

**TRANSLATING NATURAL LANGUAGE
INTO QUALITATIVE SQL SEARCH QUERIES
WITH MULTI AGENT
LARGE LANGUAGE MODELS**

**A Thesis Submitted to
the Graduate School of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
MASTER OF SCIENCE
in Computer Engineering**

**by
Sime SÖNMEZ**

**June 2025
İZMİR**

We approve the thesis of **Simge SÖNMEZ**

Examining Committee Members:

Prof. Dr. Tolga AYAV

Department of Computer Engineering, Izmir Institute of Technology

Assoc. Prof. Dr. Selma TEKİR

Department of Computer Engineering, Izmir Institute of Technology

Prof. Dr. Yusuf Murat ERTEN

Department of Computer Engineering, Izmir University of Economics

24 June 2025

Prof. Dr. Tolga AYAV

Supervisor, Department of Computer Engineering,
İzmir Institute of Technology

Prof. Dr. Onur DEMİRÖRS

Head of the Department of
Computer Engineering

Prof. Dr. Mehtap EANES

Dean of the Graduate School of
Engineering and Science

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof. Tolga Ayav for his invaluable guidance and support throughout this research.

I am incredibly thankful to my dear friends, Ercan Gerçek and Onur Azbar, for their constant support and encouragement.

Finally, I extend my deepest appreciation to my fiancé Jan Trefil, whose unprecedented support have been greatly instrumental to my success.



ABSTRACT

TRANSLATING NATURAL LANGUAGE INTO QUALITATIVE SQL SEARCH QUERIES WITH MULTI AGENT LARGE LANGUAGE MODELS

Natural Language to SQL (NL2SQL), also known as text-to-SQL, refers to the task of converting natural language questions into equivalent SQL queries. This process traditionally requires specialized SQL knowledge and can be time-consuming when performed manually by data scientists. To address this challenge, researchers have investigated methods for enabling natural language communication with databases, aiming to establish an effective interface that serves as a bridge between users and data systems.

Automating NL2SQL tasks has the potential to accelerate data-driven decision-making by streamlining the data retrieval process. Over around thirty years, approaches have evolved from rule-based grammar systems to database-aware sequence-to-SQL models. However, with the emergence of large language models (LLMs), progress in this field has significantly accelerated, achieving breakthroughs that were previously unattainable.

In this study, NL2SQL task is attempted to be addressed by leveraging the joint reasoning power of mid-sized context-aware open language models, rather than relying on large-scale models such as GPT. The objective is to explore whether competitive performance can be achieved using models that require significantly fewer computational resources.

Three mid-sized open-source models — Gemma3-IT (Google), Qwen2.5 Instruct (Alibaba) and Mistral Nemo (Mistral AI) — are utilized as the base models in this research, and it achieved an execution accuracy of 81.3% on the Spider development evaluation set by harnessing their collaborative work, slightly lower than two baseline studies, by just 0.6% compared to C3-SQL and 1.5% compared to DIN-SQL, indicating the effectiveness of this approach.

ÖZET

ÇOKLU ETMEN BÜYÜK DİL MODELLERİ İLE DOĞAL DİLİN NİTELİKSEL SQL ARAMA SORGULARINA DÖNÜŞTÜRÜLMESİ

Doğal dilden SQL'e dönüşüm (NL2SQL), kullanıcı tarafından doğal dilde ifade edilen sorguların, aynı anlamı taşıyan yapısal SQL ifadelerine çevrilmesi sürecidir. Bu işlem, uzmanlık seviyesinde SQL bilgisi gerektirir, veri bilimciler ya da iş analistleri tarafından manuel olarak gerçekleştirildiğinde zaman alıcı olabilir. Bu süreci hızlandırmak ve kolaylaştırmak için araştırmacılar, veritabanlarıyla doğal dil üzerinden iletişimi mümkün kılacak yöntemler üzerine çalışmış; kullanıcılarla veri sistemleri arasında etkili bir arayüz oluşturmayı hedeflemişlerdir.

Yaklaşık otuz yıllık bir süreçte, bu alandaki yaklaşımlar, kural tabanlı gramer sistemlerinden, veritabanı bilgisinin kodlandığı (database-aware encoding) seq2seq modellere evrilmiştir. Ancak, büyük dil modellerinin (LLM'ler) ortaya çıkmasıyla birlikte bu alanda kaydedilen ilerleme önemli ölçüde hızlanmış ve daha önce ulaşılamayan bir seviyeye erişmiştir.

Bu çalışmada, NL2SQL görevi, GPT gibi büyük ölçekli modellere güvenmek yerine, orta ölçekli açık kaynak dil modellerinin etkin akıl yürütme gücünden ve işbirliğinden yararlanılarak ele alınmaya çalışılmıştır. Çalışmanın amacı, çok daha az kaynak gerektiren modellerle rekabetçi bir performansın elde edilip edilemeyeceğini araştırmaktır.

Bu araştırma kapsamında, üç orta ölçekli açık kaynak model — Gemma3-IT (Google), Qwen2.5 Instruct (Alibaba) ve Mistral Nemo (Mistral AI) — kullanıldı. Hem modellerin iş birliğinden ve hem de bağlamsal veritabanı içeriğinden yararlanılarak, Spider geliştirme değerlendirme setinde %81,3 yürütme doğruluğuna ulaşıldı.

Bu sonuç, her bir modelin bireysel performansının üzerinde olup, C3-SQL'e kıyasla yalnızca %0,6 ve DIN-SQL'e kıyasla %1,5 daha düşüktür. Sonuçlar, çalışmanın ortaya koyduğu yaklaşımın etkinliğini göstermektedir.

TABLE OF CONTENTS

LIST OF FIGURES	viii
LIST OF TABLES.....	x
CHAPTER 1. INTRODUCTION	1
1.1. Motivation.....	1
1.2. Contribution	3
1.3. Outline.....	4
CHAPTER 2. LITERATURE REVIEW	6
2.1. NL2SQL Tasks	6
2.1.1. Evolution of NL2SQL Tasks	8
2.1.2. LLM-Based Methods	11
2.1.3. Challenges of NL2SQL Tasks	13
2.1.4. Related Work	14
2.2. Multi-Agent Systems	19
2.2.1. Frameworks	21
2.2.1.1. LangChain	21
2.2.1.2. Autogen.....	22
2.2.2. Applications in Text-to-SQL Tasks	24
CHAPTER 3. METHODOLOGY	27
3.1. Problem Definition.....	27
3.2. Hardware.....	27
3.3. Software	28
3.3.1. Models	29
3.3.2. Ollama.....	34
3.3.3. Microsoft Autogen.....	34
3.4. Evaluation Benchmark and Metrics.....	37
3.5. Baselines	41
CHAPTER 4. EXPERIMENTS.....	43

4.1. Single-Agent Experiments	43
4.1.1. Error Analysis in Single-Agent Query Generation	44
4.1.2. Model Behaviour Across Spider Benchmark	50
4.2. Multi-Agent Experiments	51
4.2.1. Task Decomposition and Customized Prompting	54
4.2.2. Context Awareness	56
4.2.3. Self-Correction	59
4.2.4. Model Bias	61
4.3. Baselines Comparison	62
CHAPTER 5. CONCLUSION	64
REFERENCES	65

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
Figure 1.1. AI powered text-to-sql query generation.....	2
Figure 2.1. (a) Linguistic parse tree (b) Mapping strategy for the nodes in parse tree (Source: Li and Jagadish 2014)	9
Figure 2.2. CFG for SELECT and FROM clauses (Source: Mahmud et al. 2015)	10
Figure 2.3. The evolution of NL2SQL solutions (Source: Liu et al. 2024).....	10
Figure 2.4. Ambiguous NL challenges (Source: Liu et al. 2024).....	13
Figure 2.5. DIN-SQL modules organization (Source: Pourreza and Rafiei 2023).....	15
Figure 2.6. DAIL prompt organization (Source: Gao et al. 2023).....	16
Figure 2.7. Search tree formulation in Alpha-SQL (Source: Li et al. 2025)	17
Figure 2.8. Autogen framework conversable agents (Source: Wu et al. 2023).....	21
Figure 2.9. Work principle of retriever (Source: Topsakal and Akinci 2023).....	22
Figure 2.10. Model asking for human input	23
Figure 2.11. Output of the agent backed with tool	24
Figure 2.12. Overview of QCMA-SQL framework (Source: Shao et al. 2025).....	25
Figure 3.1. Ollama client instantiation.....	29
Figure 3.2. Qwen2.5 14B Instruct & Gemma3-IT 12B scores on various bench- marks.....	31
Figure 3.3. Gemma3-PT 12B & Mistral Nemo Instruct scores on various bench- marks.....	32
Figure 3.4. Single agent text-to-sql experimental setup	33
Figure 3.5. Multi agent text-to-sql experimental setup.....	37
Figure 3.6. WikiSQL & Spider & BIRD benchmark comparison.....	38
Figure 3.7. An example of false negative result	41
Figure 3.8. C3-SQL & DIN-SQL Spider benchmark execution accuracy results	42
Figure 4.1. Execution accuracy results - Mistral Nemo, Qwen2.5 Instruct 14B and Gemma3 IT 12B	44
Figure 4.2. Accuracy comparison of base models across Spider-Dev questions	51

<u>Figure</u>	<u>Page</u>
Figure 4.3. Multi agent system performance comparing to the base models	52
Figure 4.4. Accuracy comparison of base models comparing to multi-agent system across Spider-Dev questions	53
Figure 4.5. Context aware multi-agent system comparing to DIN-SQL and C3-SQL	63



LIST OF TABLES

<u>Table</u>	<u>Page</u>
Table 3.1. Base model performance comparison on different benchmarks.....	30
Table 3.2. Gemma3-PT 12B & Mistral Nemo Instruct scores	31
Table 3.3. NL2SQL benchmarks with properties	38
Table 3.4. DIN-SQL vs C3-SQL execution accuracy comparison.....	42
Table 4.1. Execution accuracy for base models.....	44
Table 4.2. Number of execution errors for base models.....	45
Table 4.3. Execution accuracy of context aware multi-agent system comparing to base models.....	52
Table 4.4. Number of execution errors comparing base models with context aware multi-agent system	59
Table 4.5. Context aware multi-agent system vs DIN-SQL vs C3-SQL execution accuracy	63

CHAPTER 1

INTRODUCTION

1.1. Motivation

Large Language Models (LLMs) have recently turned into the most promising ML models performing on numerous different tasks with the utmost success. It's widely acknowledged that LLMs excel in well-defined tasks including text classification, content generation, question answering especially in knowledge-demanding domains. Techniques such as prompt engineering, fine-tuning, retrieval-augmented generation (RAG), hyper-parameter tuning help the models perform even better by the means of increasing accuracy and efficiency in clear-cut tasks.

Many organizations are now incorporating LLM-based solutions into their services and products. A key benefit is streamlined operations; for example, Salesforce has introduced Einstein Automate, an LLM-powered workflow automation tool that helps administrators simplify sales processes and boost seller productivity (Salesforce 2020). Beyond operational efficiency, LLMs are also proving valuable for user profiling and enhancing recommender systems. The PURE framework (Bang and Song 2025), for example, leverages LLMs to continuously extract and summarize user data, improving the accuracy and relevance of recommendations.

The exponential growth of data necessitates more efficient storage and processing solutions and LLMs offer a powerful response, enabling rapid and accurate data analysis. Hence, data platforms are also increasingly leveraging models to generate valuable data insights. Databricks BI (Business Intelligence) has incorporated LLMs built-in AI capabilities, providing customers with the ability to analyse data stored on the platform. The *ai_query()* function, for instance, allows users to extract information and summarize content leveraging different LLMs (Databricks 2024). Power BI has introduced Copilot, another LLM-based solution that allows users to generate data visualizations and reports using natural language prompts. For example, a data analyst could use a prompt like

“Generate a summary focused on beaches in Antalya” to create a narrative. A key aspect of this narrative creation process is translating natural language to SQL queries (NL2SQL), which will be the focus of this study.

NL2SQL tasks involve translating natural language prompts and questions into Structured Query Language (SQL), as the name suggests. The history of this field dates back to the 1970s, when rule-based methods and manually created grammars were extensively studied in text-to-SQL tasks. Recently, the growing interest in LLMs has elevated it to a significant area of research. As a result, researchers have been actively developing and evaluating a variety of techniques using established benchmarks and datasets, a discussion of which will be presented in Chapter 2.

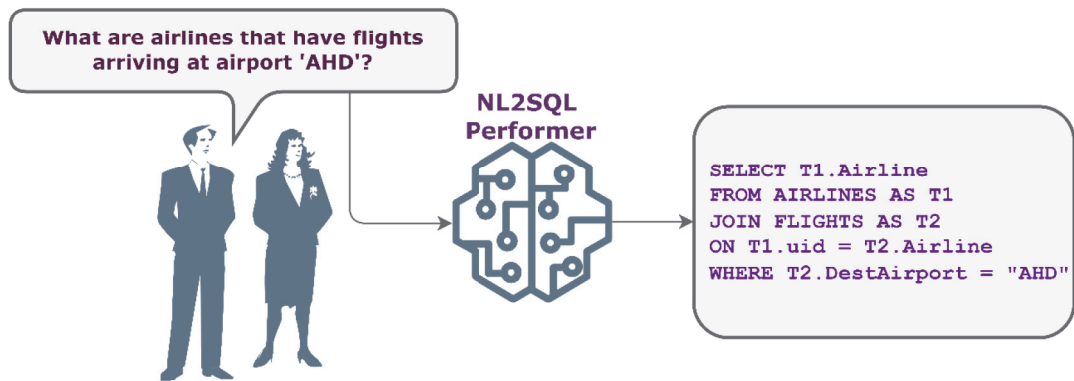


Figure 1.1. AI powered text-to-sql query generation

Several researchers have explored fine-tuning open-source LLMs to address the NL2SQL task. For example, Defog introduced SQLCoder, a highly effective fine-tuned model trained on 20,000 human-created questions across ten diverse database schemas to translate natural language into SQL queries. Similarly, Natural-SQL by ChatDB, another high-performing fine-tuned model, excels at understanding complex questions and surpasses many models of similar size. However, the lack of sufficient, high-quality datasets hinders further research in this area.

