducted within my implementation.

Text-to-SQL Pipeline – This chapter begins my implementation by investigating several of the zero-shot (no examples) prompting methods on the given training set. By evaluating these methods, I construct a text-to-SQL pipeline capable of generating SQL queries under a zero-shot scenario.

Results – To assess the impact of few-shot learning, I use my text-to-SQL pipeline under several different few-shot scenarios on the held-out set. By choosing examples from the training set, based on different criteria, I discover the most effective few-shot approach for the task of text-to-SQL on the GtoPdb.

Conclusions – I summarise my findings, provide a comparison with related work, and detail the main limitations and subsequent future work related to my research.

Chapter 2

Background

The following chapter introduces basic concepts and recent research relevant to generating SQL queries from NL (text-to-SQL) using LLMs. Since the world of LLMs is an extensive and rapidly developing one, this chapter is limited to brief overviews. For more (in-depth) information on language modelling please refer to a more rigorous natural language processing (NLP) course such as Elena Voita's NLP Course For You [9].

2.1 Language Modelling

The task of predicting the next sequence of words given the previous can be defined as language modelling. In other words, given a sequence of tokens (e.g words) $w_1, ..., w_n$ and a given index t a language model (LM) predicts a probability distribution $P(w_t|w_{<t})$ that can be used to determine the next words in the sequence [9]. An early form of statistical language modelling, namely n-gram language models, first introduced in the late 1990s, calculated straightforward distributions by considering the frequency of sequential tokens [10]. Formally,

$$P(w_t|w_1, ..., w_{t-1}) = \frac{C(w_1, ..., w_{t-1}, w_t)}{C(w_1, ..., w_{t-1})}, \tag{2.1}$$

where $C(w_1, ..., w_i)$ is the frequency (count) that the sequence of tokens $w_1, ..., w_i$ appears in the training text. The n-gram language model was built upon the Markov Property which is an independence assumption stating that the probability of a word only depends on a defined number of previous words. Therefore, assuming that a sequence of tokens $w_1, ..., w_t$ corresponds to $P(w_t|w_1, ..., w_{t-1}) = P(w_t|w_{t-n+1}, ..., w_{t-1})$. In example, a trigram model $(n = 3)$ corresponds to $P(y_t|y_1, ..., y_{t-1}) = P(y_t|y_{t-2}, y_{t-1})$ which says that the probability of the next token y_t only depends on the previous two tokens y_{t-2} and y_{t-1}.

2.2 Large Language Models (LLMs)

Large Language Models are extremely advanced LMs built using billions of parameters, hence termed 'large', allowing them to exhibit an unprecedented ability when responding to NL requests [11]. Today, the most popular architecture for LLMs was introduced by researchers at Google and was named the transformer [12].

2.2.1 The Transformer

The transformer is a type of neural network that relies on self-attention [13] to capture relationships between tokens in a sentence. In its original publication [12], the transformer consisted of an encoder and a decoder, each with 6 layers, totalling 12 layers. This self-attention mechanism lets every token consider all other tokens in the sequence, dynamically adjusting its high-dimensional representation based on context. Unlike earlier models that used attention mechanisms alongside sequential processing in recurrent networks, the transformer's parallelisable structure boasts faster training and improved translation quality. By continuously refining token representations, self-attention helps to properly encode the appropriate context of semantically ambiguous tokens.

2.2.2 Training Large Language Models

Modern-day in-context learning LLMs such as those used within this research, Generative Pre-trained Transformer (GPT) models, can generate a response based on a user-given prompt in an autoregressive manner; that is, the next token is predicted based on all preceding tokens. Formally, given a context sequence X with tokens $x_1, x_2, \ldots, x_{t-1}$ (the prompt) and current index t the LLM aims to predict the next token y. Using the chain rule, the conditional probability can be expressed as,

$$P(y \mid X) = \prod_{t=1}^{T} P(y_t \mid x_1, x_2, \ldots, x_{t-1}), \tag{2.2}$$

where T is the length of the sequence [11]. LLMs are trained by maximising these probabilities and tend to go through the following phases [14]:

Pre-training: The first phase requires massive amounts of data that the model is subjected to for training (i.e. pre-training dataset), giving the model the ability to now predict the next word in a sequence of tokens. However, the model will not yet be aligned with human intentions and will likely hallucinate (i.e. the generation of content that is irrelevant, made-up, or inconsistent with the input data).

Instruction Fine-tuning: To become a helpful interaction tool, the model must be fine-tuned to ensure engaging responses. A smaller dataset than the pre-training one is used to do this, namely an instruction dataset. This new dataset is full of extremely high-quality instruction and response pairs, guiding the model to become a helpful human aid.

Reinforcement Learning from Human Feedback: This stage is one that several modern-day LLMs go through (such as chatGPT). Users are tasked with evaluating

LLM-generated responses by rating them, picking the best response from multiple generated responses, and leaving comments that evaluate the response. Studies such as [15] have found this step to significantly improve the performance of larger models.

2.3 OpenAI's Large Language Models

To generate SQL queries from NL questions, this project will survey a wide selection of LLMs from OpenAI. In 2018, OpenAI released **gpt-1**, their first GPT model [16], and has since improved upon it with the release of several other LLMs.

Within the last year, OpenAI released a selection of reasoning models, namely, the o-series. These reasoning models were trained with a technique called Chain of Thought (CoT) prompting [17], letting the LLM perform step-by-step reasoning in a similar fashion to a human, thus excelling at complex, multi-step tasks. By March 2025, OpenAI had released three iterations of their reasoning models (**o3-mini, o1 and o1-mini**). The **o1** model is the most expensive and is built for tasks that require both a broad general knowledge and a lot of reasoning effort. On the other hand, the mini models are lower-latency models intended for cost-effective tasks that still require some reasoning effort. The reasoning models have outperformed other LLMs on benchmark STEM problems (coding exercises, biology exams, etc.) [18, 19] thus presenting as ideal candidates for the task of text-to-SQL for the GtoPdb.

Prior to these reasoning models, OpenAI released several other LLMs with their most recent release being **gpt-4o** in May 2024 [20]. Not only was **gpt-4o** better and faster than predecessor **gpt-4** [21] but it was also significantly cheaper[1]. Unlike the reasoning models, other LLMs are not trained to automatically perform Chain of Thought (CoT) reasoning; thus, they are intended for tasks that require quick responses without the need for automatic built-in reasoning. This research will investigate both reasoning and non-reasoning models to compare their performance when generating SQL on the GtoPdb.

2.4 In-context Learning

Since the OpenAI LLMs are trained on billions of parameters, they are not fine-tuned to perform domain-specific tasks (such as text-to-SQL). These LLMs are thus considered to have very broad applications, capable of completing a vast number of assignments. To ensure these LLMs can perform domain-specific tasks, a technique called in-context learning [22] is used. In-context learning refers to the information included within the prompt given to the LLM. Such information may include direct context or helpful examples (known as few-shot learning [8]), which builds upon the knowledge acquired during the LLM's pre-training phase, endowing it with the ability to generate a contextually relevant response.

[1] OpenAI's pricing

2.4.1 OpenAI Prompt

To perform in-context learning with the OpenAI API, the prompt is instantiated using both the system and user messages:

System Message: Describes the model's high-level behaviour, guiding the model to respond in a specific manner. For example, if the LLM is instantiated to be a joke-telling chatbot, then the LLM would reply in a Jester-like manner.

User Message: The specific request that prompts the LLM's direct response, which follows the guidelines established in the system message. For example, if a user requests a cooking recipe, the LLM generates a structured response containing ingredients, equipment, preparation steps, and other relevant details.

The culmination of these messages provides the in-context state for the LLM which dictates its response. Notably, when the system message is initialised it creates a 'conversation' state for the LLM. This means the LLM is able to remember all previous questions and responses within said conversation without the need to redefine the system message.

2.4.2 Prompt Engineering

Since these prompts contain task-specific information, it is important that their design can successfully convey the task to the LLM, a process known as prompt engineering. The design of the prompts given to the LLM has been shown to have a major influence on both the quality and structure of the LLM's generated response [23].

In the context of this research, the design of the prompt is essential to produce coherent SQL queries [24, 25, 26, 27]. To give the model enough context, the prompt is stipulated so that the system message includes the corresponding database schema information and a clear objective for the LLM to accomplish (i.e. 'Convert the following natural language question into an SQL query...'). In addition, the user message is usually populated with the NL question that the LLM needs to generate an SQL query for. The user message can also include NL-SQL example pairs intended to aid the LLM's response (few-shot learning) or it could include previously generated SQL queries that produced an error (self-correction), giving the LLM a chance to fix its previous generation. Therefore, this research will investigate different ways to display the database schema and choose NL-SQL example pairs for the LLM to use when generating SQL queries.

2.4.3 Few-shot Learning

Few-shot learning refers to the scenario in which the LLM is given a small (few) number of domain-specific examples, within its in-context information, to help it when generating its response. Essentially, the LLM is briefly 'tuned' with relevant examples on how it should reply for that specific invocation.

Few-shot learning represents a significant power of LLMs, letting these models adapt to domain-specific tasks without explicit retraining. Unlike traditional machine learning approaches that require extensive labelled datasets, few-shot learning leverages the

LLM's pre-trained knowledge and pattern recognition abilities to quickly learn from a minimal number of examples [8, 28].

2.5 IUPHAR/BPS Guide to Pharmacology Database

For this research, the task was to generate SQL queries, from NL questions, for the IUPHAR/BPS Guide to Pharmacology Database (GtoPdb) [1], which is a free-to-use online database founded by the International Union of Basic and Clinical Pharmacology (IUPHAR) and the British Pharmacological Society (BPS). Although the portal for the GtoPdb has been online since 2011, it was known, at the time, as IUPHAR-DB [29, 30]. It wasn't referred to as the GtoPdb until 2014 [31] when the BPS 'Guide to Receptors and Channels' [32] resource was integrated with the existing version of the IUPHAR-DB [33].

Moreover, the GtoPdb is a searchable database containing data on drug targets and the prescription medicines and experimental drugs that act on them. Overall, more than 1,000 scientists have contributed to the GtoPdb; currently, the database boasts 325 active contributors. New database releases happen regularly (quarterly) allowing for the information contained within the database to be extremely up to date. The database is maintained by experts and stores ligand-activity-target relationships found within primary relevant literature. The curators of the GtoPdb add new protein targets only when they meet the following criteria [1].

1. There exists strong evidence that it directly interacts with biological targets while altering their activity in a meaningful way.
2. Shows these effects at concentrations or doses that could realistically be used in medical treatments.
3. There exists additional data from studies performed in living organisms supporting the idea that the drug could have clinical use.

These criteria, combined with expert review and cross-referencing against primary literature, make the GtoPdb a trusted resource for academic research and clinical application. In addition, the database's design also supports integration with popular external resources such as ChEMBL [34, 35] and PubChem [36], further expanding its use for pharmacological research. For open-source accessibility, the GtoPdb is readily available through its online web portal[2].

2.6 Text-to-SQL

As mentioned previously, text-to-SQL is the process of converting a NL question into a corresponding SQL query [37, 5] that can be executed on a given database. Within industry, and beyond, databases are used everywhere; therefore, the ability to easily access them is imperative. However, accessing relational databases requires direct knowledge of writing and executing SQL queries. Thus, there is motivation to provide a

[2] www.guidetopharmacology.org

suitable method of converting NL requests into executable SQL queries. In recent years, the vast improvement of LLMs has enabled a new and improved way of approaching this task. Not only are LLMs able to generate SQL queries - but they have also outperformed all previous methods (such as [38]) on benchmark datasets Spider 1.0 [6] and BIRD [7].

The Spider 1.0 dataset [6] was developed by students at Yale University and incorporates SQL queries of varying difficulty (easy, medium, hard, and extra hard). It includes 10,181 questions and 5,693 unique complex SQL queries on 200 databases with multiple tables covering 138 different domains [6]. The leaderboard on their website[3] exhibits a total of 83 submissions showcasing their accuracy on the test set. As of April 2025, the best submission (MiniSeek) achieved an accuracy of 91% but has yet to be published. The next five best submissions all used **gpt-4** combined with other method(s) (published examples include [27, 24]) achieving accuracies in the range $[83.9\%, 86.2\%]$. Spider stopped updating their leaderboard in early May 2024 when they released their test set and later released Spider 2.0. Unlike the original dataset, Spider 2.0 [39] focuses only on extreme SQL queries.

Alternatively, the BIRD dataset [7] contains 12,751 unique pairs of NL questions and their corresponding SQL queries for 95 databases while covering more than 37 domains [7]. BIRD is intended to mimic real-life databases used by industry (i.e. databases with lots of noise such as unusual naming conventions and irregular formatting) and so poses a tougher challenge than Spider 1.0 with the top spot only achieving an accuracy of 77.14% on the test set[4] (at the time of writing). The top 25 submissions were all added in either 2024 or 2025, with the majority leveraging OpenAI LLMs (such as [40]), highlighting how recent the development in text-to-SQL research has been.

Both these benchmark datasets include various databases, so high-performing approaches must generalise well to different database schemas and queries. Consequently, OpenAI LLM-driven methods are notable candidates and will be explored throughout this research.

[3] Spider 1.0 leaderboard
[4] Bird leaderboard

Chapter 3

Methodology

Within this chapter, I describe the dataset of GtoPdb NL-SQL pairs, the evaluation metrics I used, and the LLMs investigated throughout my research. Additionally, I define the methodology used to build a text-to-SQL pipeline tailored for the GtoPdb. The methods investigated in this research were split into two categories: zero-shot and few-shot.

In the zero-shot setting, the LLM generated SQL queries without any NL-SQL examples, relying only on database information to produce SQL. Since these methods did not require any examples, they were evaluated on the training set to establish a baseline text-to-SQL pipeline. Here, I describe each zero-shot method under investigation.

Although the zero-shot scenario was able to yield coherent SQL, related research has demonstrated that few-shot learning is crucial for improving the quality of generated SQL [27, 26, 24]. As described in Chapter 2, few-shot learning involves providing the LLM with a limited number of domain-specific examples (for this research – NL-SQL pairs) to guide its output [28]. Here, I outline the strategies used to select NL-SQL examples from the training set when answering the NL questions in the held-out set.

3.1 Dataset

To conduct my research, I was given a dataset of NL questions and their corresponding SQL queries (NL-SQL pairs) for the GtoPdb. The dataset, curated by the NC-IUPHAR Database Executive Committee, consisted of 81 entries. Of these, 51 entries made up the training set and the remaining 30 formed the held-out set, corresponding to a training-to-held-out split of approximately 63% to 37%. In addition to the NL-SQL pairs, each entry provided supplementary information about related aspects of the data. Each entry contained:

Natural Language Query: A question, phrased in natural language (NL), that can be asked of the GtoPdb.

Difficulty - Tagged either easy, easy-moderate, moderate-hard, or hard in relation to the difficulty of the SQL query that the LLM would need to output. The tag was determined

by the NC-IUPHAR Database Executive Committee's own opinion of the SQL query rather than a structured SQL analysis (as seen in benchmark datasets [6, 7]).

Greek - Denotes whether the NL question contains a Greek letter (e.g. β).

Vague/No definite right answer - A flag that marks the NL question to be ambiguous, meaning the NC-IUPHAR Database Executive Committee deemed there to be multiple valid interpretations or SQL answers rather than a single definitive solution.

Minimum output columns - States the minimum number of columns that must be retrieved by an SQL query to answer the NL question.

Notes for student - Some pointers from the NC-IUPHAR Database Executive Committee to help understand aspects of the database schema and the provided gold standard SQL query.

SQL - The primary gold standard SQL query that answers the NL question.

2nd SQL - An alternative gold standard SQL query that would also reasonably answer the NL query.

Overall, the dataset included 8 easy, 30 easy-moderate, 35 moderate-hard, and 8 hard questions. Among these, 10 entries were flagged as vague, 4 contained Greek characters, and 22 included an alternative gold standard SQL query.

3.2 Evaluation Metrics

To assess the quality of the generated SQL queries in my experiments, I employed two sets of evaluation metrics. The first set focused on the robustness of the generated SQL queries, evaluating the most basic correctness without comparing the generated SQL queries to a gold standard SQL. The second set addressed the accuracy of the generated queries by directly comparing them against one or more gold standard SQL queries from the dataset. Throughout the research, I relied on these metrics to measure and compare performance.

3.2.1 Robustness Metrics

3.2.1.1 Successful Execution Rate (SER)

The Successful Execution Rate (SER) measures whether a predicted SQL query S can be successfully executed on the GtoPdb (check whether it is syntactically correct). Formally, SER is defined as,

$$\text{score}(S) = \begin{cases} 1, & \text{if } S \text{ can be executed without error,} \\ 0, & \text{otherwise.} \end{cases} \tag{3.1}$$

The overall SER is then computed as,

$$\text{SER} = \frac{\sum_{n=1}^{N} \text{score}(S_n)}{N}, \tag{3.2}$$

where N is the total number of generated queries. This metric helped track how many generated queries were syntactically correct.

3.2.1.2 Non-empty Execution Rate (NER)

Similar to SER, the Non-empty Execution Rate (NER) extends the concept of successful execution by also checking whether the query output (X) is non-empty. Formally,

$$\text{score}(X) = \begin{cases} 1, & \text{if } X \neq \emptyset, \\ 0, & \text{otherwise.} \end{cases} \tag{3.3}$$

Here, $X \neq \emptyset$ signifies that the query S both executes successfully and produces an output. Conversely, $X = \emptyset$ would indicate a syntactically valid SQL query but returned no rows of data. The overall NER is calculated as,

$$\text{NER} = \frac{\sum_{n=1}^{N} \text{score}(X_n)}{N}, \tag{3.4}$$

where N is the number of generated queries. Consequently, this metric captured not only syntactically correct queries but also those that yield non-empty results.

3.2.2 Accuracy Metrics

To evaluate the accuracy of the generated SQL queries, each generated query was compared against the gold standard SQL queries from the dataset. In cases where two different gold standard SQL queries were available, I tested the generated query against both. If it satisfied at least one of the gold standard queries for a given metric, it was counted as correct for that metric.

3.2.2.1 Execution Accuracy (EX)

The first accuracy metric, Execution Accuracy (EX) used in benchmark datasets [7, 6], measures whether the output of a generated SQL query X matches the output of the corresponding gold standard query $\tilde{X}$. Formally,

$$\text{score}(X, \tilde{X}) = \begin{cases} 1, & \text{if } X = \tilde{X}, \\ 0, & \text{otherwise,} \end{cases} \tag{3.5}$$

The overall EX is thus,

$$\text{EX} = \frac{\sum_{n=1}^{N} \text{score}(X_n, \tilde{X}_n)}{N}, \tag{3.6}$$

where N is the total number of generations. The EX metric was particularly useful as some SQL queries can often output identical data but exhibit a different syntactic structure. For illustration, consider the two SQL queries in Appendix A.1. Though different, they produce the same data output when executed on the GtoPdb.

3.2.2.2 Exact Matching (EM)

In contrast, Exact Matching (EM), also used in benchmark datasets [7, 6], is an even more stringent metric than EX as it compares the generated SQL query S and the gold standard SQL query $\tilde{S}$ directly, rather than comparing their outputs. Formally,

$$\text{score}(S,\tilde{S}) = \begin{cases} 1, & \text{if } S = \tilde{S}, \\ 0, & \text{otherwise}, \end{cases} \tag{3.7}$$

and overall EM is thus,

$$\text{EM} = \frac{\sum_{n=1}^{N} \text{score}(S_n,\tilde{S}_n)}{N}, \tag{3.8}$$

where N is the number of generated queries. Within this research, the EM metric was very strict because it disregarded functionally equivalent queries if they differed in syntactic structure. In example, the two SQL queries in Appendix A.2 may look different but, when parsed, are identical.

3.2.2.3 Partial Execution Accuracy (PEX)

To answer the research question regarding the design of an evaluation metric, I propose Partial Execution Accuracy (PEX). I designed this metric as both EM and EX were strict; but I still wanted to capture when a generated SQL query S was missing some columns (or including additional columns). Formally, the PEX score was defined as,

$$\text{score}(X,\tilde{X}) = \begin{cases} 1, & \text{if } (\tilde{X} \subseteq X \text{ or } X \subseteq \tilde{X}), \\ 0, & \text{otherwise}, \end{cases} \tag{3.9}$$

where the relation $Z \subseteq W$ means:

1. The number of rows in Z is equal to the number of rows in W, and
2. Each row in Z is a subset of some row in W.

Thus, PEX accounted for cases in which the predicted SQL query output and the gold standard output contained the same information but exhibited a different number of columns. The PEX metric was then calculated as,

$$\text{PEX} = \frac{\sum_{n=1}^{N} \text{score}(X_n,\tilde{X}_n)}{N}, \tag{3.10}$$

where, again, N is the number of generated queries. Whereas EX and EM are strict in their comparisons, PEX tolerated differences in the number of returned columns if the data in one result set was contained within the other. This allowed for the capture of partially correct SQL that met most of the gold standard SQL requirements. For example, the two SQL queries in Appendix A.3 are different in terms of their output and SQL structure, but the data in the output of the first is contained within the output of the second. Therefore, these two SQL queries would be counted as partially accurate.

By combining these metrics (SER, NER, EX, EM, and PEX) I obtained a holistic view of both the robustness and accuracy of the generated SQL queries. I used these metrics to track and compare the performances of all explored methods.

3.3 Selected Large Language Models

To investigate text-to-SQL methods that utilise Large Language Models (LLMs), I first needed to choose a set of LLMs to research. With £500 worth of monthly credit from the University for accessing the OpenAI API, I experimented with a range of LLMs. I began by evaluating the following models:

o3-mini, o1, o1-mini, gpt-4o, gpt-4o-mini, gpt-4-turbo, gpt-4, gpt-3.5-turbo.

As the project progressed, I refined my selection, based on the key factors (cost, response time, performance, and usability) mentioned in Chapter 1. By considering these key factors, I was able to identify the most suitable LLM for my final zero-shot text-to-SQL pipeline.

3.4 Zero-Shot Methods

3.4.1 System Messages

As introduced in Chapter 2, the system message is the component of the prompt given to the LLM that directs its behaviour. For the task of text-to-SQL, it was essential to configure the system message so that the LLM was capable of consistently generating SQL queries for the GtoPdb. Therefore, when generating SQL, I initialised the system message as shown in Figure 3.1.

> You are a system that converts natural language queries into PostgreSQL queries. Here is the schema of the Guide to Pharmacology database; $\cdots$

Figure 3.1: Beginning of the system message used for generating SQL queries for the GtoPdb. The ellipsis ($\cdots$) represents the GtoPdb schema.

By specifying the LLM's role and providing the necessary schema information, I instantiated the LLM towards producing SQL queries that were both syntactically and semantically aligned with the GtoPdb. Notably, to ensure a controlled experiment the system message was reinitialised every time an SQL query was generated. To determine the best way to deliver the schema information within the system message, I explored multiple approaches. Based on relevant literature, these variations in schema representation (all are visualised in Figure 3.2) are defined as follows:

Text (S_T): This representation, used in [26, 27], is the most rudimentary and shows the entire database schema in plain text. Quite simply, all the tables and columns within the database are listed without explicitly defining the roles (table or column) of any database component. Consequently, the LLM must infer the schema structure on its own when generating an SQL query.

Basic (S_B): Inspired by previous work [24, 26, 25], this straightforward design extends the Text representation (S_T) by structuring the schema to show the LLM the database components by defining the tables and columns. However, this representation omits

details such as data types and primary/foreign key relationships. As a result, the LLM must deduce these details independently.

OpenAI Suggested (S_O): Based on OpenAI's prompt examples[1] (and evaluated in [26, 25]), this schema representation presents the entire database schema by giving the LLM the 'CREATE TABLE' statements for every table in the database, thus providing the LLM with all information about column types, primary keys, and foreign keys.

OpenAI Suggested (No Primary/Foreign Key Information) (S_{ON}): This version is identical to the OpenAI suggested schema representation but omits primary and foreign key details. Consequently, the LLM must infer these relationships on its own.

Text representation (S_T):

accessory_protein: object_id, full_name
⋯ xenobiotic_expression_refs: xenobiotic_expression_id, reference_id

Basic representation (S_B):

Table: accessory_protein, columns: (object_id, full_name)
⋯
Table: xenobiotic_expression_refs, columns: (xenobiotic_expression_id, reference_id)

OpenAI suggested representation (S_O):

```
CREATE TABLE accessory_protein (
    object_id int4 NOT NULL,
    full_name varchar(1000) NULL,
    CONSTRAINT accessory_protein_pk PRIMARY KEY (object_id),
    CONSTRAINT object_accessory_protein_fk FOREIGN KEY (object_id)
    REFERENCES "object"(object_id)
);
⋯
```

OpenAI suggested representation (without PK/FK) (S_{ON}):

```
CREATE TABLE accessory_protein (
    object_id int4 NOT NULL,
    full_name varchar(1000) NULL
);
⋯
```

Figure 3.2: Visualisation of each schema representation (Text S_T, Basic S_B, OpenAI S_O and OpenAI without PK/FK information S_{ON}) used within the system messages for generating SQL queries. The ellipsis (⋯) represents missing schema information.

To create both S_T and S_B system messages, I crafted the SQL queries shown in Appendix B.1 and B.2, respectively. For the remaining system messages (S_O and S_{ON}), I obtained

[1] OpenAI's text-to-SQL suggested system message

the Data Definition Language (DDL) file containing the syntax for creating tables. I then applied regular expressions to the DDL file to extract the 'CREATE TABLE' statements required for both prompts. For S_O, I extracted the complete 'CREATE TABLE' statements, including all column definitions, constraints, and comments. For S_{ON}, I modified the extraction to exclude primary and foreign key information, ensuring that only the essential schema details were included without relational constraints.

3.4.2 Schema-linking

In addition to the system messages defined previously, I will now introduce the concept of schema-linking [38, 40, 41, 24], a popular text-to-SQL approach that aims to reduce the amount of schema information used when generating an SQL query for a given NL question. This ensured that the LLM was given only a concise subset of the GtoPdb schema when generating an SQL query; refer to Figure 3.3 for an example of the basic schema representation when schema-linking was applied. Appendix C.1 illustrates the other system messages (S_T, S_O, S_{ON}) before and after schema-linking.

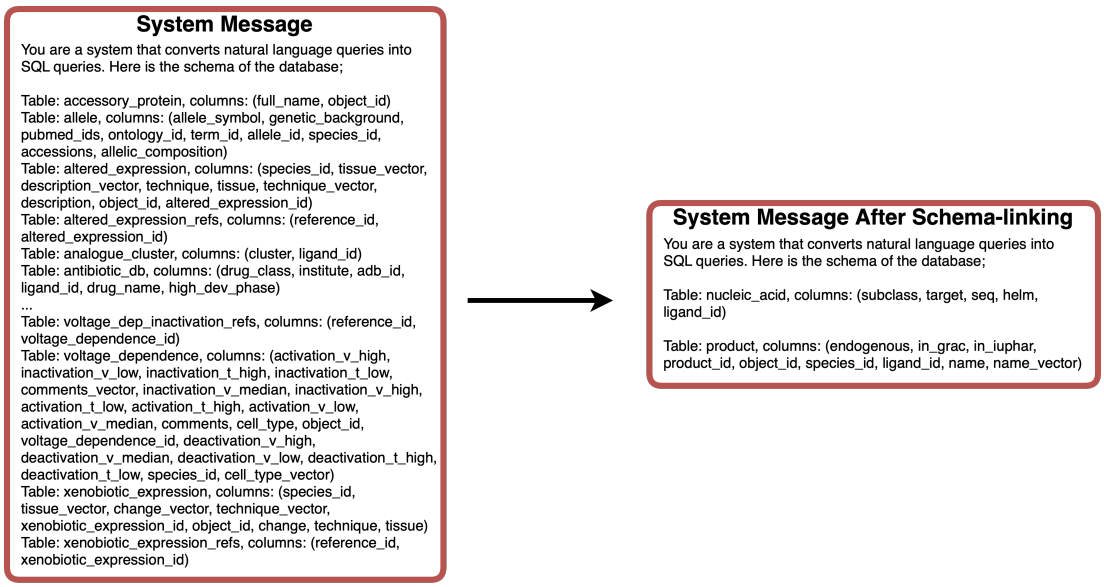


Figure 3.3: Visualisation of the Basic schema representation (S_B) before and after schema-linking. The ellipsis $(\cdots)$ in the box (on the left) represents missing schema information.

To perform schema-linking, a separate invocation of the LLM (tailored towards identifying only relevant schema information) is used. Prior to generating an SQL query for a given NL question, the LLM is asked to identify the schema components it believes are relevant to the NL question. Once these components are determined, they are extracted and used to form a concise, focused schema representation for use within the SQL generating system message. Figure 3.4 outlines the roadmap for the multi-table schema-linking approach defined in due course.

For this research, I considered several schema-linking approaches from relevant literature which utilised LLM-based strategies to perform schema-linking. Notably, these

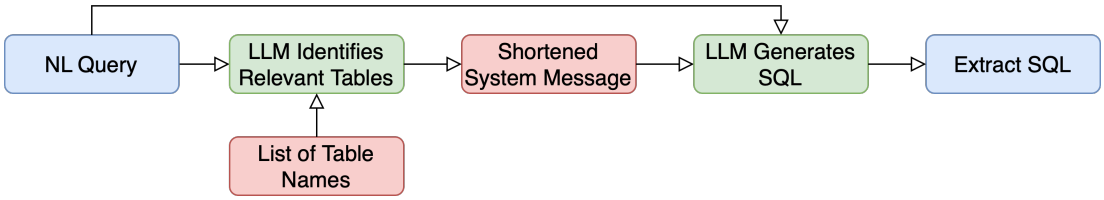


Figure 3.4: High-level roadmap of multi-table schema-linking (SL_{MT}). The blue steps denote the user input and output, the LLM required steps are highlighted in green, and the steps in red are those involving the database schema information.

schema-linking methods achieved solid performance on the BIRD benchmark [7], motivating the use of similar methods in this research. Within all schema-linking approaches, I used a straightforward version of Chain-of-Thought (CoT) prompting [17] whereby the LLM was asked to explain its reasoning step-by-step, an approach used to help the LLMs (such as **gpt-4o**) that lack automatic reasoning capabilities. The LLM then articulated its intermediate reasoning steps, much like a human would explain their thought process, rather than simply providing one final answer. These schema-linking methods can be defined as follows:

Single-column Schema-linking (SL_C): Inspired by [40], this approach makes the LLM evaluate each column in the database independently, without considering additional contextual information from the rest of the schema (see Appendix C.2 for the system message). The LLM then determines if the column could be relevant to the NL question.

Table-to-column Schema-linking (SL_{TC}): As used in [40, 41, 24], this two-step process first prompts the LLM to identify the tables relevant to the NL question (see Appendix C.3 for the system message). In the second step, the LLM is asked to specify which columns within those tables are pertinent (see Appendix C.4).

Multi-table Schema-linking (SL_{MT}): This method corresponds to the first step of SL_{TC} (used in [40, 41, 24]), where the LLM determines the relevant tables using the system message shown in Appendix C.3. Unlike SL_{TC}, it does not filter down to specific columns, leaving all columns in each identified table available for consideration.

To implement each of the schema-linking methods, I deconstructed the GtoPdb schema information into an accessible format so that it could be shortened using only relevant schema information. To do this, I parsed the GtoPdb schema information into appropriate JSON formatting (shown in Appendix C.5). With the schema information being accessible in this JSON format, I was able to design a system that effectively rebuilt each system message using only relevant schema information identified through each researched schema-linking method.