Other researchers have explored various prompting techniques to improve performance and accuracy. The most effective results were achieved using prompts that included a natural language question, the relevant database schema, and a few examples guiding model to produce responses aligned with the desired output.

Given that there are few studies have been conducted, such as QCMA-SQL and MAC-SQL, in order to explore NL2SQL performance in multi-agent systems, there are still more to explore.

In this study, primary motivation was to investigate the potential for enhancing NL2SQL task performance through the collaborative efforts of multiple, smaller-scale agents. Consequently, this research focuses on developing a customized solution for NL2SQL tasks utilizing intelligent multi-agent systems. This approach enables the decomposition of complex natural language questions and facilitates the provision of contextual data to agents, thereby improving their understanding and accuracy. To this end, Microsoft Autogen, a relatively new multi-agent framework offering significant opportunities for exploration, was chosen for use.

Regarding evaluation, it presents significant challenges, stemming from limited data availability, the wide range of existing dialects and domains, and security concerns surrounding the release of original datasets. Thus, evaluation of this study primarily relies on human-annotated synthetic datasets. The Spider Benchmark, a widely recognized and impactful cross-domain SQLite benchmark developed by students at Yale University, was leveraged to assess our experimental results.

The focus of this study was on achieving high execution accuracy rather than generating the most performant query. Therefore, the execution accuracy metric of the predicted queries was evaluated, rather than exact match or component match, and these results were compared to those of baseline studies and the base models.

1.2. Contribution

Given that fine-tuning becomes more costly and resource intensive as LLMs become bigger in size, it poses a promising alternative to leverage growing power of reasoning capabilities in text-to-sql tasks. For the purposes of this study, an endeavour has been made to make SQL construction even cheaper by using small and mid-size models in collaboration.

The focus has been placed on utilizing the collaboration of multiple models aided by contextual data possibly relevant to the question. Microsoft Autogen is leveraged to

demonstrate whether the collaboration of small-sized LLMs outperforms larger and customized models.

The main aimed contribution here is:

- A practical and economic solution by leveraging reasoning power of small-medium LLMs in collaboration,
- An evaluation of mid-size open-source models in NL2SQL tasks,
- An exploration of the capabilities of Microsoft Autogen multi-agent framework in NL2SQL tasks.

1.3. Outline

Because the objective of this study is to introduce a cost-effective solution for NL2SQL tasks, Chapter 2.1 provides a comprehensive review of existing approaches identified in prior researches.

Initially, a foundational understanding of NL2SQL, its significance and the potential benefits of developing an efficient solution is established. Subsequently, Chapter 2.1.1 details the evolution of NL2SQL tasks, tracing the progression from initial rule-based methods to the more complex LLM-backed solutions currently in use.

Chapter 2.1.2 provides a more detailed examination of LLM-based methods in the aspects of various techniques employed including prompting techniques, fine-tuning strategies and task decomposition approaches explored in prior studies. Remaining questions and discussion points are explored in the same chapter, alongside a review of recent works that have achieved high rankings on the Spider Leaderboard.

As described above, this approach leverages the Microsoft Autogen multi-agent framework to enhance performance in NL2SQL tasks. To this end, prior researches on intelligent multi-agent systems and two prominent frameworks are reviewed: LangChain and Autogen. Furthermore, two different studies whose approach is to launch multi-agent systems in NL2SQL tasks are explored, QCMA-SQL and MAC-SQL.

In Chapter 3, the software and hardware requirements, models, frameworks and technologies employed for the experimental setup are explained. This chapter also provides an overview of two key open-source technologies – Ollama and Microsoft AutoGen

– and briefly explains how they are employed throughout the experiments. A detailed explanation of the selected open models is also given, outlining the rationale behind their choice and describing how they are utilized. Furthermore, the NL2SQL benchmarks and evaluation metrics employed in the assessment are analysed.

Chapter 4 evaluates the NL2SQL system through single-agent and multi-agent setups. It first examines how individual models perform, highlighting common errors and limitations across the Spider benchmark. The focus then shifts to multi-agent strategies, showing how task decomposition, contextual awareness and self-correction improve accuracy. A comparison with both baselines and base models confirms the advantages of the proposed approach.

Finally, Chapter 5 summarizes the main findings, emphasizing the improvements brought by multi-model collaboration backed with contextual data.

CHAPTER 2

LITERATURE REVIEW

2.1. NL2SQL Tasks

Natural Language to SQL, also known as text-to-sql, is the process of translating natural language queries into SQL code which is a fundamental task that involves generating SQL queries based on the data needs of companies. This responsibility typically falls on data analysts and data engineers, who use SQL to extract, transform and load data within databases or other data systems in order to answer business questions. Meanwhile, Database Administrators focus on optimizing query performance by creating views, triggers, indexes, and rewriting inefficient queries to ensure optimal database performance. This effort requires a careful understanding and interpretation of the NL question at first, it is of high importance to grasp the user's intent and come up with several key elements which are entities, attributes, contexts and conditions.

Based on the knowledge of database schema, key elements are supposed to get mapped to the relevant tables and columns, which is known as schema linking. All the ambiguities must be clarified and figured out so as to generate the query with the utmost accuracy and the best performance. With increasingly complex queries including multiple JOINS, aggregations and nested queries and the exponentially growing amount of data, the process of generating SQL from natural language requests will require more time and effort to ensure accuracy and efficiency.

Automating the NL2SQL process can benefit data analysts by significantly reducing the time spent on writing complex queries from scratch. By helping even non-skilled users with construction the best quality SQL queries, the goal is to reduce the time spent in writing queries and lead to spending more time on the higher level analysis, strategic planning and deriving insights from the data instead of tedious SQL development tasks.

The most practical and common use case is definitely the Business Intelligence (BI) Tools such as Tableau. BI tools are commonly employed to read, analyse, and

visualize data stored in data warehouses. In the paper available at (Lian et al. 2024), the concept of Natural Language to Business Intelligence (NL2BI) has been introduced as a prevalent practical scenario. To address this challenge, ChatBI has been proposed as a solution. Given the existing tables containing a huge number of columns in BI scenarios and LLMs token limitation makes the schema linking more challenging, they utilized existing view technology to perform schema linking and integrated it with a small size language model to manage.

Additionally, SQL continues to be a popular choice for developers, consistently ranking among the most desired and admired programming languages. According to a Stack Overflow survey, it hits third (37.4% desired, 67.4% admired), trailing only Python and JavaScript. This strong demand of developers establishes another important use case (StackOverflow 2024).

Research on NL2SQL tasks has grown substantially over the years upon this increasing importance. The journey started with rule-based methods in 70s, followed by neural models. With the advent of LLMs, these researches have finally geared up and started to bear fruits. DAIL-SQL + GPT4 with self-consistency has reached up to 86.6% accuracy on Spider v1.0 Leaderboard and got the first rank among all other studies (Gao et al. 2023).

Fine-tuning Large Language Models (LLMs) has historically been the dominant approach for tackling the NL2SQL challenge. Examples of such models include SQLCoder, DuckDB-NSQL and SQL-Llama, all of which were fine-tuned on natural language-SQL pairs. These models are known to be heavily reliant on the datasets used for their training, and their performance often degrades significantly when encountering some unseen database schemas. Furthermore, the limited availability of training data, coupled with the substantial computational power and memory resources required for fine-tuning, poses a significant challenge to developing robust and powerful models.

On the other hand, in-context learning enables Large Language Models (LLMs) to generate SQL queries without tuning, requiring only a database schema and a natural language question as input. Including a handful of in-domain examples within the prompt, performance can be improved. Similarly, many studies utilize textualized database schema, natural language question and foreign key details to construct prompts. Others choose to incorporate the schema using Data Definition Language (DDL) statement formats while omitting foreign key information (Gao et al. 2023).

While few-shot learning often improves the accuracy of generated queries by

providing illustrative examples, an inverse U-shaped relationship emerges when considering prompt length. Initially, adding examples boosts performance, but beyond a certain point, increasing the prompt length can actually diminish query accuracy, likely due to factors like increased complexity and potential for distraction (Chang and Fosler-Lussier 2023). Therefore, it is of the utmost importance to maintain an optimal length for prompts to get the best performance.

There is still an ongoing debate how the best prompt can be crafted in order to get the best performance in NL2SQL tasks. Next sub-chapter of literature review, the evolution of NL2SQL tasks and the high benefits of LLM methods will be focused.

2.1.1. Evolution of NL2SQL Tasks

The field of NL2SQL has a rich history, centred on translating natural language questions into precise SQL queries. Historically, the complexity of grammar and logical reasoning limited its practical application, primarily confining it to academic research and specialized databases with tailored grammars. However, recent advancements in LLM technologies are expanding its potential. In this chapter, the evolution of NL2SQL tasks and their pivotal points will be explored:

Rule Based Methods

Early approaches often focused on rule-based methods, where researchers developed rules and grammars to interpret natural language into the database schema. They were mainly based on manually crafted grammars utilizing the syntactic nature of the sentences used when questioning the database.

The roots of rule-based approach can be traced back to 1973 with the LUNAR project (Woods 1973). This pioneering effort aimed to tackle the challenges of man-machine communication by attempting to adopt the machine to *natural English language – not to an English-like querying language* –.

Launched as a prototype system at a time when semantic search was still an emerging field, LUNAR was developed in order to enable lunar geologists to analyse the data on a lunar rock and soil composition collected as an Apollo moon mission by NASA. So, it was based on a database file provided by NASA, 13,000 entry table containing chemical analysis data of the Apollo 11 samples.

Natural Language Interface for Relational Databases (NaLIR), introduced in 2014, provides another significant application of rule-based methods (Li and Jagadish, 2014). NaLIR proposed an intermediary linguistic parse tree, called as *QueryTree*, to facilitate this translation process.

To construct a *QueryTree*, researchers defined a context free grammar (CFG) for valid query syntax and utilized the Stanford Parser. Figure 2.1 illustrates the parse tree of the query: “Return all the authors who have more papers than H. V. Jagadish in VLDB after 2005.”

They keep the user in the loop for allowing them to review and confirm the constructed *QueryTree* in order to dispose of all ambiguities in the user expression. Lexical ambiguities pose an important challenge, many experts have endeavoured to address as will be discussed in the upcoming sub-chapters.

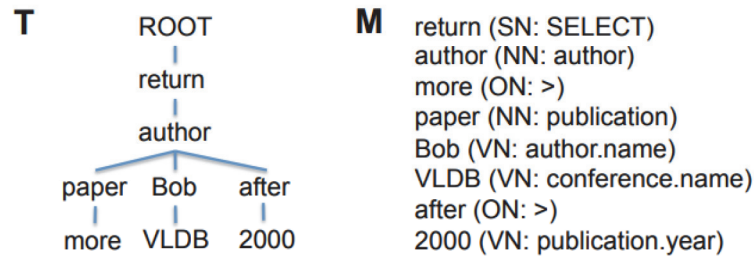


Figure 2.1. (a) Linguistic parse tree (b) Mapping strategy for the nodes in parse tree (Source: Li and Jagadish 2014)

Another example of this approach, Natural Language Interface to Databases (NLIDBs) published in 2015, introduced by (Mahmud et al. 2015), leveraging CFGs similar to NaLIR. A part of this grammar which is for SELECT and FROM can be seen in Figure 2.2.

As long and complex prompts remained a challenge, the system was designed in a way to filter out irrelevant terms to simplify processing and reduce token usage. This research involved developing this grammar based on a sample database Alumni — where the actual data of alumnae of a university were stored — and evaluated it on the same database. Hence, it remained limited with this experimental database Alumni and standard JOIN conditions only, making this research far from a general solution for similar scenarios.

<i>PERSONAL-INFO</i>	→	<i>ROLL_</i> <i>FULL_NAME</i> <i>F_NAME</i> <i>BTCH</i> <i>B_DATE</i> <i>BLD_GRP</i> <i>SPOUSE_NAME</i> <i>NO_OF_CHILDREN</i> <i>FLD_OF_INTEREST</i> <i>JOB_TITLE</i>
<i>CONTACT-INFO</i>	→	<i>COMPANY_NAME</i> <i>JOB_EXP</i> <i>ID</i> <i>CONTACT_NO</i> <i>EMAIL_ID</i> <i>FAX_NO</i> <i>WEBSITE_INFO</i> <i>CURR_ADDRESS</i>
<i>ROLL_</i>	→	<i>roll number</i> <i>roll no</i>
<i>FULL_NAME</i>	→	<i>who is</i> <i>name</i> <i>name of the student</i> <i>student name</i> <i>student's name</i>
<i>F_NAME</i>	→	<i>father</i> <i>father's name</i> <i>dad</i>
<i>BTCH</i>	→	<i>Batch</i>
<i>B_DATE</i>	→	<i>birthday</i> <i>birth date</i> <i>date of birth</i>
<i>BLD_GRP</i>	→	<i>blood group</i>
<i>SPOUSE_NAME</i>	→	<i>spouse name</i> <i>husband</i> <i>wife</i> <i>name of</i> <i>husband</i> <i>husband's name</i> <i>name of</i> <i>wife</i> <i>wife's name</i>

Figure 2.2. CFG for SELECT and FROM clauses (Source: Mahmud et al. 2015)

Neural Based Methods

The inherent complexity of real-world scenarios makes it impossible to create a complete set of grammar rules. Attempting to do so would result in a large number of parsers, incurring prohibitive costs in terms of both computational resources and processing time.

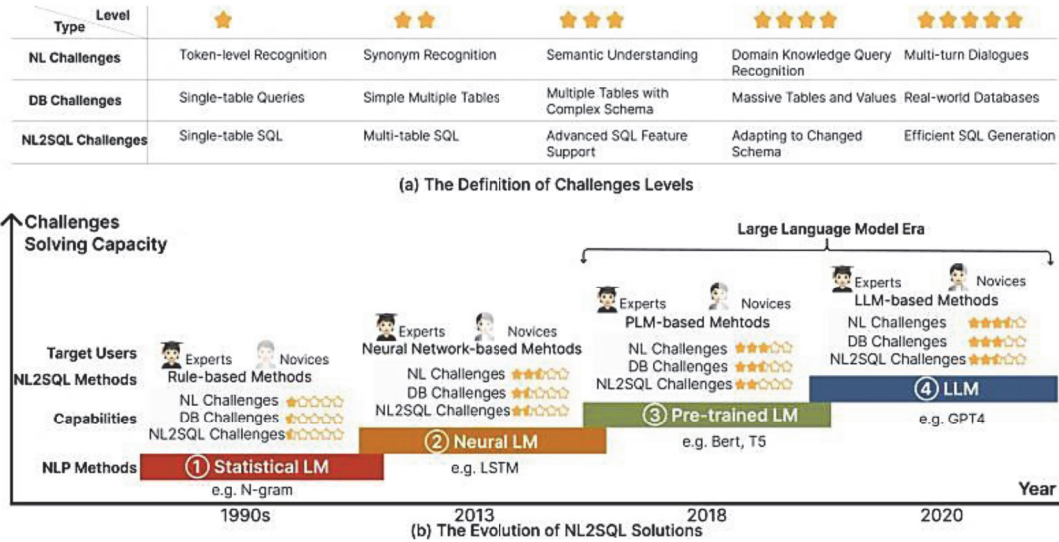


Figure 2.3. The evolution of NL2SQL solutions (Source: Liu et al. 2024)

To address the limitations of CFG grammars, neural network-based semantic parsing has emerged as an alternative. As an instance, in (Qi et al. 2022), authors proposed

RASAT, a seq2seq model, augmented self-attention module with new parameters. They incorporated the schema into T5-3B model during the encoding process and employed tree decoders to maintain the grammatical accuracy of the generated SQL queries.

In (Xiao, Dymetman, and Gardent 2016), researchers have introduced another sequence-based approach to semantic parsing. The method encodes logical forms into three different sequence representations: direct linearization, canonical form and derivation sequences. This study has achieved competitive performance compared to existing methods.

Figure 2.3 shows the evolution of NL2SQL solutions starting from 90s, including Rule-Based Methods and Neural Models approach. In the next sub-chapter, LLM-based solutions —the main focus of this research — will be examined in greater depth.

2.1.2. LLM-Based Methods

LLMs have emerged as a powerful solution in NL2SQL tasks. LLMs beat all the predecessor methods and reached the highest accuracy rates in text-to-sql tasks. As studied in (Zhu et al. 2024), there are a few methods researchers have focused on:

- **Fine-Tuning:** Initial approaches have primarily concentrated on leveraging fine-tuning techniques to develop ready-made SQL generator models. By fine-tuning pre-trained language models (PLMs) on extensive text-to-SQL datasets, researchers aimed to enhance the linguistic and semantic capabilities of these models specifically for database querying tasks. This targeted adjustment helps improve model performance in generating accurate and contextually relevant SQL queries. Defog’s SQLCoder, trained on over 20,000 human-crafted question-query pairs, has surpassed most open-source models as well as both GPT-4 and GPT-4 Turbo. It achieved a remarkable 93% accuracy on Defog’s sql-eval benchmark. However, fine-tuning requires significant computational resources and time and it may not perform optimally in more generalized cross-domain settings. The “exec match” results consistently show improvement with fine-tuned models, but it is mostly because these models have previously encountered the data during training. This fact highlights potential limitations of fine-tuning when applied to broader applications (Suhr et al. 2020).

- Prompt: Effective prompt design is critical because it directly influences the accuracy, efficiency, and overall success of interactions with LLMs. By providing clear, precise, and contextually rich input, users can harness the full potential of models.

As stated in research (Rajkumar, Li, and Bahdanau 2022), more information related to database schema and including few shots in prompts yield more accurate results than fine-tuned models. There is a comprehensive research on the importance of crafting a good prompt in NL2SQL tasks performance by (Chang and Fosler-Lussier, 2023). Allowing even a small amount of content exposure to the model can subtly improve the results by helping models understand the context, type and other relevant features of the data better.

- LLM Agents: Task decomposition can help with tackling complex tasks by dividing them into simple tasks. In LLM-agent approach, researchers assign subtasks to different agents which are able to perform them autonomously as experts of this simple tasks and aim to gain performance improvements through these agents' collaboration.

In NL2SQL, three main subtasks can be defined as studied in (Hong et al. 2024):

1. Question Understanding: The first step involves analysing the user's question relative to the database schema to determine their objectives. This includes extracting relevant entities, phrases, and terms that correspond to potential columns and conditions within the schema.
2. Schema Comprehension: Second step would be to match these entities to the database schema given, also known as schema binding or schema linking. Particularly in the Cross-Domain Semantic Parsing (XSP) setting, extracting relevant entities from the schema in natural language question is challenging unlike Single-Domain Semantic Parsing. Models are supposed to capture semantic associations in each and every unique question-database pair.
3. SQL Generation: This step can be defined as construction of SQL query based on the deduction made in the first two phases.

In study (Pourreza and Rafiei 2023), the authors proposed a method, Decomposed In-Context Learning–DIN SQL–where they broke down text-to-sql into the sub-tasks defined above. By utilizing few-shot learning, they achieved nearly 10% improvement in performance. This research adopts the same task decomposition, with the addition of an extra task aimed at correcting execution errors.

2.1.3. Challenges of NL2SQL Tasks

Researchers encountered several challenges during the study in NL2SQL tasks. Few key challenges needed to be explored more and addressed, as listed below:

- a. Lexical ambiguity: The most difficult and prevalent problem is “Lexical Ambiguity”. Say, in BI scenarios, there are tables containing plenty number of ambiguous columns. Also, the natural language question itself might be vague and have different interpretations making it more difficult to understand user’s intents and link the phrases to the schema.

For instance, the question “Return the top 3 singers” can be considered as ambiguous because it doesn’t specify whether the user wants the singers Ids or names or both. Figure 2.4 further illustrates another lexical ambiguity example. The intended task is for the LLM to identify customer names based on a specified condition, “who checked out books on 3 different genres” & “on Labor Day in 2023”. However, the term “genre” presents an ambiguity, as it’s unclear whether it refers to “LiteraryGenre” or “SubjectGenre” in this context.

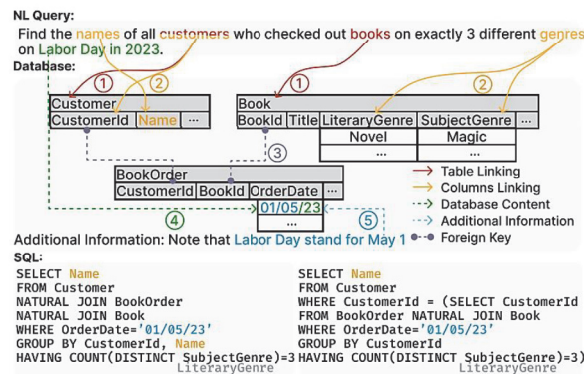


Figure 2.4. Ambiguous NL challenges (Source: Liu et al. 2024)

- b. Single Domain/Cross Domain Dilemma: Models trained on certain domain specific tasks do not perform well when applied to other domains because they are typically tailored to the structures, vocabulary and semantics of the original training data. However, they are expected to excel in adopting to new different

terminologies and phrasing across domain so as to cover other real world scenarios.

- c. **Dialect Diversity:** Most of the models have been trained on SQLite datasets, which make them blind to the other dialects such as BigQuery, PostgreSQL, Snowflake and others. Making the solution dialect agnostic. The research has been conducted in order to address this issue by providing a diverse dataset including BigQuery and PostgreSQL queries as well as SQLite queries in SQL-GEN work (Pourteza et al. 2024). This challenge still needs to be explored more.
- d. **Complex queries:** Complex queries that involve multiple JOINS, nested sub-queries, aggregations, or intricate conditions present significant challenges in NL2SQL tasks. These queries often necessitate a profound comprehension of both the database schema and the user’s intent in order to perform accurate mapping from natural language to SQL difficult. For instance, “select count(*) from cars_data where cylinders > 4” is an easy query giving an answer to “What is the number of cars with more than 4 cylinders?” (labelled as “Easy” in Spider dataset). To answer the question “What is the average life expectancy in countries where English is not the official language?”, a nested SQL query must be created that includes a JOIN operation and aggregations, along with multiple conditions, which is why it is labelled as “Extra Hard” in the Spider dataset.
- e. **Not being aware of tables content:** A lack of knowledge about table contents can also pose ambiguities in queries. For example, “American student” could be interpreted either as “WHERE country = ‘USA’” or “WHERE country = ‘The United States of America’”. In such scenarios, exposing the data to the model will be helpful to construct more accurate models.

Many studies still are ongoing to address and explore these challenges more.

2.1.4. Related Work

Both the history and the challenges of NL2SQL tasks have been covered above so far. In this sub-chapter, key recent studies that are pivotal in the literature and essential for understanding our approach will be reviewed.

a. DIN-SQL

A well-cited study DIN SQL, conducted by authors of (Pourreza and Rafiei 2023), has demonstrated that significant improvements can be achieved in text-to-SQL tasks by enhancing prompt techniques rather than through expensive fine-tuning. Before this study, prompting techniques was inferior to the fine-tuned model on many different text-to-sql leaderboards. In order to address this challenge, they have developed Decomposed In-Context Learning approach.

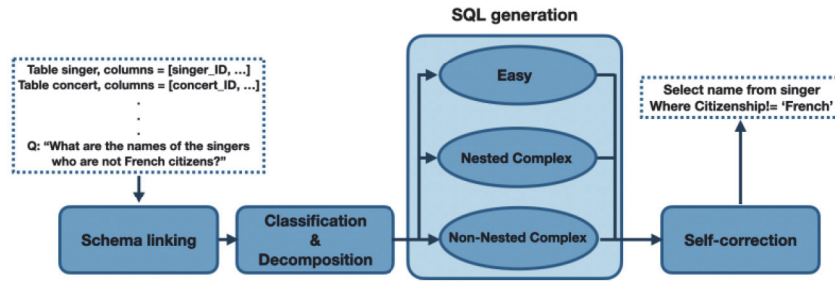


Figure 2.5. DIN-SQL modules organization (Source: Pourreza and Rafiei 2023)

First and foremost, they have classified the question-schema pairs based on the need for JOIN operations and nested queries using a model as “easy”, “non-nested” or “nested”. Based on this classification, they have enriched the query generation prompts by incorporating relevant schema links and a handful of exemplary contents.

By incorporating a self-correction module, the method became more robust and achieved impressive results in both execution match and execution accuracy metrics on hold-out test set of Spider evaluation benchmark, surpassing the previous score of 79.9% to reach an accuracy of 85.3%.

b. DAIL-SQL

Researchers from Alibaba Group have introduced DAIL-SQL as a state-of-the-art solution for text-to-SQL prompt engineering (Gao et al. 2023). They emphasize three key aspects: enhancing question representation, leveraging in-context-learning, and applying supervised fine-tuning (SFT) to improve performance. DDL statements, which contain more information with less tokens, have been adopted as a question representation.

The researchers employed in-context-learning by masking domain-specific words (tables, columns etc), enabling their solution to adapt to various domain schemas and

minimizing the impact of irrelevant information. This approach maintained the analogy between question-query pairs in examples while significantly reducing the number of tokens used. In addition to selecting an optimal question representation and implementing question masking, they employed fine-tuning techniques. This study resulted in achieving 86.2% execution accuracy on the Spider Leaderboard.

```

1 /* Some example questions and corresponding SQL queries
   ↳ are provided based on similar problems: */
2 /* Answer the following: How many authors are there? */
3 SELECT count(*) FROM authors
4
5 /* Answer the following: How many farms are there?. */
6 SELECT count(*) FROM farm
7
8 ${TARGET_QUESTION}

```

Figure 2.6. DAIL prompt organization (Source: Gao et al. 2023)

c. Alpha SQL

In (Li et al. 2025), authors have introduced a zero-shot Alpha-SQL framework that constructs SQL queries incrementally using Monte Carlo Tree Search (MCTS). MCTS, a robust search algorithm commonly used for decision-making, operates by building a tree of possible actions and states. The algorithm iteratively expands this tree, exploring potential solutions through a process of selection, expansion, simulation and backpropagation to identify the optimal decision. Alpha-SQL interpreted this algorithm into text-to-sql problem.

Researchers have defined an “*action set*” which includes column value / column function identification, question rephrasing, schema selection, generating SQL queries, or revising existing ones. They have utilized LLMs as reasoning “*action-model*”s to construct the query step by step after the action taken as illustrated in Figure 2.7.

In Alpha-SQL setup, nodes represent different phases of query building, while edges correspond to aforementioned actions that contribute to constructing the query. Through iteratively visiting the nodes, it generates multiple candidate SQL queries. Eventually, by filtering out invalid queries and scoring the remaining valid queries, it picks the query with the highest score as an output.

As demonstrated by the experiments conducted on the BIRD benchmark, Alpha SQL achieved an accuracy of 69.7%, matching a comparable result.