3.4.3 Self-correction

Next, I considered self-correction [24, 41, 42, 43, 44], a technique that has a similar workflow to Retrieval-Augmented Generation (RAG) [45, 46]. Self-correction involved improving the quality of generated SQL queries by giving the LLM an opportunity to revise its output in response to errors. Figure 3.5 shows how self-correction enables

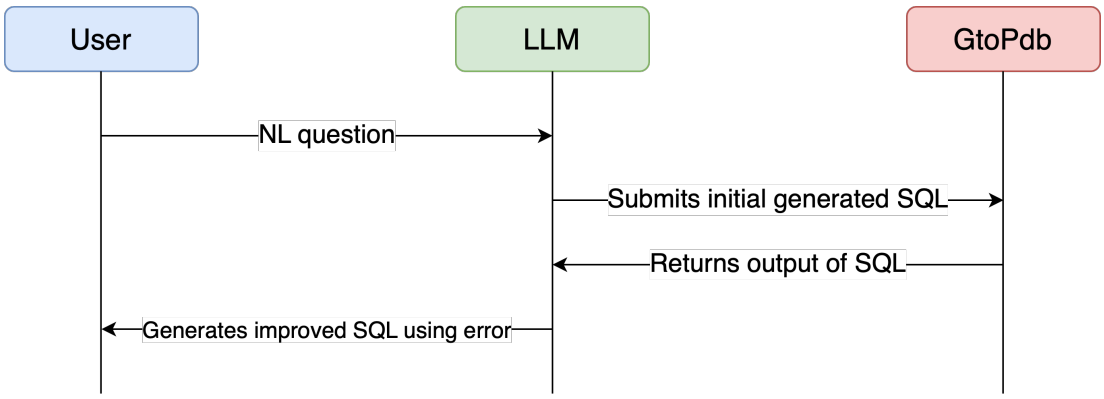


Figure 3.5: Visualisation of the self-correction process when a user asks the LLM for an SQL query. The LLM's initial SQL query is ran on the GtoPdb and if there is an error the LLM can use this information to output an improved response.

the LLM to refine its generated SQL queries using SQL-error feedback. Thus, when given an NL question via the user message and the corresponding database schema (either in full or reduced form via schema-linking) through the system message, the LLM first produces an initial SQL query that is executed on the database. If an error occurs, the user message is augmented with the NL question, the initial SQL query, and the error message while the system message remains unchanged. With this enriched user message, the LLM then corrects its query, and the revised SQL query becomes the final output. For this research, I considered two forms of self-correction:

Syntax Correction: Used when the LLM outputted an SQL query that could not be executed on the GtoPdb because it was syntactically incorrect. In this scenario, the LLM is therefore given the corresponding error message (and possibly a hint) when the SQL query is executed on the database. Using this information, the LLM revises its SQL query in hopes of producing a syntactically sound one. Figure 3.6 shows the augmented user message when using syntax correction.

Empty Output Correction: This approach was utilised when the LLM produced an SQL query that could be executed on the GtoPdb but produced no output. Thus, the LLM must revise its query to try to generate an SQL that does produce data output. Figure 3.7 shows the augmented user message used for empty output correction.

> You were asked to generate an SQL query for the natural language question: {nl_query}. You previously generated the following SQL query: {sql_query} which has the following error: {error_message}. Please correct the query

Figure 3.6: The augmented user message when using syntax correction. The LLM is given the SQL query it generated (blue), the NL question which it was given (green) and the error message of the generated SQL query when executed on the GtoPdb (red).

> You were asked to generate an SQL query for the natural language question: {nl_query}. You previously generated the following SQL query: {sql_query} When executed on the Guide to Pharmacology database the query was syntactically correct but the output was empty, please provide the corrected SQL query.

Figure 3.7: The augmented user message when using empty output correction. The LLM is given the SQL query it generated (blue), the NL question which it was given (green), and it is told that there was no data output when the SQL was executed on the database.

Note that these methods are nestable. In example, after applying self-correction to repair a syntactically incorrect query so that it executes successfully, the query may still return an empty result. In such cases, empty output correction can be applied to refine the query in hopes of generating a non-empty output.

3.4.4 Self-validation

For the proposed self-validation, when the LLM generated an executable SQL query that returned non-empty results, it was given a preview of the output, specifically the first three rows. Based on this information, the LLM was then able to evaluate whether the SQL query adequately answered the NL question, thus deciding whether to retain the original SQL query or revise it. The workflow visualisation of self-validation is identical to that of self-correction (seen in Figure 3.5), except that after the initial generated SQL was executed on the GtoPdb, the LLM was provided with a preview of the output rather than error information (see Appendix D for self-validation workflow). Figure 3.8 displays the augmented user message I used for self-validation. Again, self-validation was nestable, so it can be applied on top of other methods (e.g. self-correction approaches).

> You were asked to generate an SQL query for the natural language question: {nl_query}. You previously generated the following SQL query: {sql_query}. When executed on the database the first 3 rows of the output were: {output}. Use this information to assess whether your query is correct. If you believe your query is correct, please provide the same SQL query. If you believe your query is incorrect, please provide the corrected SQL query.

Figure 3.8: The augmented user message for self-validation. The LLM is given the SQL query it generated (blue), the NL question which it was given (green) and the output of its generated SQL query (red).

3.5 Few-shot Learning Methods

Up to this point, I have defined only zero-shot prompting methods, where the LLM generates SQL queries without being provided with any domain-specific examples (NL-SQL pairs from the GtoPdb). However, prior research has demonstrated that incorporating a small number of NL-SQL example pairs (few-shot learning [8, 28]) significantly enhances the quality of generated SQL queries [27, 26, 24].

Like some zero-shot methods, few-shot learning is implemented by augmenting the user message with a selection of NL-SQL example pairs. In practice, if the LLM is given one example, this is referred to as 1-shot learning; if provided with two examples, it is 2-shot learning. More generally, this approach can be denoted as k-shot learning, where k represents an arbitrary integer number of examples. For this project, I adopted the user message augmentation given in [27], which can be seen in Figure 3.9. This representation appends NL-SQL example pairs after the NL question, providing the LLM with in-context examples from which it can infer patterns and understand the GtoPdb schema better, thus improving the calibre of the generated SQL queries.

{nl_query}
Some examples include;
{nl_query_1}
{sql_query_1}
...
{nl_query_k}
{sql_query_k}

Figure 3.9: User message augmentation for k-shot learning where k NL-SQL examples are listed (for each example pair the SQL is in blue and the NL example is in red) after the NL question (green). The ellipsis ($\cdots$) represents NL-SQL example pairs not shown.

3.5.1 Criteria For Picking Few-shot Examples

Selecting appropriate few-shot examples was very important, since giving the LLM examples relevant to the posed NL question allowed the LLM to see similar NL-SQL pairs. For this research, I considered the following criteria when choosing NL-SQL examples:

Random Examples: Serving as a baseline for all few-shot learning methods, this approach gives the LLM a set of NL-SQL example pairs chosen at random. This method, evaluated in [27, 47, 26, 48], describes the most basic version of few-shot learning in hopes of understanding its impact on SQL generation.

Random Examples With Different Difficulties: Similar to the random examples approach, this method provides another baseline by choosing one random example from each difficulty category (easy, easy-moderate, moderate-hard, hard). This ensures that

the LLM is exposed to a diverse range of query complexities, providing exactly four examples (4-shot).

Similar SQL Examples: Unlike previous methods, the LLM first generates an initial SQL query as an answer to a given NL question. Using this initially generated SQL query, NL-SQL examples are chosen based on each SQL query's similarity to the initially generated SQL query [26, 47]. Thus, the LLM focuses on examples of NL-SQL example pairs with a similar SQL structure.

Similar NL Question Examples: In this setting, NL-SQL example pairs are chosen based on the NL questions and their similarity to the NL question asked of the LLM. Therefore, the LLM can learn from NL-SQL example pairs relevant to the NL question [48, 47].

3.5.2 Ranking Similarity

Both the similar SQL and similar NL question examples require the LLM to be given few-shot examples based on their relevance to an NL question. To do this, I invoked scikit-learn's [49, 50] `TfidfVectorizer`[2], which converts text into TF-IDF (Term Frequency-Inverse Document Frequency) features, a common NLP technique originating from [51, 52]. These features serve as numerical representations that capture the importance of words in a string (e.g. NL question in the held-out set) relative to a collection of several strings (e.g. NL questions in the training set). To compute the similarity between these vectorised representations, I used the cosine similarity score which can be calculated for two vectors $\mathbf{x}$ and $\mathbf{y}$ as,

$$\text{similarity}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}||\ ||\mathbf{y}||}. \tag{3.11}$$

Here, $\mathbf{x} \cdot \mathbf{y}$ denotes the dot product of the vectors, $||\mathbf{x}||$ and $||\mathbf{y}||$ denotes their respective Euclidean norms. By computing the cosine similarity scores [53] between a held-out example (either the NL query itself or an initially generated SQL) and each training example, the training set was ranked by its similarity to the held-out example. The top-k examples from this ranking were then selected for use in the few-shot context, corresponding to k-shot learning.

[2] TfidfVectorizer documentation

Chapter 4

Text-to-SQL Pipeline

This chapter focuses on constructing an effective zero-shot text-to-SQL pipeline by investigating each of the zero-shot methods outlined in Chapter 3. By evaluating each approach on the training set, I developed a text-to-SQL pipeline suitable for evaluation (combined with few-shot learning) on the held-out set. I will therefore answer the following research questions:

- Which database schema representation is the best?
- Does the proposed self-validation have any effect on producing SQL queries?
- Which LLM is the most suitable?

I refined my text-to-SQL pipeline by evaluating and selecting each component based on the key factors outlined in Chapter 1. I began by implementing each system message defined earlier, followed by the implementation of each schema-linking method. Next, I incorporated the method of self-correction, and investigated the proposed self-validation method.

4.1 System Message Exploration

The LLM's system message needs to be populated with information so that it is able to produce coherent SQL for the GtoPdb. Therefore, this chapter begins by investigating each of the system messages (defined in Chapter 3) on the training set NL questions.

4.1.1 Comparative Analysis

I configured all chosen OpenAI LLMs with each of the system messages then I gave the LLM every training set NL question through the user message, making sure to reinitialise the system message each time. Table 4.1 reports the EX (generated SQL output matches the gold standard SQL output), PEX (generated SQL output matches the gold standard SQL output but has extra/missing columns), and SER (generated SQL can be executed) metrics for each LLM on the training set. It is important to note that, at the time of investigation, the OpenAI API did not support system message functionality for

the **o1-mini** model. As a workaround, the **o1-mini** model received the system message as part of the user message, with the NL question appended at the end. Additionally, entries marked as 'n/a' indicate that certain models did not have sufficient input token capacity to utilise the full system message.

LLM	S_T			S_B			S_O			S_{ON}		
	EX	**PEX**	**SER**	**EX**	**PEX**	**SER**	**EX**	**PEX**	**SER**	**EX**	**PEX**	**SER**
o3-mini	5.88	15.69	90.20	5.88	15.69	88.24	5.88	11.76	98.04	**9.80**	17.65	**100**
o1	5.88	15.69	96.08	3.92	11.76	98.04	7.84	**25.49**	**100**	5.88	17.65	96.08
o1-mini*	3.92	9.80	76.47	3.92	11.76	82.35	1.96	9.80	92.16	1.96	13.73	92.16
gpt-4o	0.00	11.76	82.35	5.88	13.73	82.35	3.92	11.76	92.16	5.88	15.69	94.12
gpt-4o-mini	1.96	7.84	54.90	1.96	7.84	58.82	0.00	5.88	60.78	1.96	5.88	70.59
gpt-4-turbo	1.96	7.84	76.47	0.00	3.92	72.55	1.96	7.84	70.59	3.92	11.76	86.27
gpt-4	0.00	5.88	76.47	0.00	7.84	72.55	n/a	n/a	n/a	n/a	n/a	n/a
gpt-3.5-turbo	1.96	5.88	47.06	1.96	5.88	52.94	n/a	n/a	n/a	0.00	1.96	60.78

Table 4.1: The PEX, EX, and SER scores on the training set for each investigated system message. The asterisk on **o1-mini*** indicates the workaround for system message support, while 'n/a' denotes insufficient input token capacity.

Table 4.1 shows that the highest scores for all evaluation metrics occurred when using reasoning models (**o1** and **o3-mini**) combined with either S_O or S_{ON}. For instance, the **o1** model when using S_O obtained the highest PEX score (25.49%) but was outperformed on EX by **o3-mini** when leveraging S_{ON} (9.80%). This result was unsurprising as the reasoning models are regarded as the most powerful OpenAI LLMs and both S_O and S_{ON} contained much more information than all other system messages.

Figure 4.1 helps analyse the robustness of each experiment by displaying the SER scores. Among the evaluated system messages, S_O with the **o1** model and S_{ON} with the brand new **o3-mini** model demonstrated the highest SER scores. Both achieved a 100% SER score, meaning they successfully generated an executable SQL query for every NL question in the training set. When considering all LLMs, both S_O and S_{ON} were the most syntactically accurate system messages (apart from when used by **gpt-4-turbo**) which is an expected result, since both S_O and S_{ON} contained the syntactic information of each table in the GtoPdb. Interestingly, S_{ON} took the mantle ahead of S_O as S_{ON} outperformed S_O on every LLM (apart from **o1**). This result seemed counter-intuitive, since S_O contained all the constraints as well as the primary/foreign key information, which was missing from S_{ON}. Thus, suggesting that the primary/foreign key information did not always aid the LLMs when producing syntactically executable queries.

Although not reported in Table 4.1, there was never a single instance of any generated query being able to yield an exact match to its gold standard counterpart in the training set (EM was 0 for all). In some cases, the LLM was able to produce an almost identical SQL query but lacked a small difference, making it fail the exact match test. In contrast, the same system messages combined with either **gpt-4** or **gpt-3.5-turbo** were able to achieve an EM of at least 40% [27] on the Spider 1.0 benchmark dataset [6]. This suggests that the dataset of GtoPdb NL-SQL pairs was more complex than those used within benchmark datasets.

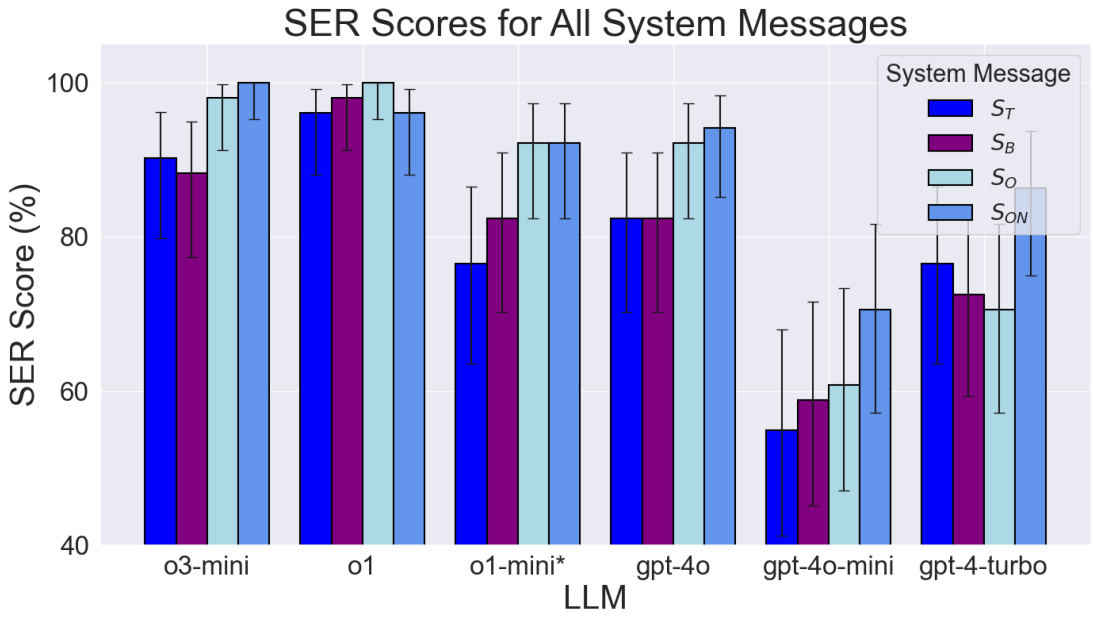


Figure 4.1: The SER scores for all LLMs (apart from **gpt-4** and **gpt-3.5-turbo**) when using each system message on the training set. Error bars indicate 95% confidence intervals computed using a Beta distribution (Jeffreys prior [54]), where for each proportion p out of n trials, the posterior parameters $a = np + 0.5$ and $b = n(1-p) + 0.5$ derive the interval.