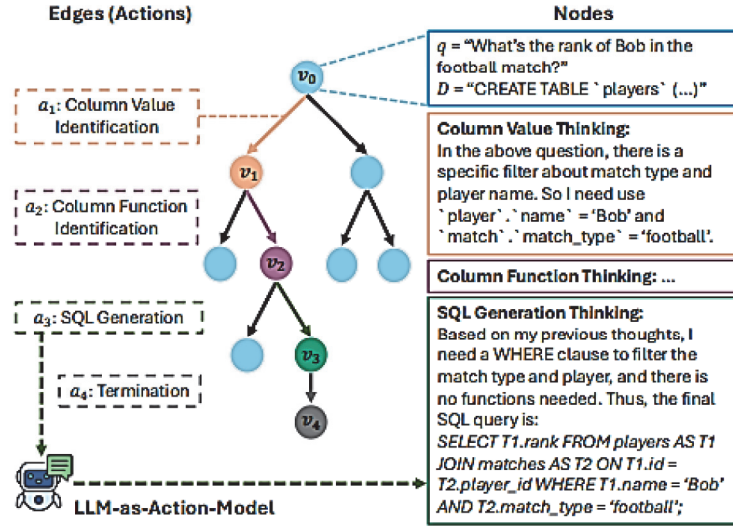


Figure 2.7. Search tree formulation in Alpha-SQL (Source: Li et al. 2025)

d. C3 SQL

Following the success of DIN-SQL, which demonstrated the potential of prompt engineering to improve text-to-SQL performance without fine-tuning, C3-SQL explores this approach further (Dong et al. 2023). DIN-SQL proved that carefully crafted prompts could significantly enhance results; however, designing these prompts required considerable expertise and was a time-consuming, labor-intensive process.

To achieve improved results, C3 relies on three key strategies: clear prompting, calibration with hints and consistency output. Researchers, adopting clear prompting technique, structured prompts with instructions, questions, and database schemas, but kept these elements distinct from one another using markers, avoiding concatenation. As experiments results that clear formats yield better results.

Researchers have found that ChatGPT tends to generate biased SQL code, including the inclusion of unnecessary columns and the preference for LEFT JOINs over stand-ard JOINs. They're addressing these biases by calibrating prompts with prior knowledge, such as the instruction, "Do not include COUNT (*) in SELECT statements unless explicitly requested."

Finally, recognizing the potential for multiple valid query formulations, researchers employed output consistency as a metric to ensure high confidence in the generated SQL. They achieved this by defining several reasoning paths, generating different SQL queries, and then determining which queries produced the most consistent results based on their execution.

This research has achieved 82.3% execution accuracy on Spider holdout test set, surpassed DIN-SQL conducted by (Pourreza and Rafiei 2023) on Spider Leaderboard.

e. XiYan-SQL

Another study (Gao et al. 2024), XiYan SQL, by Alibaba researchers achieved first place in the Spider benchmark, demonstrating an impressive accuracy rate of 89.65%. They combined prompt engineering with SFT based approach to generate high quality SQL query candidates.

This approach has introduced three phases: Schema Linking, SQL Candidate Generation using in-context learning (ICL) and supervised fine tuning (SFT) generator and eventually Candidate Selection.

- **Schema Linking:** To facilitate query generation, a retrieval module identifies the top-k columns likely to be relevant based on the semantic similarity between question keywords and the database schema. Addressing the aforementioned challenge of data value discrepancies—for example, a generated condition like ‘WHERE country.name = ‘USA’ might not match actual data like ‘America’—the authors also performed value retrieval to obtain likely query values. Integrating the results of both column and value retrieval, they linked the question to the most probable column and value combinations.
- **SQL Candidate Generation:** In this stage, the focus was on generating a diverse set of syntactically correct SQL candidates. To achieve this, a two-stage, multi-task Supervised Fine-Tuning (SFT) pipeline was developed.

The first stage involved training a pretrained language model (PLM) to generate syntactically valid queries. Then, to generate a broader range of SQL candidates, the base model was trained on tasks such as converting questions to queries, converting queries back to questions, and extracting evidence from queries to identify supporting data.

- **Candidate Selection:** Unlike many prior studies that evaluate query selection based solely on output consistency – a measure that does not guarantee correct results – XiYan-SQL utilizes a pre-trained selection model to pick the best candidate SQL as an output.

Rather than relying only on prompting techniques, XiYan-SQL has integrated SFT and prompting together within a multi-generator framework to achieve improved performance on NL2SQL tasks. This approach resulted in XiYan-SQL achieving first place on the Spider Leaderboard.

2.2. Multi-Agent Systems

A monolithic agent architecture, tasked with performing all tool calls and decision-making for a complex task, is susceptible to errors and performance degradation due to the breadth of perspectives and tasks it must manage. Alternatively, a multi-agent system can be designed with specialized agents handling smaller, more manageable tasks, leading to more robust and reliable results.

What Can Multi-Agent Systems Achieve?

- **Separation of Concerns:** Multi-Agent Systems facilitate task decomposition and specialized agent design. Developers can assign sub-tasks to individual agents and customize their behaviour based on specific requirements. This enables users for fine-grained control, one agent might benefit from a broad range of LLM outputs (achieved through parameters setting like “frequency_penalty”, “temperature”, “top_k”, and “top_n”), while another might necessitate strict, consistent results. Prompting strategies can also be customized for each sub-task to maximize effectiveness of LLM backed intelligent models. This allows for the development of specialized prompts that incorporate relevant context and targeted examples, significantly improving performance. To achieve the goals of each sub-task, language models can be fine-tuned or specialized through in-context learning or retrieval augmentation.
- **Model Diversity:** Multi-Agent Systems enable the integration of diverse open-source models for collaboration on various tasks, potentially trained on different datasets.
For example, if a model lacks a desired feature like tool-calling, the task can be divided into sub-tasks – reasoning and execution – and assigned to specialized models with tailored configurations.
- **Improved Accuracy Through Debate:** As studied in (Du et al. 2023), multi-agent conversations do not necessarily have to exhibit collaboration and share the objectives in order to achieve better accuracy. They can also be intentionally structured to encourage debate and discussion. Within the scope of aforementioned study, researchers have employed various model configurations, enabling the agents to reason about a given query and engage in a debate of their respective

justifications. Through debate and iterative, multi-turn conversation, these agents often come to the consensus in time.

- **Increased Reliability and More Robust Applications:** The use of collaborative agents increases the reliability of responses and lowers the risk of hallucinated content. Even in cases where one agent generates an inaccurate response, the others work to preserve contextual alignment.
- **Human and Tool Interaction:** Multi-agent systems allow developers to define both human and tool agents, facilitating interaction with users for tasks such as approval, evaluation, and information gathering.

A potential drawback of multi-agent approaches is increased processing time due to the higher number of calls to LLMs. Furthermore, poorly designed agent structures can lead to suboptimal results.

Conversational Orchestration

Strategic agent orchestration is vital for ensuring to meet the objectives of this study. While certain sub-tasks require a sequential execution due to dependencies between outputs, such as in the workflow of code generation, execution and evaluation, others benefit from simultaneous performance, allowing agents to reason and discuss queries in parallel. It is also possible to adopt selective approach which the speaker can pick the next agent. Thus, the conversational organization of agents can be roughly divided into two modes: centralized and decentralized.

Centralized Organization: In this approach, a designated planner agent gathers responses from various agents and then routes them to the next agent in the process. This system functions similarly to a star topology and can be arranged in a hierarchical structure to manage complexity.

Decentralized Organization: Agents can be orchestrated in a mesh-like fashion, allowing for direct communication between them. In this configuration, agents can either broadcast responses or selectively pass them to relevant agents. This way of structure is well-suited when sub-tasks are conditioned on the results of preceding agents' responses.

Structured output is a comparatively new feature, released in August 2024 by OpenAI community, that facilitates interaction between LLM agents, enabling JSON formatted outputs, particularly beneficial in mathematical and coding applications.

By employing structured outputs, robust message exchange between agents is facilitated, thereby reducing the risk of misinterpretation. In this study, structured outputs are utilized in order to enforce that the generated response conforms to a predefined JSON

format so as to make parsing easier.

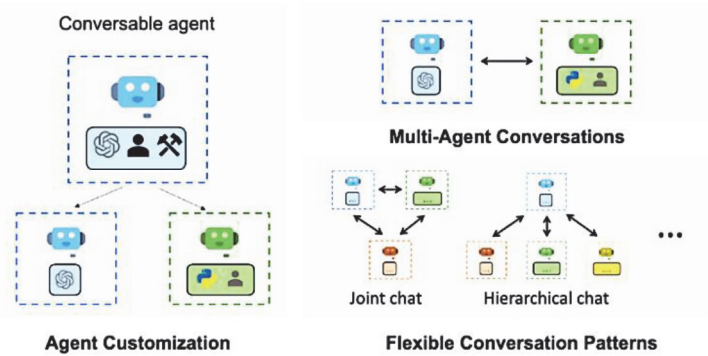


Figure 2.8. Autogen framework conversable agents (Source: Wu et al. 2023)

2.2.1. Frameworks

Two popular multi-agent frameworks will be explored in this chapter: LangChain, an open-source project launched by Harrison Chase in October 2022 (Mavroudis 2024), and Autogen, another open-source multi-agent framework designed by Microsoft Research's AI Frontiers Lab.

2.2.1.1. LangChain

In LangChain, 'Chain' represents a fundamental organization principle. Each agent's response is incorporated into the prompt for the subsequent agent, including the system prompt. The fact is that LLMs are stateless and have no memory. But an illusion of memory is created by concatenating previous messages into the constructed prompt.

Although LangChain incorporates historical data into the input prompt when transferring responses between agents, this approach is constrained by the model's context window. Any information that falls outside of this window is effectively invisible to the model and cannot be utilized (Topsakal and Akinici 2023).

Alternatively, vector stores can be utilized in order to enhance context awareness. These stores allow us to embed and retrieve contextually relevant data by performing similarity searches. Various file types, including PDFs and CSVs, can be loaded, split into smaller chunks and embedded using a pre-defined embedding model. This allows the model to access and utilize the most relevant data when reasoning about a query. The retriever, on the other hand, uses similarity search techniques to efficiently locate the most relevant pieces of information from the vector store based on the user's query.



Figure 2.9. Work principle of retriever (Source: Topsakal and Akinci 2023)

Vector store and retriever components play the pivotal role in performing Retrieval Augmentation, as described above, by providing models with access to relevant data.

The interest of OpenAI community in LangChain has rapidly increased over time, reflecting the framework's proven success in developing a wide range of applications, including chatbots, code understanding tools, automated scientific literature review and data querying systems similar to our scenario etc.

2.2.1.2. Autogen

Microsoft researchers have released Autogen, an open-source framework for multi-agent applications, as detailed in (Wu et al. 2023). In this study, the authors focused on *conversational* aspect of agents and implemented a programming paradigm they called “*conversation programming*” which has two core concepts at its heart: *conversation-centric* computation and control flow.

Computation refers to the action an agent takes during a conversational turn. This

action can be a human response or a tool call executed by an assistant agent. *Control flow*, on the other hand, defines the order in which computations are executed. This approach is conversation-centric, meaning that agents' responses dictate the sequence of actions.

Autogen introduces three types of capabilities to empower the agents:

LLM: Leveraging LLMs enhances agent intelligence, enabling capabilities like inference, reasoning, decision-making and coding. Autogen empowers developers to tailor prompts, output formats, model parameters and other skills to meet specific agent requirements.

Human: Autogen customizes a UserProxyAgent to enable developers to keep humans actively engaged in the conversation. This is particularly important for applications like customer service, virtual assistants and chatbots as humans are at the heart of conversation.

Besides, human input is often needed for tasks like data analysis, healthcare diagnosis, financial advising etc. in order to clarify intent, evaluate multiple outputs or supplement missing information. Application developers only need to set "human_input_mode"="ALWAYS" in UserProxyAgent so as to incorporate human inputs to the loop.

Tools: Tools are supplementary functions which can be executed by the agents.

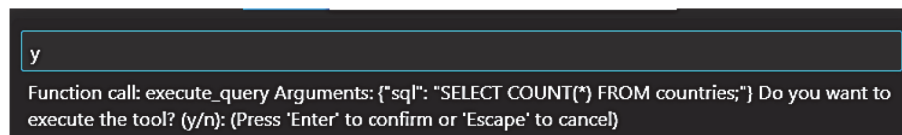


Figure 2.10. Model asking for human input

Figure 2.10 shows an example where an agent utilizes the `execute_query_tool`, a pre-defined function, to interact with a database. The agent is initially prompted with *"Execute following query: SELECT COUNT(*) FROM countries;"* and subsequently requests human confirmation before executing the query.

Upon receiving 'y', the agent forwards the prompt to the LLM for response generation. As the `execute_query_tool` has been previously introduced to the model, it can intelligently interpret the prompt and invoke this external tool with the input 'sql' provided as illustrated in Figure 2.11:

```
Executing query: SELECT COUNT(*) FROM countries;  
The result of the query is:  
  
**Total number of rows in the 'countries' table:** 249
```

Figure 2.11. Output of the agent backed with tool

Code-executing agents are typically defined as *UserProxyAgent* and can run code within Docker containers to ensure safety and isolation. By simply setting the `use_docker` configuration to `True`, a Docker container—either user-defined or the default—will be automatically launched for execution and stopped afterward.

A *ConversableAgent* is an agent is the highest level of abstraction which is inherited by *AssistantAgent*, *UserProxyAgent* and *GroupChatManager* where each subclass represents a common usage mode. They communicate through unified interfaces, `send/receive` and `generate_reply`. Once the custom reply functions are registered and the conversation is initialized, the conversation will proceed autonomously without requiring external control or intervention.

Autogen has been elected as a multi-agent application framework in this study for two reasons:

1. Autogen is a relatively new tool with a lot of potential for exploration,
2. Easy to set up and use, open source framework which is also open to modifications and contributions.

Given the decision to utilize AutoGen in this study, its significant recent evolution—marked by enhanced robustness and capabilities—is acknowledged. Current features include support for document loaders, structured outputs, vector stores and retrievers with customized options, and AzureAI client integration. Although keeping up with these updates posed challenges, the latest version of AutoGen was utilized in our research, so as to be able to leverage its up-to-date capabilities.