4.1.2 Refining LLM Selection

For the remainder of the research, I decided to narrow the focus to a smaller subset of LLMs – **o3-mini, o1, o1-mini, gpt-4o, and gpt-4o-mini**. This decision was based on both performance and cost considerations. Firstly, according to OpenAI's recommendations, **gpt-4o-mini** should replace **gpt-3.5-turbo**, as it offers better performance at a lower cost[1]. Additionally, the initial findings (see Table 4.1) indicated that **gpt-4o** consistently outperformed both **gpt-4** and **gpt-4-turbo**. Beyond performance, **gpt-4o** was significantly more cost-effective than these models. As a result, continuing to investigate **gpt-4** and **gpt-4-turbo** was not justifiable. By narrowing the focus to **o3-mini, o1, o1-mini, gpt-4o, and gpt-4o-mini**, it ensured that my approach remained appropriate for the duration of the investigation.

4.2 Cost Efficiency and Token Optimisation

Although the system messages produced promising results, they introduced a big problem with affordability, which is because, when using the API, OpenAI charges based on the number of tokens used within each prompt. Since the system messages included every table (206 total) and column (1,571 total) in the database, the resulting prompts were extremely lengthy (especially for S_{ON} and S_O). This excessive token count not only increased costs (the cost to produce the data in Table 4.1 was roughly £300

[1] OpenAI's summary of their models

alone) but also rendered several models (e.g. the most expensive model **o1**) unaffordable for the remainder of the project. To address this issue, I implemented schema-linking which involves dynamically associating only the parts of the schema that are relevant to the given NL question. By presenting the LLM with a concise and context-specific system message, the token count of the system message was substantially reduced, thereby lowering costs and making the system message sustainable throughout the remainder of this research.

The schema-linking methods required the LLM to produce output in a structured JSON format. While all the selected OpenAI LLMs supported the `response_format` argument via the API, forcing the LLM to output in JSON format, **o1-mini** did not [19]. As a result, **o1-mini** lacked the usability needed to consistently generate output in the required format, making its results impractical to extract. Therefore, I restricted the investigation to **o3-mini**, **o1**, **gpt-4o**, and **gpt-4o-mini**.

4.2.1 Single-Column Schema-linking

Firstly, I implemented the single-column schema-linking (SL_C) experiment but quickly discovered that it was not feasible due to a massive wait time. This extreme time consumption can be attributed to the process of independently evaluating every single column in the database. The independent evaluation meant that the API had to be called $1,571$ times for every NL question in the training set. The API calls took too long in this scenario to be considered a viable option; therefore, I regarded the single-column schema-linking method as impractical for this research. If deploying the LLM via a GPU or through some other high-compute process, it may be possible to investigate single-column schema-linking, but here, it was not feasible.

4.2.2 Multi-table Schema-linking

In the multi-table schema-linking approach (SL_{MT}), the LLM was first prompted, prior to generating an SQL query, to identify which tables in the database were relevant to a given NL question. Once the relevant tables were identified, my pipeline automatically rebuilt each system message (S_T, S_B, S_O, and S_{ON}) by extracting only the schema information corresponding to those tables. This new system message, containing only the pertinent table details, was then provided to the LLM to generate an SQL query that addresses the NL question. The results of this experiment are shown in Table 4.2.

LLM	S_T			S_B			S_O			S_{ON}		
	EX	**PEX**	**SER**	**EX**	**PEX**	**SER**	**EX**	**PEX**	**SER**	**EX**	**PEX**	**SER**
o3-mini	1.96	7.84	70.59	0.00	7.84	62.75	5.88	13.73	82.35	0.00	9.80	86.27
o1	5.88	**17.65**	88.24	**7.84**	13.73	90.20	1.96	13.73	98.04	3.92	**17.65**	**100**
gpt-4o	1.96	5.88	66.67	1.96	5.88	70.59	1.96	9.80	88.24	1.96	7.84	90.20
gpt-4o-mini	0.00	1.96	62.75	0.00	1.96	64.71	1.96	5.88	82.35	1.96	5.88	80.39

Table 4.2: The PEX, EX, and SER scores on the training set for each investigated system message when using multi-table schema-linking (SL_{MT}).

4.2.3 Table-to-column Schema-linking

Building upon the multi-table schema-linking approach (SL_{MT}), the table-to-column schema-linking method (SL_{TC}) introduced an additional refinement step. In SL_{TC}, after the LLM identified the relevant tables for a given NL query, it was further prompted to determine which specific columns within those tables were also relevant. The schema-linking pipeline step then reconstructed each system message (S_T, S_B, S_O, and S_{ON}) to include only the relevant tables and identified columns. The results for the SL_{TC} experiment are presented in Table 4.3.

LLM	S_T			S_B			S_O			S_{ON}		
	EX	**PEX**	**SER**	**EX**	**PEX**	**SER**	**EX**	**PEX**	**SER**	**EX**	**PEX**	**SER**
o3-mini	0.00	3.92	35.49	0.00	3.92	35.29	3.92	11.76	80.39	1.96	5.88	76.47
o1	3.92	11.76	68.63	**5.88**	11.76	68.63	**5.88**	**13.73**	**94.12**	3.92	9.80	84.31
gpt-4o	1.96	1.96	43.14	0.00	3.92	45.10	1.96	5.88	78.43	1.96	5.88	72.55
gpt-4o-mini	0.00	1.96	45.10	0.00	3.92	43.14	0.00	1.96	64.17	0.00	3.92	62.75

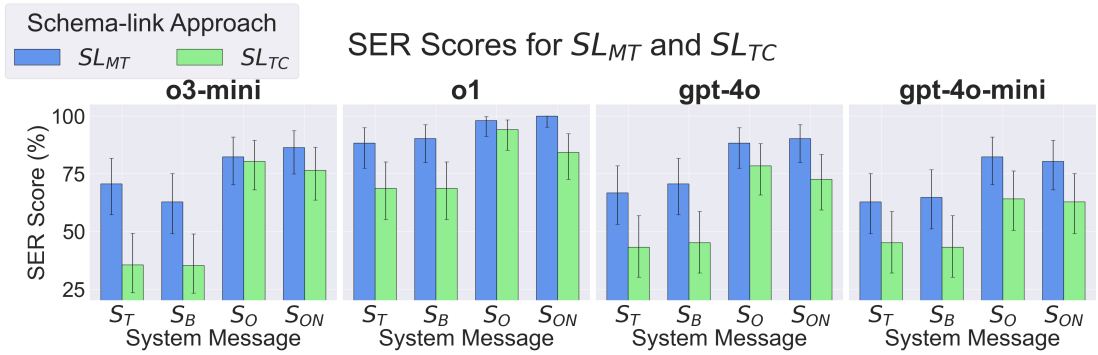
Table 4.3: The PEX, EX, and SER scores on the training set for each investigated system message when using table-to-column schema-linking (SL_{TC}).

4.2.4 Comparative Analysis

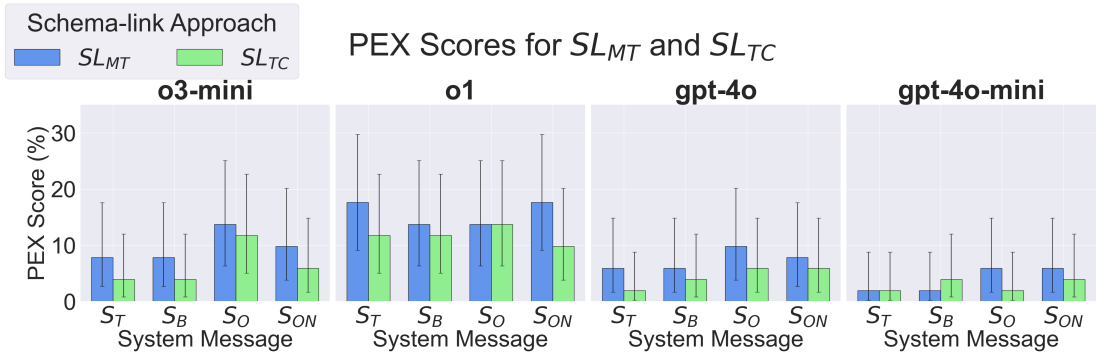
Before the comparison of the schema-linking results begins, it is important to acknowledge the decrease in performance when considering the previous system message results given in Table 4.1. Across almost all schema-linking experiments, the reported results (seen in Table 4.2 and 4.3) fell short of the scores observed when the entire schema was given to the LLM (seen in Table 4.1). This highlighted the LLM's inability to select the correct schema information when attempting to refine the system messages. However, since the massive system messages were deemed unaffordable for the remainder of this research, schema-linking remained the preferred approach.

Figure 4.2a reports the SER scores for both schema-linking methods; this indicated that SL_{MT} consistently outperformed SL_{TC} across all instances. Therefore, the extra refinement step of narrowing down to relevant columns led to a higher rate of syntactically incorrect queries. Notably, the **o3-mini** model suffered a serious decrease when generating syntactically correct SQL queries after the extra refinement step was added. Moreover, Figure 4.2b displays the PEX scores for each schema-linking method, revealing the same trend as Figure 4.2a, that SL_{MT} performed on par with or surpassed SL_{TC} in nearly all experiments. The only time SL_{TC} outperformed SL_{MT} was when **gpt-4o-mini** was combined with the S_B system message; however, the error bars suggested that this may not be statistically significant.

These findings agreed that SL_{MT} significantly outperformed SL_{TC}. Also, SL_{TC} required an additional step (identifying columns for each detected table), so it inherently took longer than SL_{MT}. Given these findings, SL_{MT} was adopted into the pipeline for the remainder of the research.



(a) SER scores.



(b) PEX scores.

Figure 4.2: SER and PEX scores for both schema-linking approaches on the training set. Error bars indicate 95% confidence intervals computed using a Beta distribution (Jeffreys prior [54]), where for each proportion p out of n trials the posterior parameters $a = np + 0.5$ and $b = n(1-p) + 0.5$ were used to derive the interval.

4.2.5 Decrease in Token Count

The original schema representations included information on every table and column in the GtoPdb, resulting in extremely high token counts (e.g. over 20,000 tokens for S_O which can be seen in Appendix C.6). Figure 4.3 displays the reduction in token usage (on the training set) across all system messages before and after SL_{MT} was used, using a logarithmic scale to effectively show the magnitude of differences in token count. These token counts were computed using the `tiktoken` package from OpenAI [55], ensuring consistent tokenisation across all system messages.

The token counts differences indicated that the SL_{MT} schema-linking significantly reduced token counts across all system messages, with reductions approaching an order of magnitude (e.g. a decrease from approximately 10,000 to 1,000 or similar). The relatively small confidence intervals also suggested that the reduction was consistent across different system messages. In particular, the token counts of the much larger system messages (S_O and S_{ON}) were both reduced to well below the full token count of the two smallest system messages (S_B and S_T).

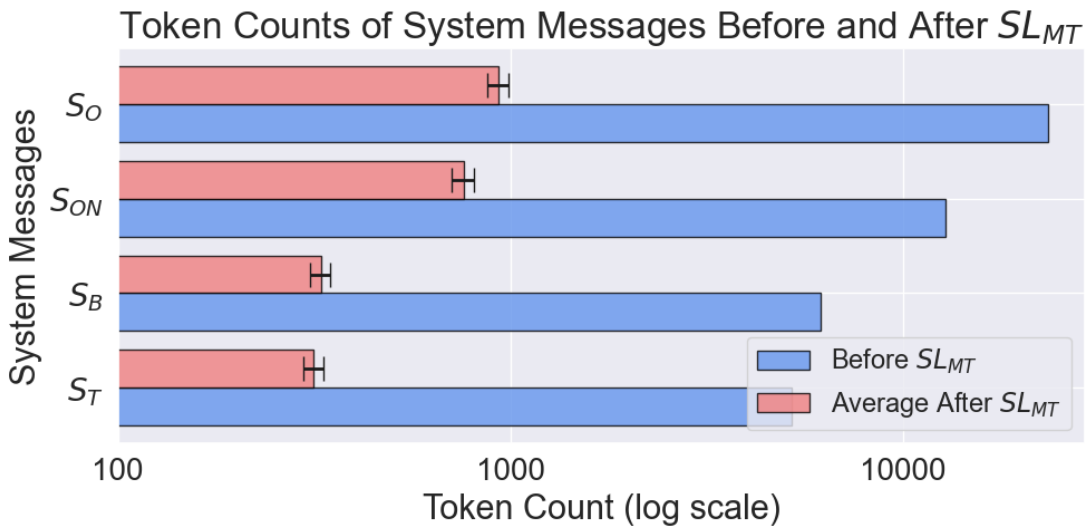


Figure 4.3: Difference in token count before and after SL_{MT} on the training set. The token count is represented in log scale, and the error bars represent the 95% confidence intervals for the average schema-linked token counts. The error bars were computed by multiplying the standard error of the mean by the appropriate t-distribution critical value [56].

4.2.6 Time to Respond

While schema-linking managed to address affordability challenges, it introduced a multi-stage prompting process. Firstly, the LLM was prompted to identify relevant schema information. It was then prompted again to generate the corresponding SQL query. This two-step process increased the overall response time, as it required two separate LLM invocations. Figure 4.4 illustrates the average response times for each LLM across all four system messages when outputting an SQL query using multi-table schema-linking (SL_{MT}).

Despite achieving some of the highest scores across all evaluation metrics, the **o1**

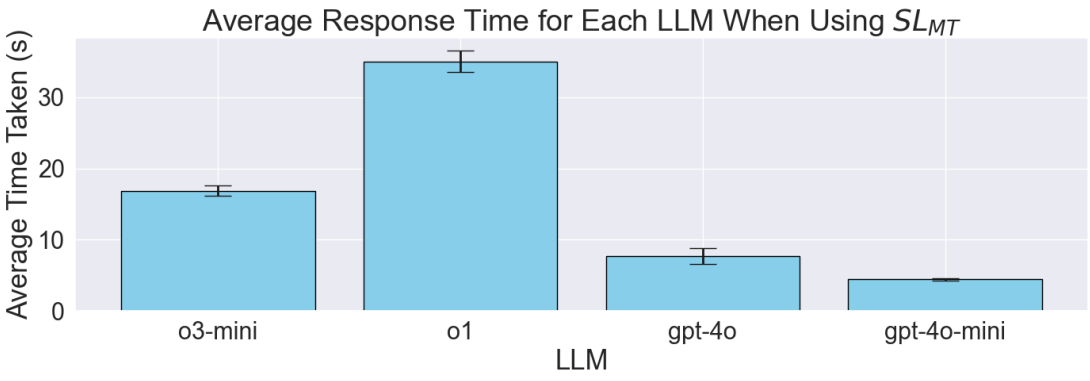


Figure 4.4: Average response time (in seconds) for each LLM to complete SL_{MT} across all system messages. Error bars represent 95% confidence intervals, computed as 1.96 times the standard error of the mean [57] for the response times.

reasoning model needed, on average, 35.03 seconds to produce an SQL query when using SL_{MT}. This response time was considerably longer than all other LLMs, meaning a user would need to wait over half a minute for an SQL response—a response time I have deemed unacceptable. Taking these factors into account, the **o1** model was excluded from further investigation. Its extended response time rendered it impractical for text-to-SQL, and even after addressing the affordability issue, the **o1** model remained significantly more expensive than all other investigated LLMs (six times more expensive than the next most expensive model **gpt-4o** as of April 2025).

4.3 Improving Robustness

Considering all previous experiments, it is apparent that several of the LLMs generated syntactically incorrect SQL queries (SER scores were not always 100%). Since these SQL queries cannot be executed, their intended outputs cannot be evaluated to determine if they are correct, not to mention if they possibly produced an empty output. Therefore, to improve the robustness (the SER and NER scores) of the generated queries, I implemented self-correction [24, 41, 42, 43, 44].