2.2.2. Applications in Text-to-SQL Tasks

Leveraging Multi-Agent collaboration offers some compelling advantages that address limitations of traditional, single-agent approaches. Firstly, NL2SQL tasks are complex and better be handled in several phases. Prior work, as already discussed in

NL2SQL Chapter, commonly decomposed the challenging text-to-SQL task into several subtasks, including schema linking, query understanding and query construction. Building upon this, multi-agent systems are capable of assigning each of these subtasks to individual agents, and utilizing the tool-calling functionality of LLMs to guarantee the constructed SQL query is valid and executable within the specific database.

Importantly, multi-agent systems facilitate the creation of highly specialized agents by using retrieval augmentation and fine-tuning the models for particular tasks and domains, leading to improved performance.

a. QCMA-SQL (Question Classification and Multi-Agent Collaboration)

To address the challenges encountered in previous single-agent Text-to-SQL studies, authors of (Shao et al. 2025) have proposed a two-stage solution. This approach divides the task into question classification / *schema selection* and query construction / *multi-agent collaboration*.

The schema selection component classifies questions based on difficulty and selects the most relevant database schema from a pool of schema options at the initial phase. This model leverages various cross-attention mechanisms, on the purpose of capturing connections between question-table and difficulty-question pairings, during training.

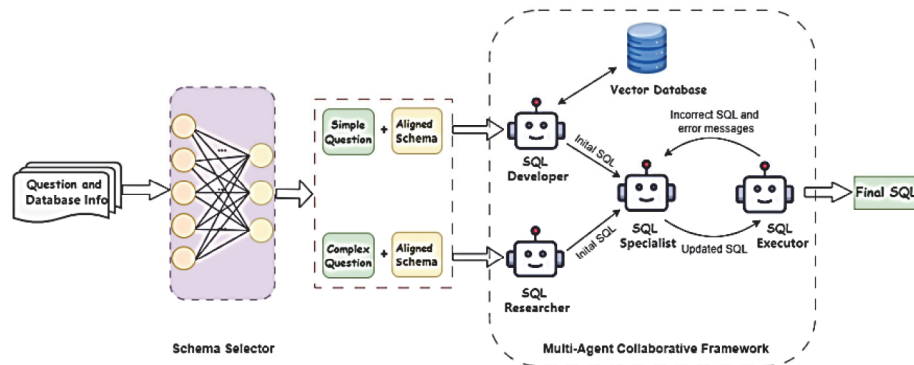


Figure 2.12. Overview of QCMA-SQL framework (Source: Shao et al. 2025)

Once the schema selector component labels a question based on its difficulty, the question-schema pair is then passed to a multi-agent framework composed of *developer*, *researcher*, *executor* and *specialist* agents. To maximize each agent's capabilities, the authors have designed individual prompts.

Easy questions are directed to the *developer agent*, while more difficult questions are handled by the *researcher*. The *executor* performs syntax checks and executes queries, collaborating with the *specialist agent* to ensure query validity. *Specialist agent* revises and refines invalid queries as the whole framework depicted in Figure 2.12 above.

For complex questions, *researcher agent* utilizes a chain-of-thought (CoT) approach, unlike the *developer agent*, which constructs queries by retrieving similar question-query pairs from a pre-built vector store database.

Through ablation studies, authors revealed that question classification contributes a 2.8% improvement in execution accuracy. QCMA-SQL has hit 87.4% accuracy on Spider Leaderboard.

b. MAC SQL

Within the scope of MAC-SQL study (Wang et al. 2025), a new approach called the Multi-Agent Collaborative Framework has been developed to tackle the text-to-SQL challenge. As of the completion of this research, March 2025, many researchers had largely overlooked the potential for models to utilize tool-based capabilities. MAC SQL integrates a *decomposer* agent at its core and is supported by two additional auxiliary agents, namely the *refiner* and *selector* agents.

Decomposer agent handles dividing tasks into smaller, manageable subtasks, while the *selector* agent focuses on eliminating redundancies from the schema when needed to ensure it fits within the prompt. Finally, *refiner* agent employs an external tool via the large language models' (LLMs) external call capability to execute the query. If any errors are detected, it further refines the query to ensure accuracy.

This framework not only facilitates collaboration among different agents but also incorporates SQL-Llama, which is customized for question-query pairs using Code Llama 7B as its foundation.

As a result of these innovations, the MAC SQL system achieved a 59% success rate on the BIRD benchmark.

CHAPTER 3

METHODOLOGY

3.1. Problem Definition

Given a natural language question Q , and a database schema S , where S consists of table T , columns C and relations R , denoted as $S = \{T, C, R\}$. The objective of this study is to generate a corresponding SQL query q .

$$q = func(Q, S) \quad (3.1)$$

where $func$ is a function that maps the question Q to the database schema S to a syntactically and semantically correct q .

3.2. Hardware

This research focuses on developing an NL2SQL framework capable of running on readily available consumer-grade GPUs. To minimize computational costs, relatively compact models with approximately 14 billion parameters were utilized, employing Q4 precision. The models are used solely for inference and not for fine-tuning. Therefore, the VRAM requirement can be calculated as follows:

$$VRAM = \frac{N \times Q}{8 \times 2^{30}} \times 1.2 \quad (3.2)$$

where, N = number of parameters, which is 14 billion in our case and Q = quantization parameter, representing 4 bits per parameter (since 4-bit quantization was used in this study).

Calculated by applying a factor of 1.2 to account for the additional 20% VRAM required for activations and other memory overhead and dividing by 8 in order to convert the result into bytes and 2^{30} to get the result in GB.

$$VRAM = \frac{(14 \times 10^9) \times 4}{8 \times 2^{30}} \times 1.2 \cong 7,82GB \quad (3.3)$$

While a minimum of 8 GB of memory is required, Nvidia 4070, which offers 16 GB of VRAM will be sufficient to run this NL2SQL experimental framework effectively.

3.3. Software

This study was conducted using Python 3.11.2 within a dedicated virtual environment ‘venv’ to ensure isolation and reproducibility. VS Code, configured with Python and Jupyter Lab extensions, was utilized as the Integrated Development Environment (IDE).

Autogen, along with its dependencies, has been installed, and is leveraged by our Natural Language to SQL (NL2SQL) framework. Autogen, as discussed in prior chapters, allows users to build conversational multi-agent applications. This project utilizes OpenAI capabilities, which usually require installing the OpenAI Python package and having an OpenAI account with a valid API key. However, to make this solution more cost-effective, Ollama is used instead.

Ollama is a lightweight and convenient solution that allows users to download and run OpenAI-compatible open-source models directly on their own computers, eliminating the need to authenticate with external servers. Although pre-loaded models are currently being used, the flexibility to create custom models is also offered by Ollama.

Autogen provides the “OllamaChatCompletionClient” API to interact with models hosted by Ollama. This API allows developers to define and customize models according to the specific requirements. Figure 3.1 shows one of the ollama clients customized in this study. Open-source models with the tuning parameters and customizations employed in this study will be covered with reasons in the next sub-chapter.

```
query_helper_client = OllamaChatCompletionClient(  
    model="qwen2-5",  
    host="http://127.0.0.1:11434",  
    model_info={  
        "vision": False,  
        "function_calling": False,  
        "json_output": True,  
        "family": ModelFamily.UNKNOWN,  
        "structured_output": True,  
    },  
    options={  
        "num_ctx": 16384,  
        "frequency_penalty": 0.0,  
        "temperature": 0.0,  
    },  
    response_format=SQLOutput,  
    max_tokens=300,  
)
```

Figure 3.1. Ollama client instantiation

3.3.1. Models

Qwen2.5 14B Instruct: Instruction tuned 14.7B parameter language model, part of Qwen2.5 LLM family. It supports a context window of up to 128K tokens and can generate outputs of up to 8K tokens.

This series was trained on Alibaba’s extensive dataset and demonstrates superior performance compared to equivalent models and became prominent in generation, reasoning and fixing in coding tasks. There are also “coder” and “math” versions tailored specifically for coding and math tasks.

Gemma3-IT 12B: 12 billion parameter, instruction tuned model from Gemma family, introduced in March 2025 by Google. Gemma3 models are trained with distillation technique where a smaller model is trained to mimic well-performing larger model. It demonstrates strong performance in question answering, reasoning and summarization

tasks, making it well-suited for understanding user goals expressed in natural language questions.

Mistral Nemo Instruct 2407 12B: Mistral Nemo, developed through the joint efforts of Mistral AI and Nvidia, is released under the Apache 2.0 license. Notably, this model supports a 128K context window, indicating its capacity to handle substantial prompts. Both base and instruct versions are available.

In subsequent chapters, the utilization of Mistral Nemo as a refiner agent, specifically for correcting queries based on error messages, will be detailed.

Model Evaluation

The performance of Gemma3-IT and Qwen 2.5 Instruct was assessed by visualizing and analysing previously published results across a diverse set of benchmarks as shown in Figure 3.2, including MATH (mathematical reasoning), MMLU-Pro (multi-task language understanding), and HumanEval (code generation and execution).

For both science-related question answering and reasoning tasks—including intermediate and advanced mathematical reasoning—Gemma3-IT and Qwen2.5 Instruct demonstrate comparable performance. However, as demonstrated by the MBPP scores, Qwen 2.5 outperforms Gemma3-IT by a significant margin of 9%, indicating stronger performance on Python programming tasks.

Table 3.1. Base model performance comparison on different benchmarks

	MMLU-Pro	HumanEval	MBPP	GPQA	GSM8K	MATH
Qwen 2.5 Instruct 14B	63.7%	83.5%	82.0%	45.5%	94.8%	80.0%
Gemma3 IT 12B	60.6%	85.4%	73.0%	40.9%	94.4%	83.8%

Figure 3.3 below, on the other hand, visualizes the performance of Gemma3-PT (pre-trained) 12B and Qwen 2.5 14B Instruct across several benchmarks—WinoGrande, HellaSwag, and TriviaQA—that assess reasoning and question-answering abilities. Given that the IT versions of Gemma3 perform equally well or better than the pre-trained (PT) versions of the family, we can infer that Gemma3 IT is high likely to achieve better scores than Mistral Nemo on these benchmarks as well. So, Qwen 2.5 Instruct and Gemma3-IT

each exhibit distinct strengths in reasoning tasks and achieve comparably strong results across various benchmarks. In contrast, Mistral's performance generally falls behind these two models.

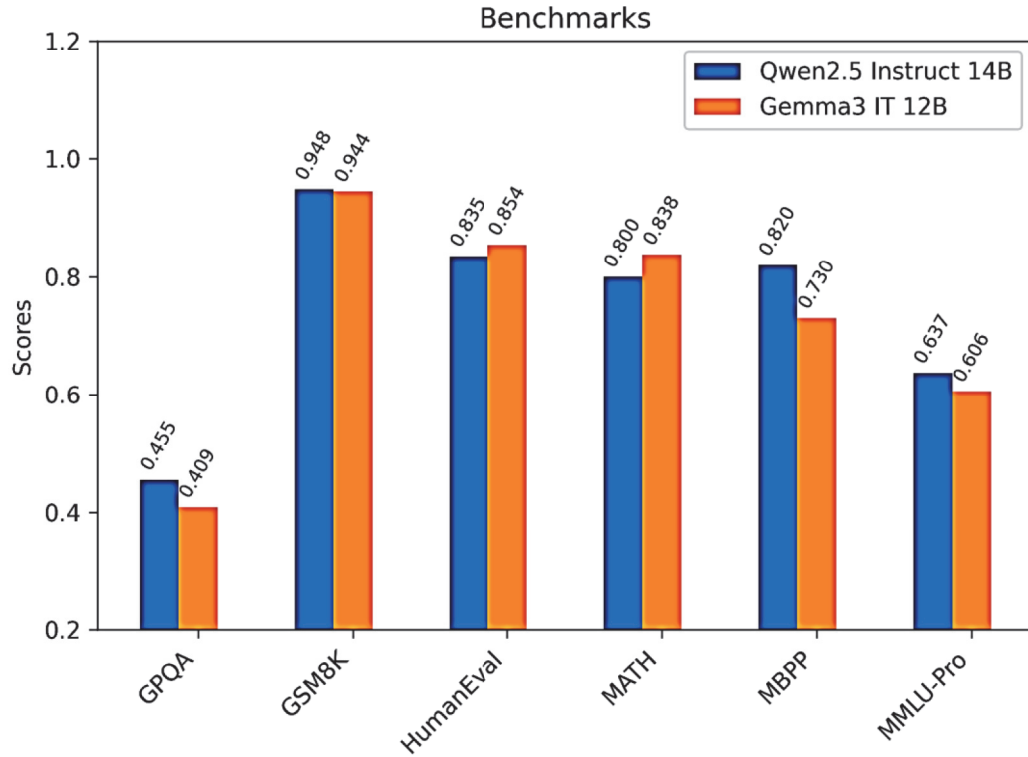


Figure 3.2. Qwen2.5 14B Instruct & Gemma3-IT 12B scores on various benchmarks

Table 3.2. Gemma3-PT 12B & Mistral Nemo Instruct scores

	HellaSwag	WinoGrande	N. Questions	TriviaQA	MMLU
Mistral Nemo	83.5%	76.8%	31.2%	73.8%	68.0%
Gemma3 PT 12B	84.2%	74.3%	31.4%	78.2%	74.5%

By collaborating four agents, backed with these three models and customized for NLSQL subtasks, it will be explored in this study whether the NL2SQL system can be strengthened and the accuracy of SQL results improved.

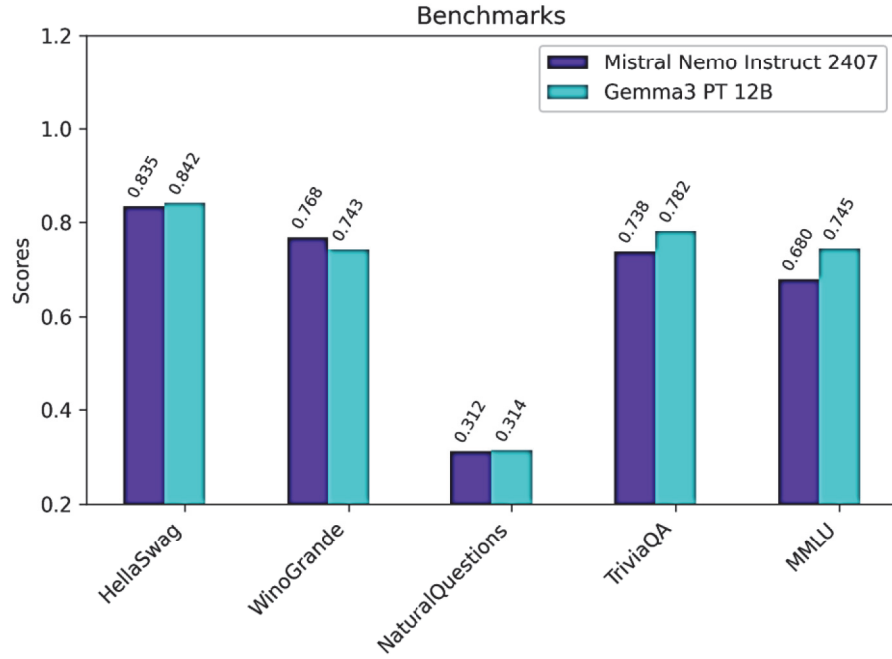


Figure 3.3. Gemma3-PT 12B & Mistral Nemo Instruct scores on various benchmarks

To evaluate the effectiveness of the models on NL2SQL tasks, experiments were performed on the Spider development dataset using each individual model, with particular emphasis on execution accuracy. The detailed results are presented in the following chapter.

It is important to note that results may vary depending on experimental conditions, such as prompt design, parameter settings, and even the hardware used. For this reason, thorough documentation of the experimental setup will be provided to ensure transparency. Our code is available online to allow for independent verification and replication of our findings.

Prompting Techniques and Model Parameters

The prompting techniques and model settings configured for generating SQL results within our *single-agent setup*, as illustrated in Figure 3.4, will now be examined:

OpenAI Prompt Format:

A zero-shot prompting technique was used, with prompts structured using a clear layout that included the instructions, database schema, and the question itself, without the inclusion of example prompts.

These elements were separated with ‘#’ delimiter rather than concatenation, as this layout has been shown to yield better performance in various studies.

Parameter Tuning

Temperature: Higher temperature values increase creativity and randomness, which can be useful in experimental setups aiming to generate diverse SQL candidates. However, this approach also carries a higher risk of hallucination. To mitigate this and ensure query stability within our single-agent experiment, the temperature has been set to 0.0, minimizing the introduction of randomness.

Frequency & Presence Penalty: Both penalty values are allowed to be between -2.0 and 2.0 as described in OpenAI API. A higher penalty values encourage the model to avoid repeating tokens within the same generated text as covered above. Lowering the frequency penalty value is a common technique for generating more deterministic and predictable results. Following the settings in DIN-SQL, both values were tuned to 0.0.

Context Window: Number of tokens that model can process at once when generating response. While larger context windows allow for processing longer prompts, as discussed in Chapter 2, increased prompt length does not always correlate with improved accuracy.

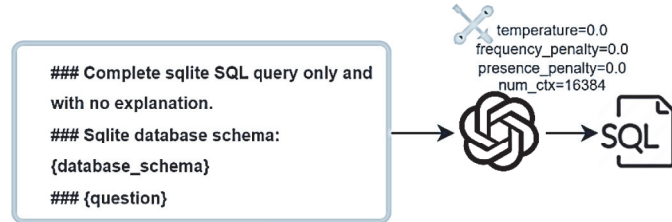


Figure 3.4. Single agent text-to-sql experimental setup

In NL2SQL tasks, it is observed that performance tends to decrease with longer database schemas. Some researchers have explored reducing token count by omitting irrelevant portions of the database schema from the prompt. Furthermore, in study (Chang 2023), it is clearly stated that more increasing examples in the prompt got performance declined too after 4 examples.

Given that all schemas in the Spider dataset comfortably fit within a 16K window and the models used in the experiments support up to a 128K window size, the schema was included in the prompt as is, and a context window limit of 16K was established.

3.3.2. Ollama

Ollama is a lightweight application, primarily written in Go, that allows developers to run open-source language models locally. It provides simple commands for downloading models and customizing their behaviour through parameters and templates. To minimize costs, Ollama's ability to run free models was leveraged, allowing the expense of paid alternatives to be avoided.

Structured output support was introduced by Ollama in December 2024. This functionality was utilized to format the SQL builder agent's output, simplifying parsing and facilitating easier communication between agents. In the following chapters, a detailed explanation will be provided regarding how outputs were structured and inter-agent communication was orchestrated.

While alternative frameworks like Vllm (developed initially in the Sky Computing Lab) are optimized for high-speed token generation, Ollama was chosen in this study due to its broader support for multi-agent systems, ongoing rapid development, extensive documentation, ease of use and widespread adoption.

3.3.3. Microsoft Autogen

The Autogen framework was utilized to facilitate multi-agent experiments.

Rationale for Using AutoGen

The AutoGen framework was employed in the implementation of the multi-agent solution in this study. Its use was motivated by several factors:

- to explore the framework's capabilities, including memory management, tool invocation, and human-agent interaction,
- to leverage its modular architecture and role-based encapsulation,
- and to facilitate implementation and future modifications, particularly in cases requiring control flow or retry mechanisms, without necessitating a complete reengineering of the pipeline.

Nevertheless, due to the inherently sequential nature of NL2SQL tasks, a

predefined control flow was adopted rather than decentralizing the conversation among agents.

Building upon previous research, the text-to-SQL task was divided into four distinct subtasks, with each subtask assigned to a dedicated agent through the use of four designated clients. The four agents and their tasks are defined below:

Conversable Agents

- 1- Analyst Agent:** An intelligent agent powered by an LLM. Its primary function is to interpret user goals and intent from nl questions. It receives a question, extracts the desired output and how to achieve the user's objectives.

Our experiments revealed that the model sometimes interpreted user questions in a way that produced redundant outputs, posing challenges for generating accurate SQL queries. However, precise interpretation of natural language queries remains essential for correct query generation. To overcome this challenge, Analyst Agent was elaborately instructed to carefully analyse the natural language question combined with the contextual data and output only the information explicitly requested by the user.

Another challenge, as discussed in Challenges in NL2SQL Tasks (Chapter 2), is the models' lack of awareness regarding database context, which can lead to ambiguity and incorrect queries. For example, a user request for the number of singers from America might generate a query using `WHERE country = 'U.S.A.'`, even though the data is stored as "America".

To address this, the Analyst Agent is provided with the necessary context the most relevant to the question. Further details on this approach and challenges encountered will be provided in a subsequent chapter.

- 2- Schema Linker Agent:** Also, an LLM powered intelligent agent, tasked with establishing correspondences between the database schema and the entities identified in the user's goals. It receives user goals, previously analysed and provided by an analyst agent, with the DDL statements of database schema and generates a schema linking output.

Both the data analyst and schema linker agents are powered by Gemma3-IT, as explained above, which provides the core intelligence for their operations.

- 3- SQL Builder Agent:** The third intelligent agent, tasked with generating the SQL query by integrating the analysis of the user's question with schema linking information performed by the Schema Linker Agent.

To mitigate potential model biases that could affect the results, the *Builder Agent*

is instructed to rely solely on the provided information when generating the query.

4- Refiner Agent: serves as the final executor of SQL queries generated by a builder agent. It leverages a language model to correct and re-execute queries in case of errors, with a maximum of five attempts.

A limit is placed on the number of query correction attempts to avoid scenarios where the agent enters an infinite loop or inadvertently alters the query in a way that deviates from the user's original objectives. To mitigate deviation, Refiner Agent is provided with the initial Analyst Agent's output and instructed to stick to the user goals described in this output when correcting the query.

Conversational Orchestration Between Agents

The Analyst Agent and Schema Linker Agent utilize the Gemma3-IT 12B model in a two-stage process to facilitate query generation.

The Analyst Agent receives access to the relevant contextual data, which may include valuable information such as string literals.

Stage 1: *Natural Language Query Analysis* – Model is prompted to:

- Identify user goals, target data and relevant string literals for condition checks.
- Determine the expected output format (single-row or multi-row).
- Analyse expected ordering, grouping, filtering and conditional requirements.
- Explain the reasoning behind its analysis.

The analysis results are subsequently forwarded to the Schema Linker Agent. To avoid redundant processing, the original natural language question is omitted.

Stage 2: *Schema Linking* – The Schema Linker Agent then prompts model to:

- Perform schema linking, identifying the list of tables and columns corresponding to the user goals and objectives based on the database schema.
- Give an explicit explanation about reasoning.

The *Builder Agent* then utilizes the Qwen2.5 14B Instruct model to construct a SQL query. This process leverages the reasoning outputs from the *Analyst Agent* and *Schema Linker Agents*, along with the original nl question and database schema, to avoid any deviation from the user's true intentions by any chance.

After the query is constructed, the *Refiner Agent* executes the generated SQL query. If an error occurs during execution, it attempts to resolve the issue by analysing the error message and the faulty query, while striving to preserve the original user intent.

Otherwise, the query is outputted directly. Figure 3.5, illustrates the flow of conversation between agents.

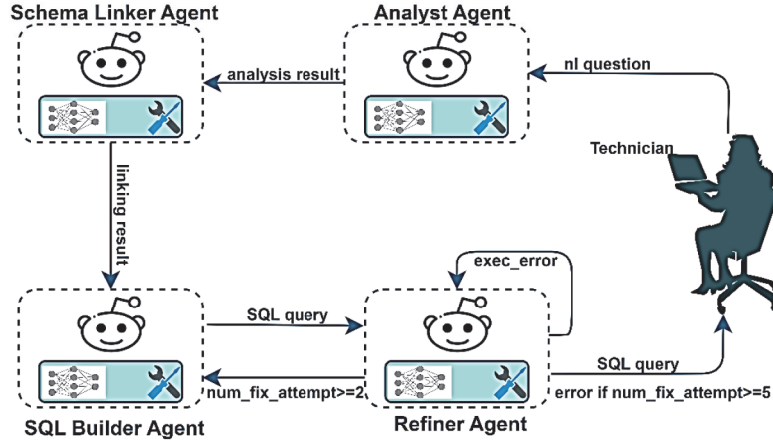


Figure 3.5. Multi agent text-to-sql experimental setup

3.4. Evaluation Benchmark and Metrics

The Spider benchmark dev tables and questions were selected for evaluating our results due to their popularity and comprehensive nature within the NL2SQL task domain. The Spider dataset, introduced by the authors of (a group of 11 Yale University students), is a cross-domain resource containing 10,181 questions paired with 5,693 unique SQL queries, spanning 200 distinct databases (Yu et al. 2018).

Several alternative benchmarks, including ATIS, Geo, and Academic, are available. However, a key limitation of these is their dependence on single databases. Optimizing prompts and experimental setups to perform well on one of these benchmarks carries a high risk of poor performance when applied to other, different databases.

WikiSQL provides an alternative large, cross-domain benchmark. But despite its size, it focuses primarily on simpler query structures. To evaluate our setup’s robustness across a spectrum of complexity, Spider remains a valuable and appropriate choice.

As another large, cross-domain benchmark, BIRD (Big Bench for laRge-scale Database), introduced in (Li et al. 2023), is increasingly being adopted, much like Spider. It consists of 12,751 text-to-SQL pairs based on 95 different databases.

Utilizing the Spider benchmark enables direct comparison between our work and existing baselines, as it is the most widely adopted NL2SQL benchmark at the time of this study.

To ensure an unbiased evaluation of system performance, Spider dataset is partitioned into three distinct subsets: training, development and testing.

- Train set has 7000 question-query pairs,
- Development set 1034 question-query pairs,
- Finally, the test set 2047 question-query pairs and 40 schemas kept hidden.

Table 3.3. NL2SQL benchmarks with properties

	Examples	DB	Table/DB	Rows/DB	Keywords	JOINS/SQL
WikiSQL	80654	26521	1	17	12.0	0.0
Spider	10181	200	5.1	2000	36.0	0.5
BIRD	12754	95	7.3	549000	51.0	1.0

There is a possibility that the models have already seen the Spider train data, so for evaluation, our setup was rigorously tested using the development set.

Table 3.3 compares three of the benchmarks, in terms of question-query pairs, #databases, #tables per database, #rows our database, #keywords and #JOINS per SQL. Figure 3.6, visualizes the evaluation result.

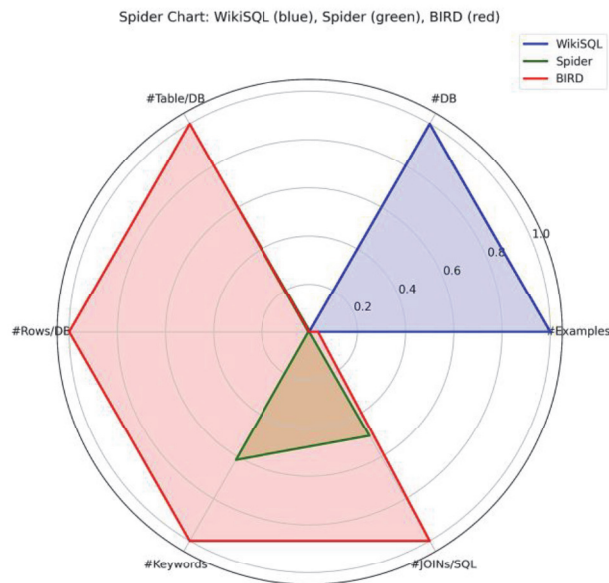


Figure 3.6. WikiSQL & Spider & BIRD benchmark comparison

Evaluation Metrics

Spider Benchmark has four different hardness levels: easy, medium, hard and extra hard. These levels are defined based on:

Core Components: WHERE, GROUP BY, ORDER BY, LIMIT, JOIN, OR, LIKE, HAVING

Advanced Components: EXCEPT, UNION, INTERSECT, NESTED

Others:

- number of agg > 1,
- number of select columns > 1,
- number of where conditions > 1,
- number of group by clauses > 1,
- number of group by clauses > 1

Hardness Criteria

Easy: If SQL query has zero or one core component and no advanced component or other rules.