4.3.1 Syntax Correction

The LLM was given the ability to see the erroneous message reported back from an invalid SQL query it generated. This feedback not only indicated that the SQL query was incorrect but also offered a hint as to why it failed. Using this information, the LLM was allowed an opportunity to revise and regenerate the SQL query. To implement syntax correction, it was applied upon the SQL queries produced by the selected schema-linking method (SL_{MT}). Each LLM was allowed to correct any query it initially generated incorrectly (from results shown in Table 4.2). To maintain schema consistency, the shortened system message, produced from SL_{MT}, was reused by the LLM when generating its revised SQL query. Table 4.4 shows the results of syntax correction.

LLM	S_T			S_B			S_O			S_{ON}		
	EX	**PEX**	**SER**	**EX**	**PEX**	**SER**	**EX**	**PEX**	**SER**	**EX**	**PEX**	**SER**
o3-mini	1.96	7.84	86.27	0.00	7.84	82.35	**5.88**	**13.73**	88.24	0.00	9.80	94.12
gpt-4o	1.96	5.88	84.31	3.92	7.84	90.20	3.92	11.76	**98.04**	1.96	7.84	**98.04**
gpt-4o-mini	0.00	1.96	84.31	0.00	1.96	80.39	1.96	5.88	92.16	1.96	5.88	88.24

Table 4.4: The EX, PEX and SER scores when using SL_{MT} + syntax correction on the training set.

To assess the impact of syntax correction, Figure 4.5 displays the SER scores from the SL_{MT} experiment before (seen in Table 4.2) and after applying syntax correction (seen in Table 4.4). This showed that syntax correction had an instrumental impact on SER: across all LLMs and system messages, the SER scores improved significantly, indicating that the LLMs now generated a higher proportion of syntactically sound SQL queries. In contrast, the improvements in PEX and EX scores were negligible. In

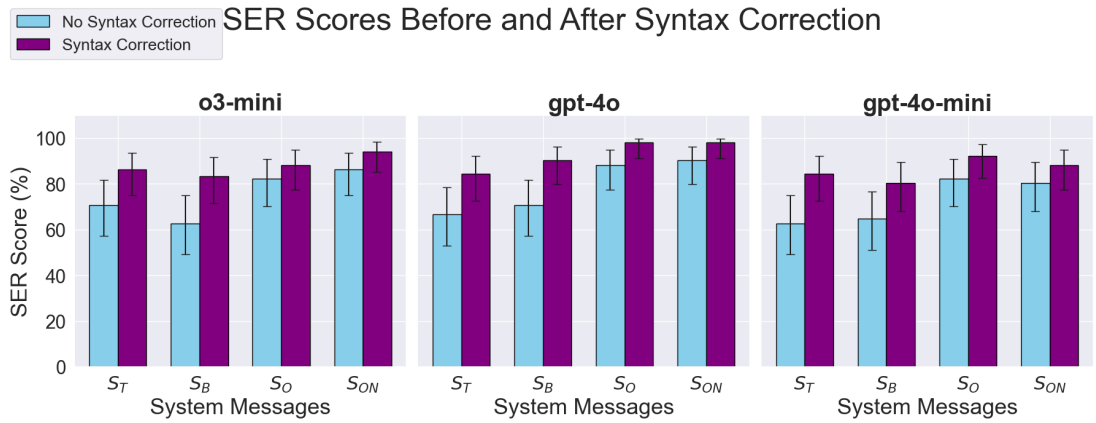


Figure 4.5: The effect of syntax correction on SER score when using SL_{MT} + syntax correction on the training set. Error bars indicate 95% confidence intervals computed using a Beta distribution (Jeffreys prior [54]), where for each proportion p out of n trials, the posterior parameters $a = np + 0.5$ and $b = n(1-p) + 0.5$ were used to derive the interval.

fact, when comparing the multi-table schema-linking results before and after syntax correction (Table 4.2 and Table 4.4), only one notable improvement was observed, specifically the **gpt-4o** model when using the S_B system message, which showed a 1.96% increase in both PEX and EX (insignificant as it fell within the margin of error). This suggested that while syntax correction effectively enhanced the robustness of SQL queries, it did not substantially improve the accuracy of the generated SQL, a result that could be attributed to the LLM's inability to select the correct schema information when it completed the initial schema-linking step. Nevertheless, because syntax correction significantly improved the robustness of the generated SQL queries, it was incorporated into the pipeline. Furthermore, Table 4.4 indicated that the highest scores across all evaluation metrics occurred when using both the S_O and S_{ON} system messages. Therefore, for the remainder of the research, subsequent methods focused exclusively on these two system messages.

4.3.2 Empty Output Correction

With significantly improved SER scores, it is crucial to acknowledge that several of the generated SQL queries, although they executed, produced no output. Table 4.5 shows the Non-empty Execution Rate (NER) scores for the previous experiment of SL_{MT} with syntax correction applied. While the SER (seen in Table 4.4) scores may have improved,

LLM	S_O	S_{ON}
o3-mini	49.02	41.18
gpt-4o	41.18	45.10
gpt-4o-mini	41.18	33.33

Table 4.5: The NER scores for each LLM when using SL_{MT} + syntax correction on the training set.

the percentage of SQL queries that actually produced data output remained low.

To improve this, the second self-correction phase, empty output correction, was implemented, whereby if the output of a generated SQL query was empty, the LLM was allowed to regenerate its response. The results of this experiment are reported in Table 4.6. While the accuracy metrics (EX and PEX) showed no improvement, the NER scores generally either improved or remained consistent with previous results (seen in Table 4.5). However, this benefit came with a trade-off as some SER scores decreased, a reduction that must have occurred when the LLM re-generated an SQL query and, in doing so, produced a syntactically incorrect one. Nevertheless, a query that produced a non-empty output was considered more valuable than one that was merely syntactically correct. Thus, the slight reduction in SER was deemed an acceptable trade-off for achieving a higher NER, and so output correction was retained in the pipeline.

LLM	S_O				S_{ON}			
	EX	**PEX**	**SER**	**NER**	**EX**	**PEX**	**SER**	**NER**
o3-mini	**5.88**	**13.73**	84.31	**52.94**	0.00	9.80	83.35	41.18
gpt-4o	3.92	11.76	**96.08**	45.10	1.96	7.84	**96.08**	49.02
gpt-4o-mini	1.96	5.88	88.24	45.10	1.96	5.88	88.24	37.25

Table 4.6: The EX, PEX and SER results for each LLM when using SL_{MT} + syntax correction + empty output correction on the training set.

4.4 Self-validation

Having achieved a higher rate of SQL queries with a non-empty output, the proposed self-validation was investigated. Self-validation provided the LLM with the (non-empty) output of its previously generated SQL query along with the original NL question. Using this output, the LLM revised its generated SQL to better answer the NL question. The results of this experiment are given in Table 4.7.

LLM	S_O				S_{ON}			
	EX	**PEX**	**SER**	**NER**	**EX**	**PEX**	**SER**	**NER**
o3-mini	**5.88**	**13.73**	82.35	**49.02**	0.00	9.80	80.39	37.25
gpt-4o	3.92	11.76	**96.08**	45.10	1.96	7.84	94.12	45.10
gpt-4o-mini	1.96	5.88	86.27	43.14	1.96	5.88	88.24	33.33

Table 4.7: The EX, PEX, SER, and NER results when using SL_{MT} + syntax correction + empty output correction + self-validation on the training set.

Table 4.7 revealed that incorporating self-validation into the pipeline did not yield any performance improvements; in fact, both the SER and NER scores decreased across several instances. These results indicated that self-validation failed to improve the performance of the text-to-SQL pipeline, and so, it was considered ineffective.

4.5 Final Pipeline

In this research, various OpenAI LLMs and zero-shot prompting methods were systematically evaluated, refining the focus based on factors such as cost, time, accuracy, and usability. Although significant progress was made in identifying effective strategies, a single LLM and system message had not yet been selected for the final pipeline. To resolve this, the final training set results (seen in Table 4.6) for the pipeline indicated that the new **o3-mini** model, when paired with the OpenAI suggested system message (S_O), achieved the highest performance across the EX, PEX, and NER metrics. Consequently, the final pipeline employed both the S_O system message in combination with the **o3-mini** model. It is also worth mentioning that the **o3-mini** not only outperformed the next best LLM (**gpt-4o**) but was newer and cheaper. The final pipeline, in its entirety, is displayed in Figure 4.6.

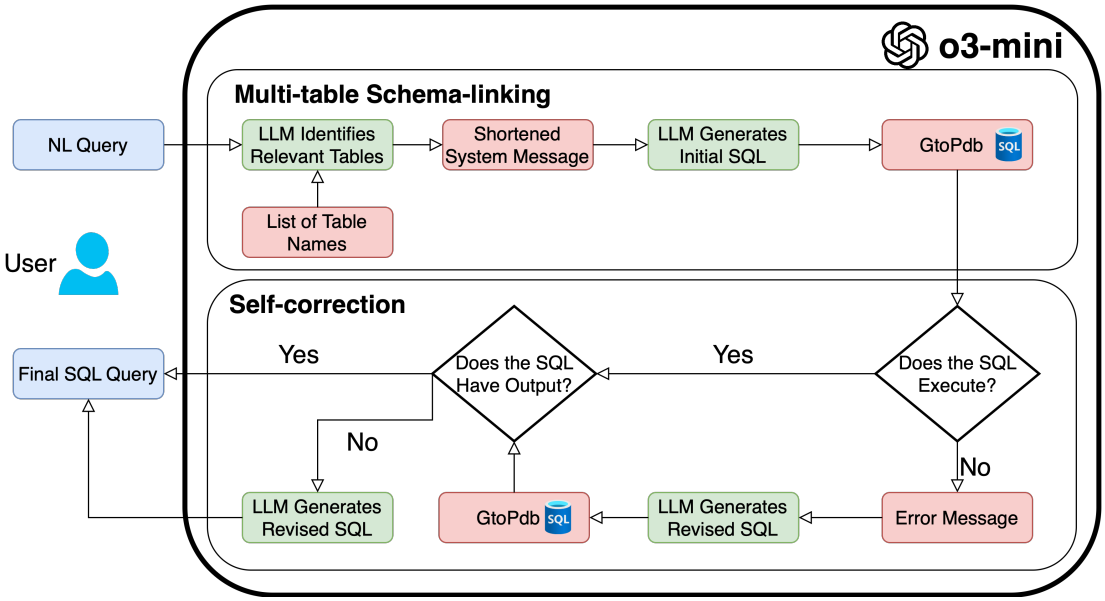


Figure 4.6: Visualisation of the final zero-shot pipeline discovered from exploring methods on the training set. Distinct colours differentiate key components: blue cells represent user steps, green cells indicate LLM-generated outputs, red cells correspond to GtoPdb-related processes, and diamond-shaped cells denote binary decision points.

Chapter 5

Results

To evaluate the constructed text-to-SQL pipeline, it was tested on the held-out set while invoking few-shot learning. As a reminder, few-shot learning is when the LLM is given only a small number of NL-SQL example pairs from the training set. The text-to-SQL pipeline was first evaluated under a zero-shot (no examples) scenario and then with the few-shot learning strategies outlined in Chapter 3. Therefore, these experiments address the following research questions:

- How important is few-shot learning compared to zero-shot learning?
- What is the most effective way of choosing examples for few-shot learning?

For each experiment, I chose to investigate one (1-shot), three (3-shot) and five (5-shot) learning examples from the training set. The only other selection being a special case of 4-shot learning, where a diverse set of 4 examples of 4 different difficulties was chosen. A visualisation of the process of combining few-shot learning within the pipeline is

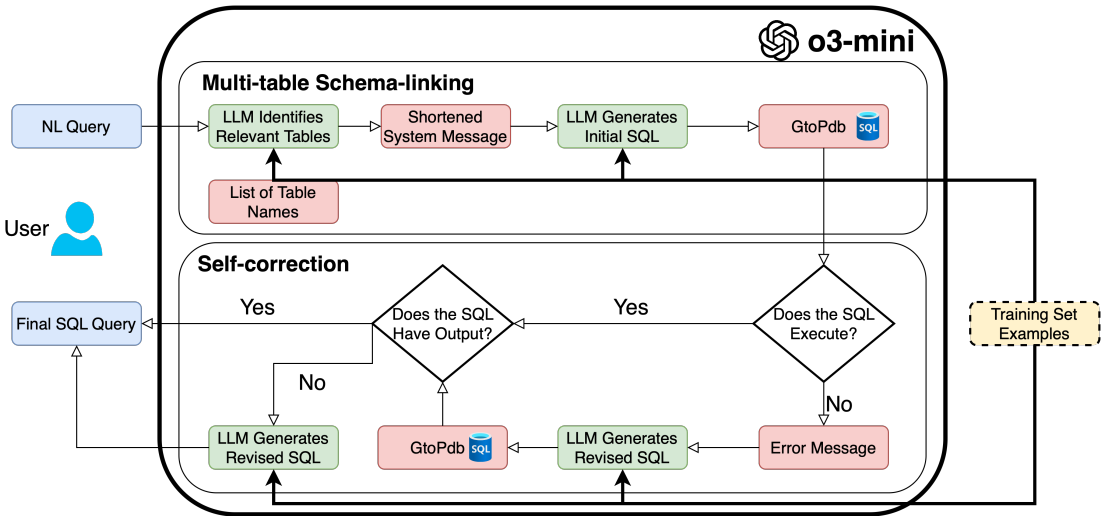


Figure 5.1: The final Text-to-SQL pipeline with few-shot learning. The LLM uses training set examples (highlighted in yellow) at each generation step to help identify relevant schema information and also produce/correct SQL queries.

shown in Figure 5.1. Unlike the zero-shot final pipeline (shown in Figure 4.6) this displays the LLM (**o3-mini**) receiving NL-SQL examples at each generation stage.

5.1 Establishing Baseline Performances

5.1.1 Zero-shot

To establish a performance baseline for the text-to-SQL pipeline, I began by evaluating the pipeline on the held-out set under a zero-shot scenario. In this scenario, the pipeline produced SQL queries in the same way as on the training set, that is, without being given any NL-SQL example pairs to aid its generation. This baseline evaluated the pipeline's pure ability to generate SQL on the held-out set and served as a reference point when comparing the impact of few-shot learning.

Looking at Table 5.1, the results for the zero-shot experiment on the held-out set show that the zero EM score was consistent with all previous experiments, where no generated query was syntactically identical to the provided gold standard. More notably, the EX, PEX, and NER scores were considerably lower than most experiments on the training set. These differences could suggest that the held-out set contained more challenging NL-SQL pairs than the training set. In contrast, the SER score was higher (93.33%) than the same pipeline on the training set (84.31%) which suggested that, although the SQL queries may not be very accurate and often lack an output, the pipeline was able to produce a higher rate of syntactically correct SQL queries on the held-out set. Nonetheless, these results established a solid zero-shot baseline for subsequent comparisons with few-shot learning approaches.

LLM	***k*-shot**	**EM**	**EX**	**PEX**	**SER**	**NER**
o3-mini	0-shot	0.00	0.00	3.33	93.33	40.00

Table 5.1: Results of all evaluation metrics when the final text-to-SQL pipeline was used on the held-out set under a zero-shot scenario.

5.1.2 Few-shot

Following the zero-shot evaluation, a few-shot baseline was established by providing the LLM with completely random in-context examples (from the training set) when answering each NL question in the held-out set. Within this scenario, the LLM used the fixed set of random NL-SQL pairs to aid its generation of SQL queries for the GtoPdb. As shown in Figure 5.1, the LLM received these examples at every single step of the pipeline. To ensure consistency, a hash code was generated for each NL question in the held-out set. This hash code was then used to randomly order the training examples, from which the top-k examples were selected. By doing so, the same set of NL-SQL examples was consistently used throughout the pipeline, avoiding any extra unintended few-shot examples that would have caused variation throughout each generation step. The results for all random few-shot example experiments are presented in Table 5.2.

LLM	k-shot	EM	EX	PEX	SER	NER
o3-mini	0-shot	0.00	0.00	3.33	93.33	40.00
o3-mini	1-shot	0.00	0.00	6.67	86.67	46.67
o3-mini	3-shot	0.00	**3.33**	**16.67**	96.67	53.33
o3-mini	5-shot	0.00	**3.33**	**16.67**	**100.00**	56.67
o3-mini	4-shot*	0.00	0.00	**16.67**	93.33	**60.00**

Table 5.2: Results of all evaluation metrics when the pipeline was tested on the held-out set when choosing k random examples. The 4-shot scenario is denoted with an asterisk (*) as it is a special case whereby the LLM received one random example from each difficulty.

The results revealed a consistent trend of improvement as the number of few-shot examples increased. Initially, the 1-shot example did not improve the scores much, with even a slight decrease in SER when considering the 0-shot pipeline. However, when moving from 1 to 3 examples, there was a significant increase across all metrics, some of which were slightly improved on in the 5-shot scenario. When considering the special 4-shot scenario, the LLM benefitted in terms of NER, possibly suggesting that a diverse selection of random examples was better than choosing completely random examples when it came to producing non-empty SQL.

5.2 Choosing Few-shot Examples

Building on the few-shot baseline, the two methods for selecting few-shot examples from the training set were explored. In the first experiment, the selection of NL-SQL examples was dictated by the similarity between an initially generated SQL query (based on the held-out set NL question) and the SQL queries contained within the training set. For the second experiment, NL-SQL examples were chosen from the training set based on the similarity of the held-out set NL question when compared to the NL questions from the training set. This investigation provided a suitable comparison for the effectiveness of different selection criteria against the random few-shot examples baseline, giving further understanding of how few-shot example choice influenced the performance of the text-to-SQL pipeline.

5.2.1 Similar SQL Examples

To implement this experiment, an initial SQL query was generated for each NL question in the held-out set using a preliminary pipeline stage that relied solely on SL_{MT} schema-linking. These initial SQL queries then served as references when the training set of NL-SQL examples was ranked for each NL question in the held-out set. As described in Chapter 3, the training set included a gold standard SQL query that answered its corresponding NL question, but in some cases, an alternative gold standard SQL query was also provided. When ranking the training set, the reference SQL query was compared to both the primary and the alternative SQL queries. If the alternative SQL query was more similar to the reference SQL query than the primary SQL query, then it

was used in the ranking, and the primary SQL was disregarded (or vice versa). This ensured that the LLM was not provided with the same NL example twice but with two different SQL queries. For each NL question in the held-out set, the pipeline was executed from the beginning with the top-k ranked NL-SQL examples (based on the initial SQL generated for that held-out NL question) provided to the LLM at each generation stage (visualised in Figure 5.1). Table 5.3 shows the results of this experiment.

LLM	k-shot	EM	EX	PEX	SER	NER
o3-mini	0-shot	0.00	0.00	3.33	93.33	40.00
o3-mini	1-shot	0.00	**6.67**	**20.00**	96.67	46.67
o3-mini	3-shot	0.00	3.33	13.33	96.67	63.33
o3-mini	5-shot	0.00	3.33	13.33	**100**	**80.00**

Table 5.3: Results when deploying pipeline on the held-out set under the similar SQL k-shot scenario.

The results showed that introducing just one similar SQL example was able to boost the PEX score significantly, increasing it from 3.33% (0-shot) to 20%. The EX score also increased from 0% to 6.67% which highlighted that one, well-chosen example was able to aid the LLM in generating more accurate SQL queries. Also, when compared to the random few-shot baseline (see Table 5.2), the 1-shot similar SQL example scenario outperformed all k-shot examples on PEX and EX scores.

As the number of few-shot examples increased from 1-shot to 5-shot, both the SER and NER scores improved. SER climbed from 96.67% (at 1-shot and 3-shot) to 100% at 5-shot, while NER increased from 46.67% to 80%. This indicated that more few-shot examples had a positive impact on producing robust SQL queries. However, while more examples benefitted SER and NER, there was a trade-off for accuracy. The EX and PEX metrics peaked at 1-shot and then declined when moving to 3-shot and 5-shot, with the PEX score falling below even the few-shot baseline performance (seen in Table 5.2) reported for its counterpart k-shot examples. This suggested that the initial SQL generation may have led to a poor ranking of the training set examples, likely due to the initially generated SQL being inaccurate. Thus, as k increased, the likelihood of selecting examples that were not relevant to the NL question also increased, possibly confusing the LLM into generating inaccurate SQL.

5.2.2 Similar NL Question Examples

For this experiment, the training set was ranked based solely on the similarity between all training set NL questions and the held-out set NL question. Just as in previous experiments, the top-k NL-SQL example pairs were then given to the LLM at each stage of the pipeline. The results for the similar NL experiment are displayed in Table 5.4.

Table 5.4 shows a non-zero instance of the most stringent metric EM, which was the first time this had occurred throughout all of the research. As a reminder, the EM metric reports whether the generated SQL query is syntactically identical to the gold

LLM	*k*-shot	EM	EX	PEX	SER	NER
o3-mini	0-shot	0.00	0.00	3.33	93.33	40.00
o3-mini	1-shot	0.00	6.67	16.67	**100**	66.67
o3-mini	3-shot	**3.33**	6.67	20.00	**100**	63.33
o3-mini	5-shot	0.00	**10.00**	**30.00**	**100**	**83.33**

Table 5.4: All evaluation metric results when the pipeline was tested on the held-out set under the similar NL question k-shot scenario.

standard SQL query. Therefore, this research officially recorded an instance of an LLM mimicking the exact syntactic style of a gold standard SQL query. Interestingly, this non-zero EM score was reported in the 3-shot scenario but not in the 5-shot scenario, an outcome that may be attributed to the black-box nature of the LLM.

A general trend of improvement was evident across all evaluation metrics as the number of *k* examples increased, the only exception being the aforementioned EM and a small dip in NER (from 66.67% to 63.33%) when *k* increased from 1 to 3. The SER score particularly benefitted from NL similar examples as it achieved a perfect 100% for all experiments, even in the single example (1-shot) scenario. Overall, the 5-shot configuration reported the highest EX (10%), PEX (30%) and NER (83.33%) scores in this study. Therefore, indicating that incorporating a broad number of NL-SQL examples based on NL question similarity gave substantial context, which in turn led to improved SQL query generation.

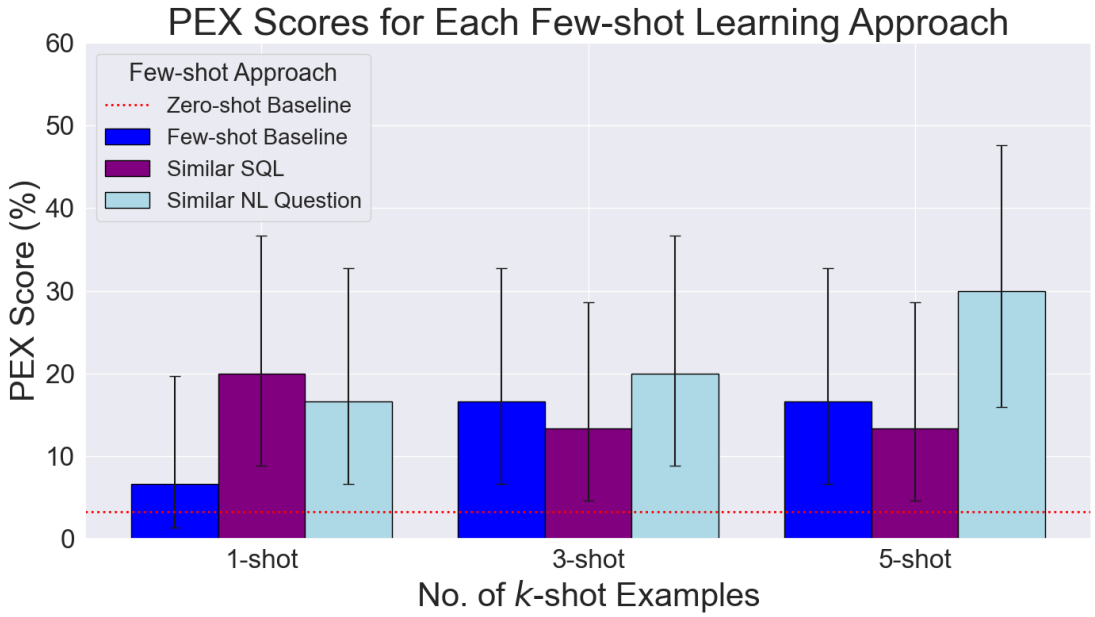
5.3 Summary

Through the comparison of a zero-shot baseline against various few-shot configurations, I discovered that few-shot learning was very important for generating SQL queries on the GtoPdb. In terms of accuracy, Figure 5.2a summarises the scores of the proposed PEX accuracy metric across all few-shot learning scenarios. Figure 5.2a shows that introducing few-shot examples from the training set, even when chosen at random, led to improvements in PEX. When few-shot examples were selected based on similarity, the benefits improved even more. The similar SQL examples experiment showed that a single well-chosen example could substantially boost PEX scores, although adding more examples resulted in a reduction in accuracy. Meanwhile, the similar NL question examples strategy consistently improved PEX as *k* increased, resulting in the 5-shot configuration yielding the highest PEX scores observed.

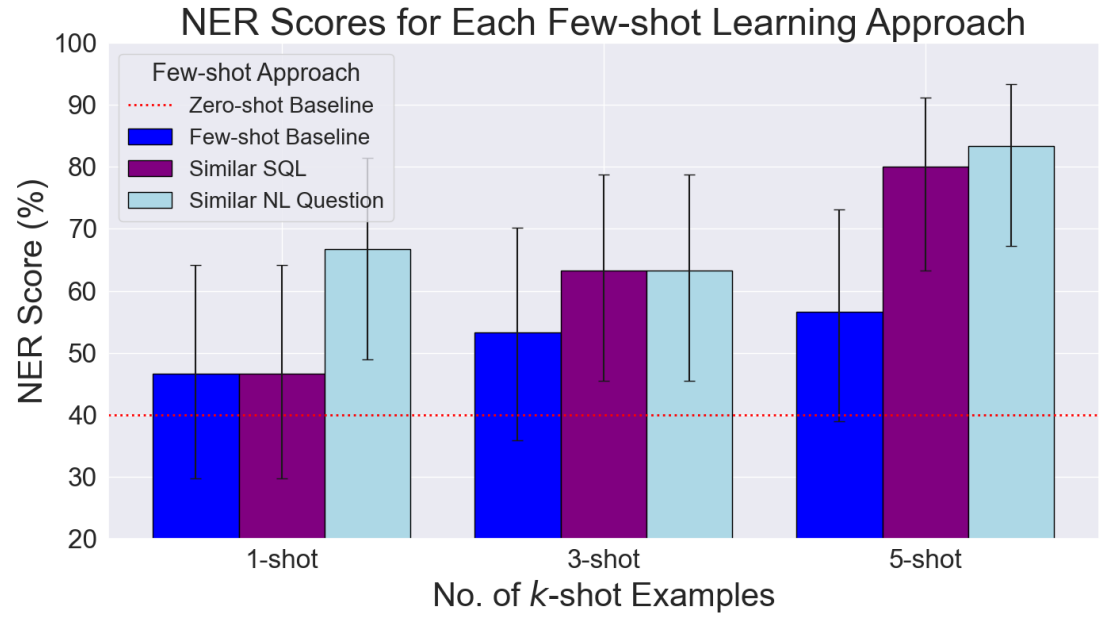
Figure 5.2b shows the NER scores for all few-shot experiments, whereby improvements in robustness were apparent when invoking few-shot learning. Interestingly, all few-shot configurations outperformed the zero-shot baseline, and both criteria for choosing few-shot examples outperformed the baseline few-shot NER scores. The improvements in NER also exhibited the same trend of improvement for each experiment, as that of PEX, which showed a consistent increase as *k* increased.

These results highlight the importance of both the quantity and the selection strategy

of few-shot examples, indicating that few-shot learning is extremely important for improving the quality of generated SQL. While increasing the number of examples generally enhanced robustness, the choice of few-shot examples played a major role in improving the accuracy. Consequently, this research found that choosing examples based on their NL question similarity was the best criterion.



(a) PEX scores.



(b) NER scores.

Figure 5.2: PEX and NER results across every few-shot learning experiment on the held-out set. Error bars indicate 95% confidence intervals computed using a Beta distribution (Jeffreys prior [54]), where for each proportion p out of n trials the posterior parameters $a = np + 0.5$ and $b = n(1-p) + 0.5$ were used to derive the interval.

Chapter 6

Conclusions

6.1 Discussion

My research successfully investigated the utilisation of Large Language Models (LLMs) for translating natural language (NL) questions into corresponding SQL queries for the IUPHAR/BPS Guide to Pharmacology database (GtoPdb). By focusing on two methodologies, zero-shot and few-shot, I was able to combine both to construct a text-to-SQL pipeline capable of producing SQL for the GtoPdb. Throughout the investigation, I managed to answer several research questions regarding the task of text-to-SQL for the GtoPdb.

6.1.1 Contributions

Firstly, I was able to construct my own, less stringent accuracy metric Partial Execution Accuracy (PEX) which gave a better insight when analysing explored methods. The construction of this metric came about as the common metrics, used on benchmark datasets [7, 6], proved to be less well suited for the GtoPdb, further suggesting that the dataset of GtoPdb NL-SQL pairs were possibly more complex than those included in benchmark datasets.

For the zero-shot methods, I explored several techniques on the training set allowing for the construction of a zero-shot text-to-SQL pipeline. I found that combining multi-table schema-linking, syntax correction, and empty output correction led to the construction of the most suitable zero-shot text-to-SQL pipeline. Also, I discovered that the proposed self-validation, where the LLM was able to view the output of its generated SQL query, did not yield any improvements in SQL generation. On top of this, I found that OpenAI's suggested schema representation combined with the **o3-mini** model was the most suitable to integrate within the text-to-SQL pipeline.

Considering the few-shot learning methods, I investigated several criteria for picking NL-SQL example pairs from the training set when answering each NL question in the held-out set. To do this, I combined the few-shot examples within each step of the text-to-SQL pipeline, ensuring the LLM had access to these examples throughout. When analysing the results, I discovered that the few-shot learning baseline significantly

outperformed the zero-shot pipeline, which suggested that few-shot learning had a significant impact on SQL generation. Moreover, I found that choosing five targetted NL-SQL example pairs from the training set based on their similarity to the held-out set's NL question contributed to the highest results across all evaluation metrics.

6.1.2 Related Work Comparison

A companion dissertation was conducted by Nikita Rameshkumar, who explored the same research topic [58]. Nikita's final approach shared some similarities with my methods but also introduced some alternatives. In particular, Nikita chose **gpt-4o** as her primary LLM and supplied it with a comprehensive database schema that included every table and column name. Similar to my work, Nikita employed self-correction, enabling the LLM to detect and fix SQL errors by regenerating SQL queries. In addition, Nikita provided the LLM with eight handcrafted rules to enhance SQL generation. These rules largely targeted thorough string matching; for example, when filtering for a single word (e.g. delta), the LLM was instructed to use `ILIKE '%delta%' OR ILIKE '%Delta%'` so that potential capitalisation differences would be covered. When evaluating her method on the held-out set, Nikita chose to use few-shot learning with all 51 NL-SQL example pairs from the training set. Along with these examples, Nikita also included the 'minimum number of columns' and 'notes for student' data. Table 6.1 presents Nikita's final results on the held-out set.

LLM	***k*-shot**	**EX**	**PEX**	**SER**	**NER**
gpt-4o	51-shot	13.33	43.33	100	90.00

Table 6.1: Nikita's final results on the held-out set.

Despite some methodological overlap, there were differences in both approach and performance. When comparing my pipeline's best results on the held-out set (seen in Table 5.4), it is apparent that Nikita's method outperformed my text-to-SQL pipeline on the accuracy metrics; Nikita reported an EX of 13.33% and a PEX of 43.33%, while my best approach attained an EX of 10% and PEX of 30%. Moreover, Nikita's final method reached a NER of 90%, exceeding the top NER of 83.33% observed in this research. These improvements are likely attributable to the inclusion of all table and column information at once and a full 51-shot learning approach (providing the LLM with the entire training set every time an SQL was generated).