Medium: If query satisfies one of the conditions below, it is labelled as medium:

- At most one core component, no advanced components and at most two rules in others,
- Exact two core components,
- No advanced component and less than two rules in others.

Hard: If SQL query satisfies one of the conditions below, it is labelled as hard:

- At most two core components, no advanced components and at more than two rules in others,
- Three core components, no advanced component and at most two rules in others.
- At most one core component and exact one advanced component with no rule in others.

Extra Hard: All others.

All: All question-query pairs to be scored.

1. Exact Match (EM)

The Exact Match (EM) evaluation method checks against the whole string, without condition values, match between gold query and predicted query.

For a given question, there may be numerous potential SQL queries, each with a different level of efficiency. However, the primary objective of our research is to generate queries that produce the correct results, rather than to identify the most efficient solution.

Consequently, an evaluation approach that prioritizes accuracy has been adopted.

2. Component Matching

The evaluation process compares the gold standard and predicted queries component by component, analysing elements like SELECT and WHERE clauses. To account for cases where the order of components doesn't affect the result (e.g., 'avg(weight), max(weight)' vs. 'max(weight), avg(weight)'), the evaluation is performed using an unordered set of components.

The objective of this research was to enhance accuracy rather than to achieve the most optimized or high-performance results. Consequently, execution accuracy (EX) was employed as the primary evaluation metric for the experimental results, in comparison to individual base models and established baselines.

3. Execution Accuracy (EX)

The Exact Match evaluation is susceptible to producing false negatives; a predicted query that yields the correct output can still be penalized if it does not precisely match the gold query. To mitigate this problem, Spider also utilizes an Execution Accuracy method, which assesses performance by comparing the results generated by executing both the predicted and gold queries.

While system accuracy was primarily assessed using Execution Accuracy, it is acknowledged that this metric can be influenced by false positives. Specifically, situations where two distinct queries result in an empty execution result are flagged as exact matches, which can be misleading. An exact execution match does not guarantee a full alignment between the query and the intended natural language question.

However, as illustrated in Figure 3.7, false negatives may arise due to ambiguity in natural language. In the example question, the user requests “*first name, last name, and the size.*” While *first name* and *last name* are straightforward, the term *size* is ambiguous—it can reasonably refer to either *size_description* or *size_code*. Since both interpretations are semantically valid, treating only one as correct during evaluation results in a false negative, despite the prediction being justifiable.

According to (Pourreza and Rafei, 2023), the existing NL2SQL benchmarks have limitations, and the authors pointed out that achieving 100% accuracy is fundamentally impossible, even with the best possible model.

Being aware of the possibility of false positives and false negatives, the Spider Benchmark was selected as the testing environment, and Execution Accuracy (EX) was employed as the primary evaluation metric.

In the next chapter, the challenges and failures encountered during evaluation will be discussed, along with an exploration of the risks of biases likely stemming from the training data used in NL2SQL tasks.

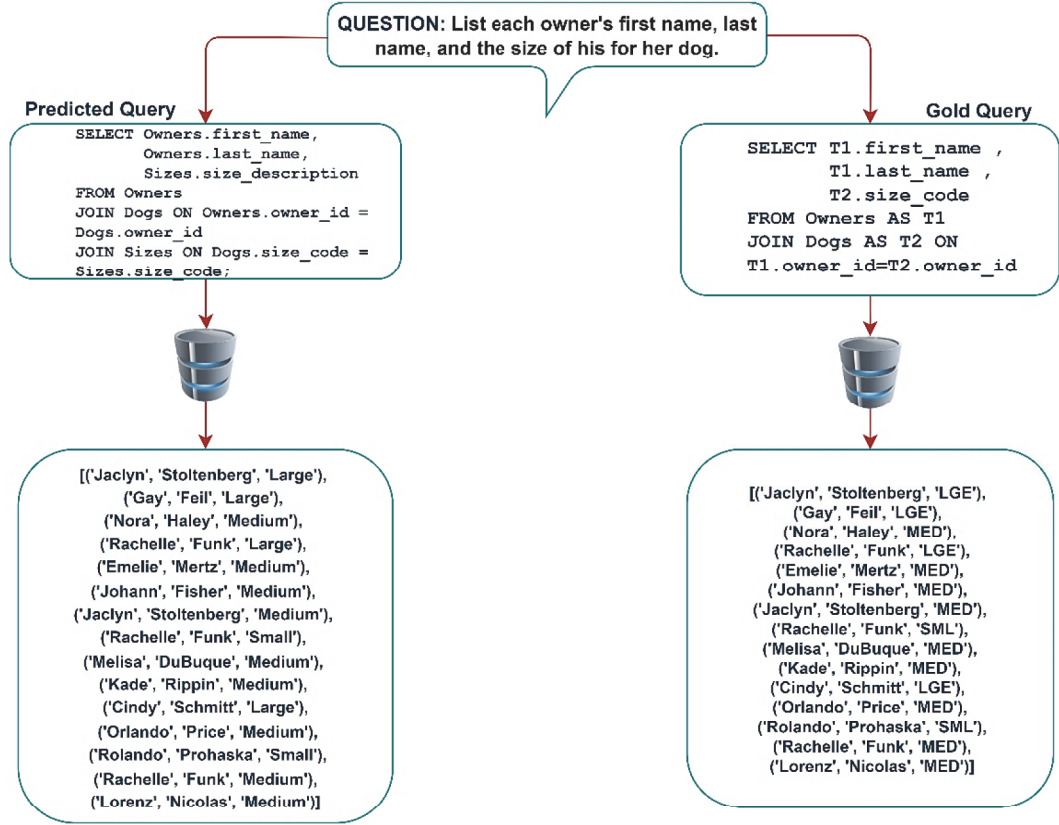


Figure 3.7. An example of false negative result

3.5. Baselines

In studies, QCMA-SQL (Shao et al. 2025) and MAC-SQL (Wang et al. 2025) discussed above, multi-agent collaboration approach was explored in NL2SQL tasks. However, these studies employed distinct training methodologies. QCMA-SQL trained a selector model to classify questions and identify the appropriate schema, while MAC-SQL fine-tuned a new model, SQL Llama, which is based on Code Llama 7B.

In this study, rather than training a model, the NL2SQL task was broken down into smaller components. Tailored agents were then prompted for each individual subtask,

enabling cooperation and the utilization of their inherent reasoning abilities through collaborative interaction. Accordingly, the experimental results were compared against those of C3 SQL and DIN SQL. Because these studies similarly focused on improving performance by separating concerns and designing clearer prompts.

Table 3.4. DIN-SQL vs C3-SQL execution accuracy comparison

	Easy (248)	Medium (446)	Hard (174)	Extra Hard (166)	All (1034)
DIN-SQL	92.3%	87.4%	76.4%	62.7%	82.8%
C3-SQL	92.7%	85.0%	77.6%	62.0%	81.9%

The previously published predicted queries from both studies were evaluated using Test Suite Accuracy (Zhong, Yu, and Klein, 2020) in order to compare and evaluate these studies. Execution accuracy results are presented in Table 3.4 and Figure 3.8.

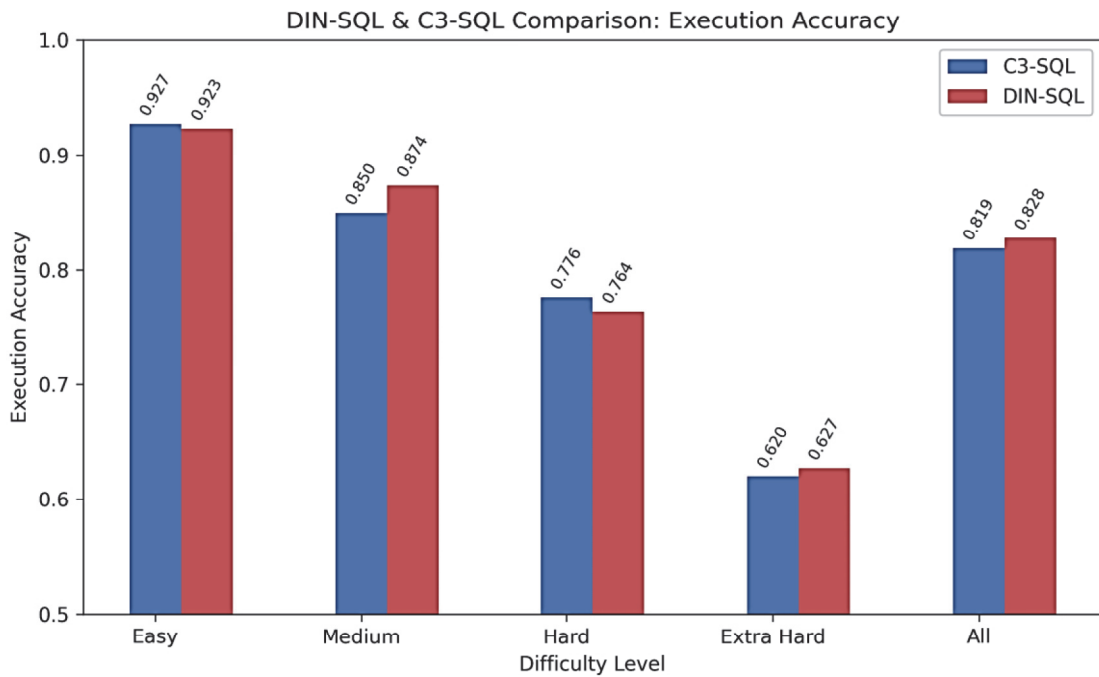


Figure 3.8. C3-SQL & DIN-SQL Spider benchmark execution accuracy results

CHAPTER 4

EXPERIMENTS

4.1. Single-Agent Experiments

In the single-agent experiments, a zero-shot prompting method was utilized, in which the model was provided with the database schema and a natural language question. To ensure deterministic results, the model's temperature and frequency penalty were set to 0, in accordance with the methodology outlined in Chapter 3, Figure 3.4.

Since the Ollama framework employs 4-bit quantization by default, and this study aims to develop a low-cost solution, the default quantization setting was retained in order to minimize memory consumption.

Results presented in Table 4.1 and Figure 4.1 indicate that Gemma3-IT 12B and Qwen2.5 Instruct 14B demonstrated competitive results on Spider-Dev benchmark. Qwen2.5 demonstrated a performance advantage on extra-labelled questions, which represent the most challenging queries in the benchmark. These questions incorporate multiple core and advanced components, along with rule-based constraints, as defined in Chapter 3. Although Mistral Nemo's performance on easy queries closely matched Qwen2.5's, ultimately it came after the other two models overall.

Mistral Nemo, while competitive on *Easy* and *Medium* questions, shows a sharp decline in performance as difficulty increases, reaching just 36.1% on *Extra Hard*. This indicates that Mistral Nemo may be less capable of handling complex reasoning or multi-step logic. In contrast, Qwen2.5 and Gemma3 maintain relatively high performance across difficulty levels, highlighting their stronger generalization and reasoning capabilities, especially in challenging tasks.

The results of this evaluation informed our decision to focus query construction primarily on Qwen2.5 Instruct and Gemma3-IT while Mistral Nemo could be employed as a complementary assistant to identify and correct syntactic errors that arise during the query construction process.

Table 4.1. Execution accuracy for base models

	Easy (248)	Medium (446)	Hard (174)	Extra Hard (166)	All (1034)
Mistral Nemo 12B	86.7%	74.9%	56.9%	36.1%	68.5%
Qwen2.5 Instruct 14B	89.9%	79.4%	63.8%	53.0%	75.0%
Gemma3 IT 12B	91.9%	80.3%	67.2%	47.6%	75.6%

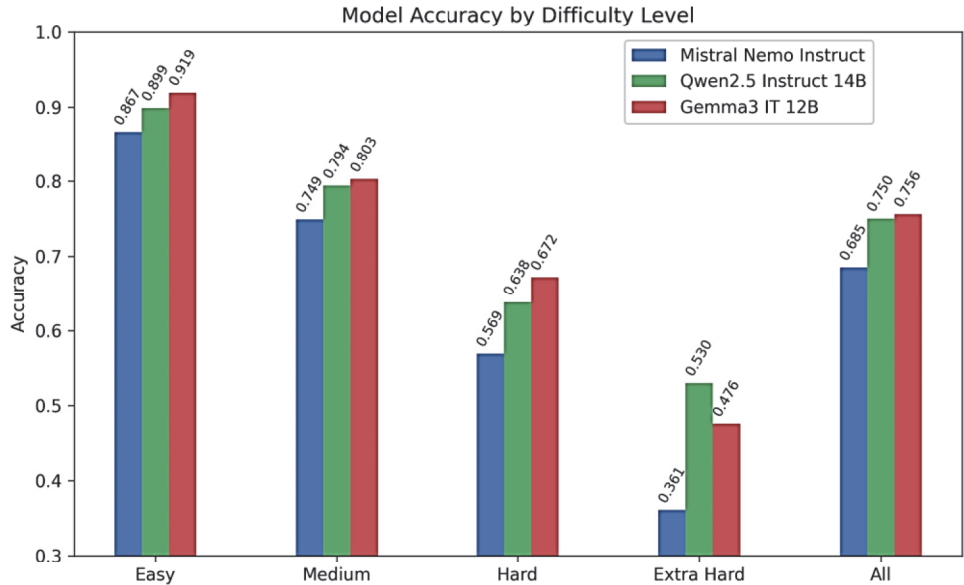


Figure 4.1. Execution accuracy results – Mistral Nemo, Qwen2.5 Instruct 14B and Gemma3 IT 12B

4.1.1. Error Analysis in Single-Agent Query Generation

i. Execution Errors

Execution errors in SQL may arise due to various factors, including invalid syntax, mismatches between queried tables and their columns, or references to non-existent database objects such as tables, columns, views or functions.