However, it is important to acknowledge the trade-offs within Nikita's strategy. By presenting the LLM with all 51 NL-SQL example pairs and the full schema information, the token usage was considerably elevated. In contrast, my design was more cost-effective for research, relying on a concise subset of few-shot examples, a reduced database schema, and the cheaper **o3-mini** model. Ultimately, Nikita's results demonstrated the performance benefits that can be gained by saturating the LLM with as many examples as possible and the entire database schema. At the same time, my pipeline showed that strong outcomes are still attainable through the use of less tokens and a cheaper LLM.

6.2 Limitations and Future Work

In a practical scenario, security remains a serious concern. Since the pipeline executed some of the queries, to improve its final output, (e.g. self-correction) it is possible that a malicious user interacting with the pipeline (e.g. through a chatbot) could potentially exploit SQL injection vulnerabilities to execute harmful queries. Although the pipeline would execute SQL queries on a separate copy of the database, such attacks would still disrupt other users' experiences. Future work should investigate several sanitisation techniques [59], to stop harmful requests from being submitted.

This research was limited to a selection of LLMs from OpenAI. However, recent advancements have introduced new LLMs, such as DeepSeek [60], which in some scenarios have claimed to outperform the OpenAI models in both performance and affordability. Evaluating these emerging models for the task of text-to-SQL on the GtoPdb would allow for a more comprehensive exploration.

Another limitation which warrants acknowledgment is the relatively small dataset (81 examples) used for this research. Although I am extremely grateful to the NC-IUPHAR Database Executive Committee for curating the dataset on such short notice, its limited size restricted the confidence and generalisability of my findings. Future research should focus on expanding and diversifying the dataset to enable an improved evaluation of the text-to-SQL approaches. For example, I mentioned the use of a very basic version of Chain of Thought (CoT) prompting. This basic version simply asked the LLM to display its step-by-step reasoning; in improved CoT prompting, the LLM is given few-shot examples that display the thought process the LLM should mimic. Future work could focus on creating a dataset of step-by-step reasoning behind what schema information is relevant to each of the NL questions in the dataset. These examples could then be displayed to the LLM when it is choosing relevant schema information to try and improve its selection.

Next, the few-shot experiments focused on three scenarios (1-shot, 3-shot, and 5-shot), giving a concrete assessment of the impact of in-context examples. The similar NL question approach displayed a trend of improvement across all evaluation metrics as the number of examples (k) increased. As observed in Nikita's approach, the full 51-shot scenario was able to yield the best results across all evaluation metrics. Future work should explore the effects of incorporating more similar NL question examples to determine an optimal number needed to meet Nikita's results.

Lastly, another limitation arises from the need to reinitialise the system message for every NL question within this research. This was done to ensure the LLM retained no prior context, thus enforcing experimental control for each NL question. In a real-world scenario, an LLM could reuse a single system message (e.g. S_O) across multiple NL questions, negating the need to pay for a new system message for every NL question. In addition, this would allow the LLM to build contextual continuity to help influence its output. Future work should investigate the effect of retaining one system message in a conversational chatbot scenario to determine its effect and affordability in comparison to my text-to-SQL pipeline.

Bibliography

[1] Simon D Harding, Jane F Armstrong, Elena Faccenda, Christopher Southan, Stephen P H Alexander, Anthony P Davenport, Michael Spedding, and Jamie A Davies. The IUPHAR/BPS Guide to PHARMACOLOGY in 2024. *Nucleic Acids Research*, 52(D1):D1438–D1449, 10 2023.

[2] Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, 1974.

[3] Chris J Date. *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[4] Jim Melton and Alan R Simon. *Understanding the new SQL: a complete guide*. Morgan Kaufmann, 1993.

[5] Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, et al. A survey on text-to-sql parsing: Concepts, methods, and future directions. *arXiv preprint arXiv:2208.13629*, 2022.

[6] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*, 2018.

[7] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

[8] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[9] Elena Voita. NLP Course For You. https://lena-voita.github.io/nlp_course.html, Sep 2020. Accessed: 2024-10-08.

[10] Peter F Brown, Vincent J Della Pietra, Peter V Desouza, Jennifer C Lai, and Robert L Mercer. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–480, 1992.

[11] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on

evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45, 2024.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. 2017.

[13] Sebastian Raschka. Understanding and coding the self-attention mechanism of large language models from scratch, February 2023. Accessed: 2024-10-10.

[14] Andreas Stöffelbauer. How Large Language Models Work. *Data Science at Microsoft*, October 2023. Accessed: 2024-10-02.

[15] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, Nicholas Joseph, Saurav Kadavath, Jackson Kernion, Tom Conerly, Sheer El-Showk, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Tristan Hume, Scott Johnston, Shauna Kravec, Liane Lovitt, Neel Nanda, Catherine Olsson, Dario Amodei, Tom Brown, Jack Clark, Sam McCandlish, Chris Olah, Ben Mann, and Jared Kaplan. Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback, April 2022.

[16] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[17] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[18] OpenAI. Openai o3-mini: Pushing the frontier of cost-effective reasoning. https://openai.com/index/openai-o3-mini/, January 2025. Accessed: March 19, 2025.

[19] OpenAI. Openai o1 system card. https://cdn.openai.com/o1-system-card-20241205.pdf, December 2024. Accessed: April 2, 2025.

[20] OpenAI. Gpt-4o system card. https://cdn.openai.com/gpt-4o-system-card.pdf, August 2024. Accessed: March 19, 2025.

[21] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[22] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

[23] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[24] Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems*, 36, 2024.

[25] Shuaichen Chang and Eric Fosler-Lussier. How to prompt llms for text-to-sql: A study in zero-shot, single-domain, and cross-domain settings. *arXiv preprint arXiv:2305.11853*, 2023.

[26] Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies. *arXiv preprint arXiv:2305.12586*, 2023.

[27] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*, 2023.

[28] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)*, 53(3):1–34, 2020.

[29] Anthony J. Harmar, Rebecca A. Hills, Edward M. Rosser, Martin Jones, O. Peter Buneman, Donald R. Dunbar, Stuart D. Greenhill, Valerie A. Hale, Joanna L. Sharman, Tom I. Bonner, William A. Catterall, Anthony P. Davenport, Philippe Delagrange, Colin T. Dollery, Steven M. Foord, George A. Gutman, Vincent Laudet, Richard R. Neubig, Eliot H. Ohlstein, Richard W. Olsen, John Peters, Jean-Philippe Pin, Robert R. Ruffolo, David B. Searls, Mathew W. Wright, and Michael Spedding. IUPHAR-DB: the IUPHAR database of G protein-coupled receptors and ion channels. *Nucleic Acids Research*, 37:D680–D685, 10 2008.

[30] Joanna L Sharman, Chidochangu P Mpamhanga, Michael Spedding, Pierre Germain, Bart Staels, Catherine Dacquet, Vincent Laudet, Anthony J Harmar, and NC-IUPHAR. Iuphar-db: new receptors and tools for easy searching and visualization of pharmacological data. *Nucleic acids research*, 39(suppl_1):D534–D538, 2011.

[31] Adam J Pawson, Joanna L Sharman, Helen E Benson, Elena Faccenda, Stephen PH Alexander, O Peter Buneman, Anthony P Davenport, John C McGrath, John A Peters, Christopher Southan, et al. The iuphar/bps guide to pharmacology: an expert-driven knowledgebase of drug targets and their ligands. *Nucleic acids research*, 42(D1):D1098–D1106, 2014.

[32] Stephen PH Alexander, Alistair Mathie, and John A Peters. Guide to receptors and channels (grac). *British journal of pharmacology*, 164:S1–S2, 2011.

[33] Joanna L Sharman, Helen E Benson, Adam J Pawson, Veny Lukito, Chidochangu P Mpamhanga, Vincent Bombail, Anthony P Davenport, John A Peters, Michael Spedding, Anthony J Harmar, et al. Iuphar-db: updated database content and new features. *Nucleic acids research*, 41(D1):D1083–D1088, 2012.

[34] Anna Gaulton, Anne Hersey, Michał Nowotka, A. Patrícia Bento, Jon Chambers,

David Mendez, Prudence Mutowo, Francis Atkinson, Louisa J. Bellis, Elena Cibrián-Uhalte, Mark Davies, Nathan Dedman, Anneli Karlsson, María Paula Magariños, John P. Overington, George Papadatos, Ines Smit, and Andrew R. Leach. The chembl database in 2017. *Nucleic Acids Research*, 45(D1):D945–D954, 11 2016.

[35] David Mendez, Anna Gaulton, A Patrícia Bento, Jon Chambers, Marleen De Veij, Eloy Félix, María Paula Magariños, Juan F Mosquera, Prudence Mutowo, Michał Nowotka, María Gordillo-Marañón, Fiona Hunter, Laura Junco, Grace Mugumbate, Milagros Rodriguez-Lopez, Francis Atkinson, Nicolas Bosc, Chris J Radoux, Aldo Segura-Cabrera, Anne Hersey, and Andrew R Leach. Chembl: towards direct deposition of bioassay data. *Nucleic Acids Research*, 47(D1):D930–D940, 11 2018.

[36] Sunghwan Kim, Jie Chen, Tiejun Cheng, Asta Gindulyte, Jia He, Siqian He, Qingliang Li, Benjamin A Shoemaker, Paul A Thiessen, Bo Yu, Leonid Zaslavsky, Jian Zhang, and Evan E Bolton. Pubchem 2023 update. *Nucleic Acids Research*, 51(D1):D1373–D1380, 10 2022.

[37] John M Zelle and Raymond J Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pages 1050–1055, 1996.

[38] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql, 2023.

[39] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*, 2024.

[40] Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. The death of schema linking? text-to-sql in the age of well-reasoned language models. *arXiv preprint arXiv:2408.07702*, 2024.

[41] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*, 2024.

[42] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. Mac-sql: Multi-agent collaboration for text-to-sql. *arXiv preprint arXiv:2312.11242*, 2023.

[43] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[44] Eduardo R Nascimento, Grettel M Garcıa, Lucas Feijó, Wendy Z Victorio, Yenier T Izquierdo, Aiko R de Oliveira, Gustavo MC Coelho, Melissa Lemos, Robinson LS Garcia, Luiz AP Paes Leme, et al. Text-to-sql meets the real-world. In *26th Int. Conf. on Enterprise Info. Sys*, 2024.

[45] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[46] OpenAI. Retrieval augmented generation (rag) and semantic search for gpts. https://help.openai.com/en/articles/8868588-retrieval-augmented-generation-rag-and-semantic-search-for-gpts, January 2024. Accessed: March 28, 2025.

[47] Chunxi Guo, Zhiliang Tian, Jintao Tang, Pancheng Wang, Zhihua Wen, Kang Yang, and Ting Wang. A case-based reasoning framework for adaptive prompting in cross-domain text-to-sql. *CoRR*, 2023.

[48] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[50] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[51] Hans Peter Luhn. A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of research and development*, 1(4):309–317, 1957.

[52] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[53] Baoli Li and Liping Han. Distance weighted cosine similarity measure for text classification. In *Intelligent Data Engineering and Automated Learning–IDEAL 2013: 14th International Conference, IDEAL 2013, Hefei, China, October 20-23, 2013. Proceedings 14*, pages 611–618. Springer, 2013.

[54] Harold Jeffreys. An invariant form for the prior probability in estimation problems. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 186(1007):453–461, 1946.

[55] OpenAI. tiktoken: A fast bpe tokenizer for use with openai's models. https://github.com/openai/tiktoken, 2022. Accessed: March 31, 2025.

[56] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[57] Ronald Aylmer Fisher. Statistical methods for research workers. In *Breakthroughs in statistics: Methodology and distribution*, pages 66–70. Springer, 1970.

[58] Nikita Rameshkumar. Tuning large language models for text-to-sql on the iuphar/bps guide to pharmacology. Unpublished dissertation, April 2025.

[59] William GJ Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of sql injection attacks and countermeasures. In *ISSSE*, 2006.

[60] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Appendix A

Example SQL for Understanding Evaluation Metrics

A.1 Execution Accuracy (EX)

Listing A.1: Example SQL 1 to help with understanding EX.

```
select ligand_id
from ligand where ligand_id in
(select distinct ligand_id from interaction where object_id in
(select distinct object_id from gpcr)
and type ='Agonist' and affinity_units ='pKi' and affinity_median
   >10);
```

Listing A.2: Example SQL 2 to help with understanding EX.

```
select distinct ligand_id
from interaction where object_id in
(select distinct object_id from object where object_id in
(select distinct object_id from receptor2family rf
where family_id in
(select family_id from family where type in ('gpcr'))))
and type ='Agonist' and affinity_units ='pKi' and affinity_median
   >10;
```

A.2 Exact Matching (EM)

Listing A.3: Example SQL 1 to help with understanding EM.

```
select name from ligand where in_gtip = true;
```

Listing A.4: Example SQL 3 to help with understanding EM.

```
select name from ligand where in_gtip is true;
```

A.3 Partial Execution Accuracy (PEX)

Listing A.5: Example SQL 1 to help with understanding PEX.

```
select ligand_id from ligand where in_gtip = true;
```

Listing A.6: Example SQL 2 to help with understanding PEX.

```
select name, ligand_id from ligand where in_gtip = true;
```

Appendix B

SQL Used to Output System Messages

B.1 SQL For Text Representation

Listing B.1: SQL crafted to produce S_T system message.

```
SELECT t.table_name || ': ' || string_agg(c.column_name, ', ') AS
   table_info
FROM information_schema.tables t
JOIN information_schema.columns c
ON t.table_name = c.table_name AND t.table_schema = c.table_schema
WHERE t.table_schema = 'public'
AND t.table_type = 'BASE TABLE'
GROUP BY t.table_name
ORDER BY t.table_name;
```

B.2 SQL For Basic Representation

Listing B.2: SQL crafted to produce S_B system message.

```
SELECT 'Table: ' || t.table_name || ', columns: (' ||
string_agg(c.column_name, ', ') || ')'
AS table_info
FROM information_schema.tables t
JOIN information_schema.columns c
ON t.table_name = c.table_name AND t.table_schema = c.table_schema
WHERE t.table_schema = 'public'
AND t.table_type = 'BASE TABLE'
GROUP BY t.table_name
ORDER BY t.table_name;
```

Appendix C

Schema-Linking Prompts/Visualisations

C.1 Schema Linking Visual For Each System Message

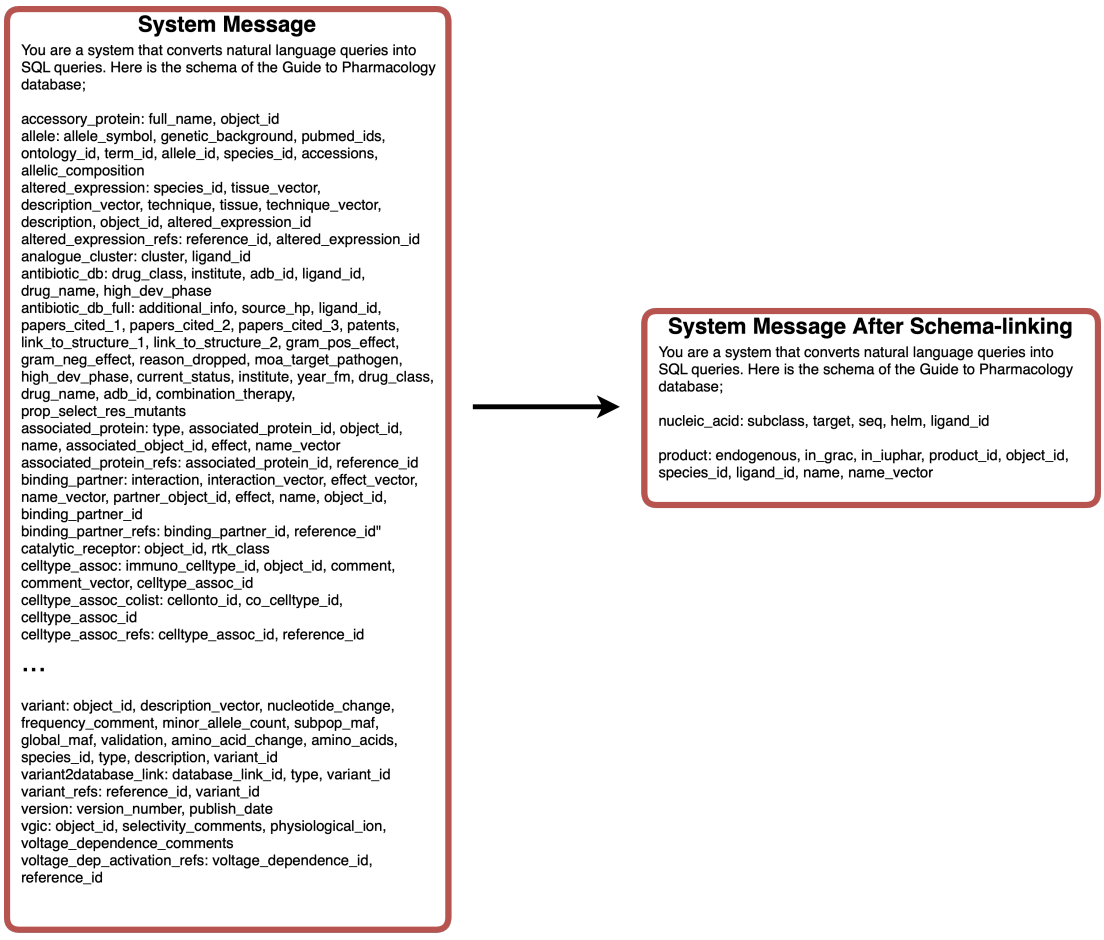


Figure C.1: Visualisation of the Text representation schema representation S_T before and after schema-linking.

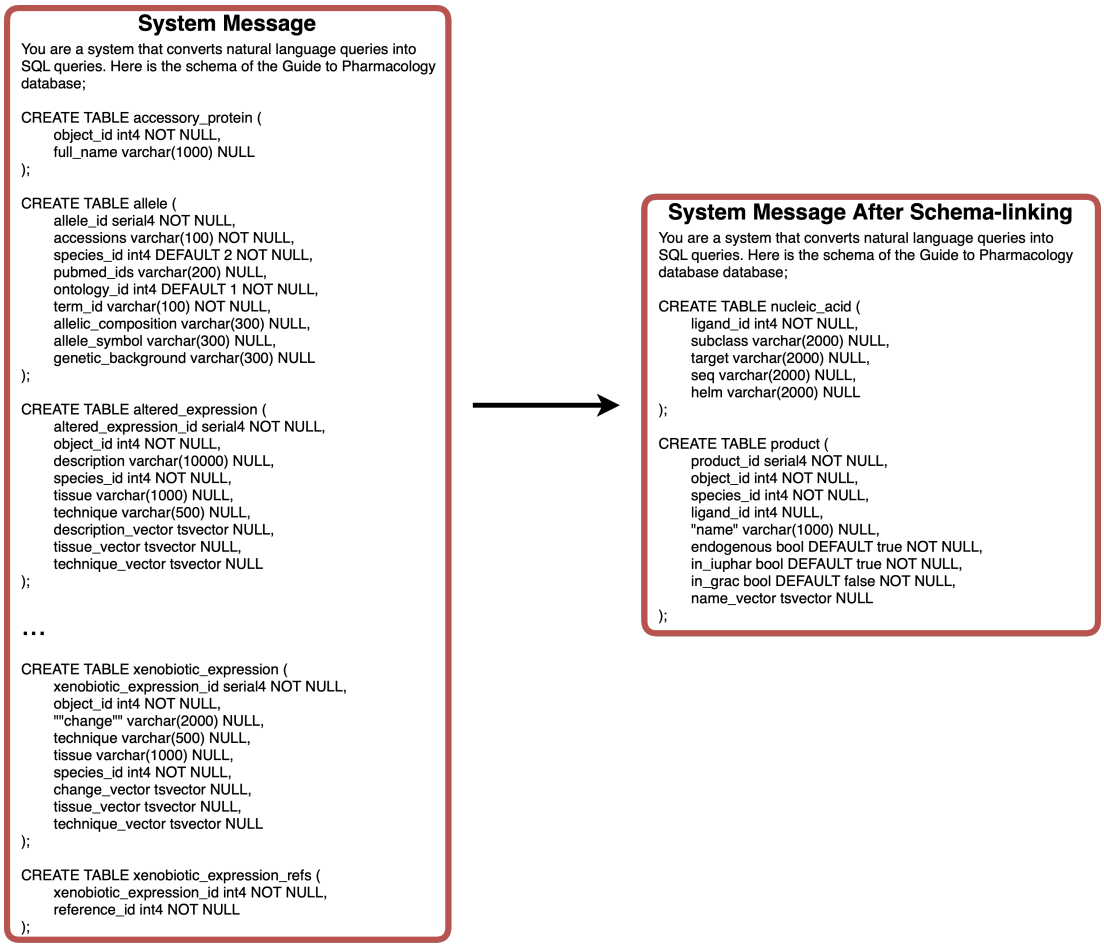


Figure C.2: Visualisation of the OpenAI suggested (no FK/PK information) schema representation S_{ON} before and after schema-linking.

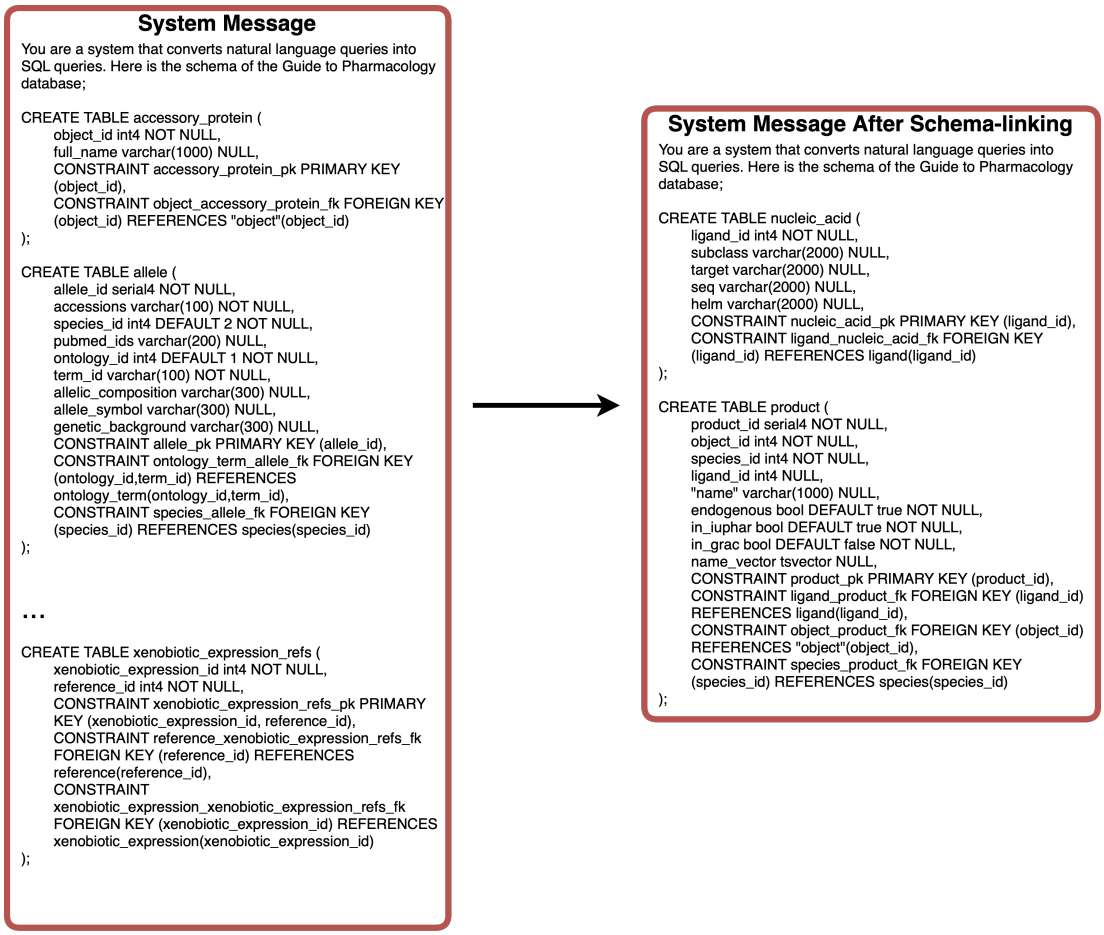


Figure C.3: Visualisation of the OpenAI suggested schema representation S_O before and after schema-linking.

C.2 Single-column Schema-Linking Prompt

You are a system that determines if a given column from an SQL database (The Guide to Pharmacology) is possibly relevant to a given natural language query. You will be provided with both a column and a natural language query in the format;

Column: column_name
Query: natural_language_query

Your output should be of the following json format;
{
"rationale": <str: the step-by-step reasoning behind the decision>,
"decision": <bool: True if the column is relevant, False otherwise>
}

Figure C.4: Single-column schema-linking (SL_C) prompt.

C.3 Multi-table Schema-Linking Prompt

You are a system that determines what tables from an SQL database (The Guide to Pharmacology) are possibly relevant to a natural language query. You will be provided with a list of tables and a natural language query in the format;

Tables: table_name_1, table_name_2, ...
Query: natural_language_query

Your output should be of the following json format;
{
"rationale": <str: the step by step reasoning behind the decision>,
"tables": <list[str]: relevant tables >
}

Figure C.5: First prompt in table-to-column schema-linking (SL_{TC}) and prompt used for multi-table schema linking (SL_{MT}).

C.4 Table-to-column Schema-Linking Prompt

You are a system that determines what columns from an SQL database (The Guide to Pharmacology) are possibly relevant to a natural language query. You will be provided with a table, the columns from said Table and a natural language query in the format;

<str: table_name>: column_name_1, column_name_2, ...
Query: natural_language_query

Your output should be of the following json format;
{
"rationale": <str: the step by step reason behind the decision>, <str: table_name>: <list[str]: relevant columns>
}

Figure C.6: Second prompt in table-to-column schema-linking (SL_{TC}).

C.5 System Messages in JSON format


```
[
    {
        "table": "accessory_protein",
        "columns": [
            "full_name",
            "object_id"
        ]
    },
    {
        "table": "allele",
        "columns": [
            "allele_symbol",
            "genetic_background",
            "pubmed_ids",
            "ontology_id",
            "term_id",
            "allele_id",
            "species_id",
            "accessions",
            "allelic_composition"
        ]
    },
    {
        "table": "altered_expression",
        "columns": [
            "species_id",
            "tissue_vector",
            "description_vector",
            "technique",
            "tissue",
            "technique_vector",
            "description",
            "object_id",
            "altered_expression_id"
        ]
    },
```


Figure C.7: JSON formatting used to schema-link S_T and S_B.

```
        "table": "gtip_process",
        "columns": {
            "gtip_process_id": {
                "definition": "int4 DEFAULT nextval('gtip_process_gtip_process_seq'::regclass) NOT NULL",
                "comment": null
            },
            "term": {
                "definition": "varchar(1000) NOT NULL",
                "comment": null
            },
            "definition": {
                "definition": "varchar(1500) NOT NULL",
                "comment": null
            },
            "last_modified": {
                "definition": "date NULL",
                "comment": null
            },
            "short_term": {
                "definition": "varchar(500) NULL,",
                "comment": "Short term for wbe display, may contain html"
            },
            "anchor": {
                "definition": "varchar(10) NULL,",
                "comment": "very short text to use as link anchor on webpages"
            },
            "term_vector": {
                "definition": "tsvector NULL",
                "comment": null
            },
            "definition_vector": {
                "definition": "tsvector NULL",
                "comment": null
            }
        },
        "primary_key": {
            "gtip_process_id": "CONSTRAINT gtip_process_pk PRIMARY KEY (gtip_process_id)"
        },
        "foreign_keys": {},
        "unique": {}
    },
```


Figure C.8: JSON formatting used to schema-link S_O and S_{ON}.

C.6 Token Count Reduction

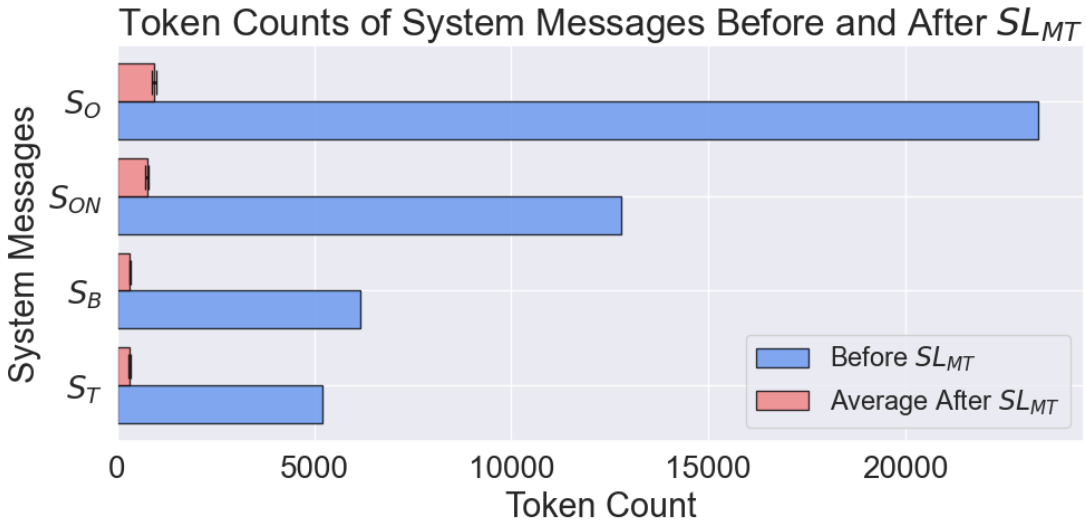


Figure C.9: Difference in token count before and after SL_{MT} on the training set. The error bars represent the 95% confidence intervals for the average schema-linked token counts. The error bars were computed by multiplying the standard error of the mean by the appropriate t-distribution critical value.

Appendix D

Self-Validation Workflow

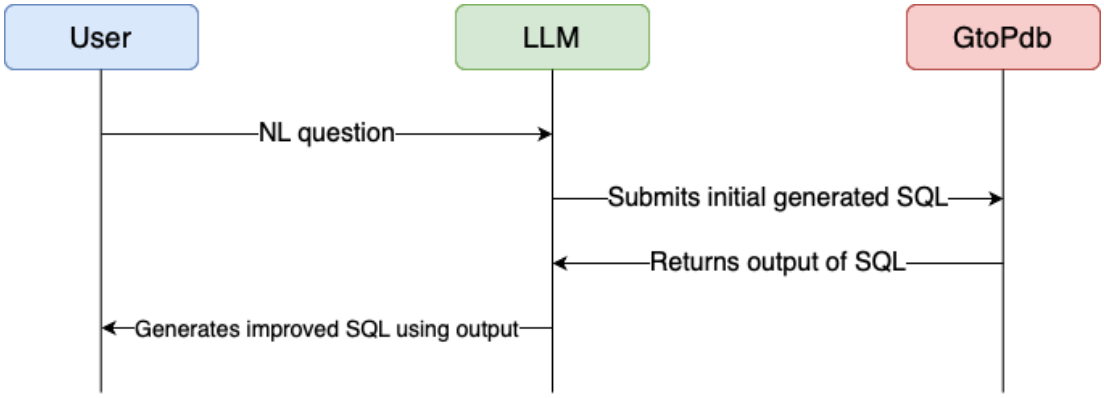


Figure D.1: Visualisation of the self-validation process when a user asks the LLM for an SQL query. The LLM's initial SQL query is ran on the GtoPdb and its output is used by the LLM to generate an improved response.