Among the evaluated models, **Mistral Nemo** exhibited the highest number of execution errors (58), including 2 attributable to syntax issues. In comparison, **Qwen2.5**

Instruct and **Gemma3 IT** encountered 31 and 33 execution errors respectively, with 5 of Qwen’s and 2 of Gemma3’s errors identified as stemming from syntax-related causes. Other observed issues predominantly consist of “*no such column/table*” errors, typically arising from the use of incorrect or non-existent table or column names. Additionally, *ambiguous column name* errors were encountered when a column name appeared in multiple tables within a query context, preventing the SQL from resolving the reference.

Table 4.2. Number of execution errors for base models

	Mistral Nemo Instruct	Qwen2.5 14B Instruct	Gemma3-IT 12B
#(Exec Errors)	58	31	33

ii. Discrepancies Between Predicted and Expected Query Results

a. **Natural Language Question Misinterpretation:** Semantic problem where the nl input question is misunderstood by the model, leading to the generation of an incorrect or irrelevant SQL query.

For instance, Spider question below, ‘*Show all template type codes and the number of documents using each type.*’, presents an ambiguity requiring clarification. It can be interpreted in two distinct ways, resulting in different model outputs:

- Inclusive Interpretation: Show every template type code, even those not used by any documents (i.e., count = 0).
- Restrictive Interpretation: Only show template type codes that are actually used by at least one document. So, types with 0 usage must be excluded.

Qwen2.5 Instruct interpreted the natural language question according to the first interpretation—i.e., to include all template type codes, even those not associated with any documents. Hence, it generated the following SQL query with LEFT JOIN:

```
SELECT T1.Template_Type_Code, COUNT(T3.Document_ID)
AS Document_Count
FROM Ref_Template_Types AS T1
JOIN Templates AS T2 ON T1.Template_Type_Code = T2.Template_Type_Code
LEFT JOIN Documents AS T3 ON T2.Template_ID = T3.Template_ID
GROUP BY T1.Template_Type_Code
```

This query returned the output:

[('AD', 3), ('BK', 5), ('CV', 0), ('PP', 4), ('PPT', 3)]

However, the gold query assumes that only template type codes **in use** should be included. Therefore, the expected output must be:

[('AD', 3), ('BK', 5), ('PP', 4), ('PPT', 3)]

This illustrates that natural language misinterpretation is not always about a clear mistake—it can stem from *reasonable ambiguity* in the input. The model must either resolve it correctly (e.g., using context) or prompt for clarification.

b. Schema Linking Problems: A problem, where the model misidentified columns or tables. This likely stems from the model's difficulty in accurately interpreting the user's intent and connecting it to the database schema.

Question, "Show the name and the release year of the song by the youngest singer," requires a literal interpretation. The prompt specifically requests the song name and release year.

However, Gemma3-IT and Qwen2.5 Instruct incorrectly associated the song name with the 'name' column instead of the 'song_name' column, resulted in inaccurate <singer name, song release year> pairings.

Question: Show the **name** and the **release year of the song** by the youngest **singer**.

SINGER						
SINGER_ID	Name	Country	Song_Name	Song_Release_Year	Age	Is_male

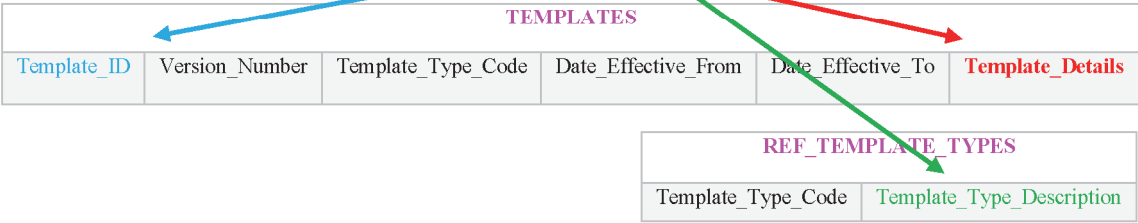
```
SELECT Name, Song_release_year
FROM singer ORDER BY Age LIMIT 1;
-- Gemma3 IT 12B

SELECT T1.Name, T1.Song_release_year
FROM singer AS T1
JOIN (SELECT Singer_ID, MIN(Age) AS MinAge
      FROM singer
      GROUP BY Age
      ORDER BY MinAge ASC LIMIT 1
) AS T2 ON T1.Singer_ID = T2.Singer_ID
-- Qwen2.5 Instruct 14B

SELECT song_name, song_release_year FROM singer
ORDER BY Age LIMIT 1;
-- Gold Query
```

Another exemplary Spider question and the inaccurate queries generated by the models are illustrated below. Both Gemma3-IT and Mistral Nemo incorrectly mapped the phrase "template type description" to the Template_Details column in the Templates table above. This incorrect schema linking leads to a query that does not align with the intended semantics of the question.

Question: What are the *template ids* with *template type description* "Presentation"



```

SELECT Template_ID
FROM Templates
WHERE Template_Details = 'Presentation'           --Gemma3-IT & Mistral Nemo
Which returned an empty result: []

SELECT T2.template_id
FROM Ref_template_types AS T1
JOIN Templates AS T2
ON T1.template_type_code = T2.template_type_code
WHERE T1.template_type_description = "Presentation"           --Gold Query
Expected output: [(6,), (7,), (10,)]

```

To link schemas effectively and accurately, it's vital to really understand what users are trying to achieve and what they're asking for. Dividing the text-to-SQL task into two parts – understanding the question first and then linking it to the schema – helps with this. We'll show you how with some practical examples in the next chapter.

c. Lack of Data Context : Lack of data context means that the model is not aware of where the data is stored and how it's formatted, the model may generate wrong literals, leading to the generation of incorrect string literals, because the model doesn't know how values are actually represented or stored in the database.

This represents a significant limitation in SQL generation, where queries that are semantically well-formed may nonetheless produce incorrect outputs. For example, a column like 'high_definition_tv' might store values as 'Yes', 'yes', 'Y', 'y', or even as a pre-defined code string. The only reliable solution to address this is to provide the model with

access to the actual data, allowing it to observe the specific literals used and understand the data's storage conventions.

This lack of data context proved to be a consistent challenge across the three models assessed in our single-agent experiments, hindering their ability to generate accurate filters and join conditions and highlighting the need for retrieval-augmented generation or stronger models to address this limitation. For instance;

Question / Spider: Find the package choice and series name of the TV channel that has high definition TV.

Predicted Queries:

```
SELECT Package_Option, series_name
FROM TV_Channel
WHERE Hight_definition_TV = 'Yes';
```

-- Mistral Nemo Instruct

```
SELECT T1.Package_Option, T1.series_name
FROM TV_Channel AS T1
JOIN TV_series AS T2 ON T1.id = T2.Channel
WHERE T1.Hight_definition_TV = 'Yes';
```

-- Qwen2.5 Instruct 14B

```
SELECT Package_Option, series_name
FROM TV_Channel
WHERE Hight_definition_TV = 'Yes';
```

-- Gemma3 IT 12B

Gold Query:

```
SELECT package_option , series_name FROM TV_Channel
WHERE hight_definition_TV = "yes";
```

In this example, the case sensitivity of the string literal is critical, as it directly impacts the correctness of the filter applied in the `WHERE` clause. Notably, all three models utilize 'Yes' with an uppercase "Y," while the actual value stored in the database is "yes" in lowercase. This mismatch leads to the query returning an empty result set which is incorrect.

Another example detailed below, where the values of actual `property_type_code` column are stored as House and Apartment, all three models generated nonexistent codes — hse and apt—, resulting in queries that fail to retrieve accurate results due to this hallucination.

Question: What are the names of properties that are either houses or apartments with more than 1 room?

Predicted Queries:

```
SELECT property_name FROM Properties
WHERE (property_type_code = 'HSE' OR property_type_code = 'APT')
AND room_count > 1
```

-- Mistral Nemo Instruct

```
SELECT property_name FROM Properties
WHERE (property_type_code IN ('HSE', 'APT'))
AND room_count > 1
```

-- Qwen2.5 Instruct 14B

```
SELECT property_name FROM Properties
WHERE property_type_code
IN ('hse', 'apt') AND room_count > 1
```

-- Gemma3 IT 12B

Gold Query:

```
SELECT property_name FROM Properties
WHERE property_type_code = "House"
UNION
SELECT property_name FROM Properties
WHERE property_type_code = "Apartment" AND room_count > 1
```

While not a complete solution, exposing a portion of the contextual data offered a step toward mitigating this problem. The benefits of this approach and the reasons for its limitations will be elaborated upon in the next chapter.

d. Model Biases: A significant bias in the models' responses was observed as another challenge, and it proved difficult to mitigate. Training datasets reflect the biases, stereotypes, and patterns that exist within the real world practice. These pre-existing biases can subtly or dramatically influence the model's output, even when all the necessary context, database schema and natural language question are provided.

For example, as demonstrated in the queries below, the model frequently defaulted to assuming the aggregation function was AVG. However, in this example there is a column with "average" name that the model should link "the average" to the column instead of assuming it directly AVG function.

Question: What is the maximum capacity and **the average** of all stadiums?

STADIUM						
STADIUM_ID	Location	Name	Capacity	Highest	Lowest	Average

Predicted Queries:

SELECT MAX(Capacity) AS Max_Capacity,

AVG(Capacity) AS Avg_Capacity

FROM stadium;

-- Mistral Nemo 12B

SELECT MAX(Capacity) AS Max_Capacity,

AVG(Capacity) AS Average_Capacity

FROM stadium;

-- Qwen2.5 14B

SELECT MAX(Capacity), AVG(Average)

FROM stadium;

-- Gemma3 12B

Gold Query:

SELECT MAX(Capacity), Average FROM stadium;

Model biases can lead to literal hallucinations and the neglect of edge cases. This behaviour results in errors or inaccuracies, especially when the model encounters scenarios that deviate from its learned biases.

4.1.2. Model Behaviour Across Spider Benchmark

Initially, Qwen outperformed the others for the first ~500 questions. But, approximately after the 500th question, Gemma3-IT and Qwen2.5 Instruct achieved comparable accuracy levels, maintaining a relatively stable performance thereafter.

Across the majority of question-query pairs represented on the x-axis, all three models exhibit almost a consistent pattern of behaviour.

However, in the interval between 300 and 400, while accuracy tends to increase for both Gemma3 and Qwen2.5, Mistral shows a decline in performance, with its trend

line exhibiting a downward trajectory. Similarly, between 400 and 500, Qwen2.5 exhibits a decline in performance, as reflected by a downward trend in the plot, whereas Mistral Nemo and Gemma3 demonstrate a slight improvement over the same interval.

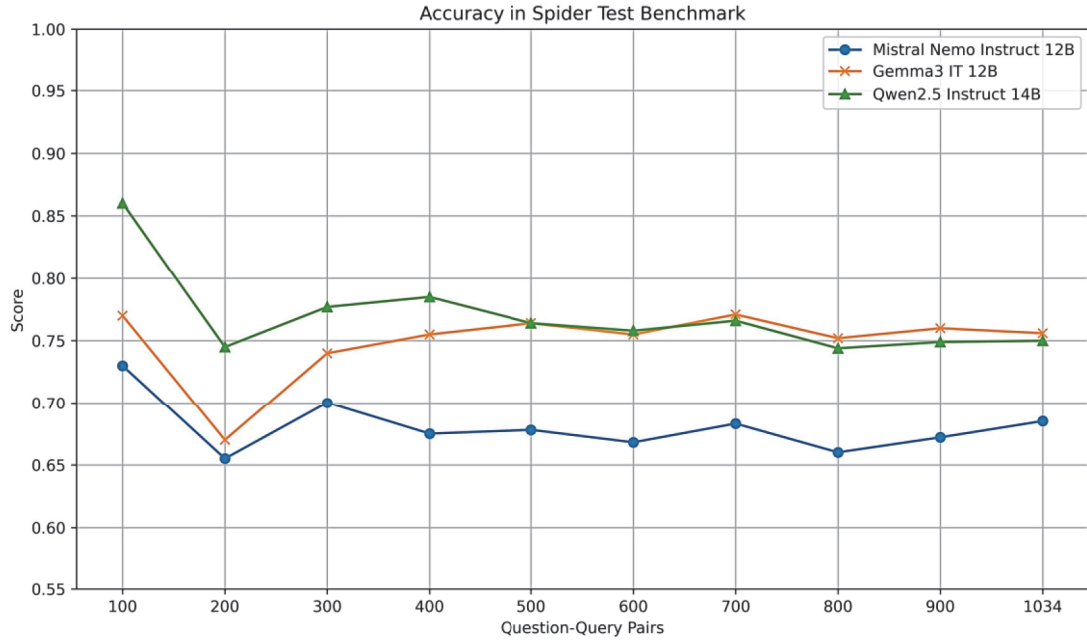


Figure 4.2. Accuracy comparison of base models across Spider-Dev questions

This performance pattern may indicate the dominance of a particular model while others encounter difficulties. Consequently, across all benchmark question-query pairs the models have the potential to play a complementary role by working jointly, mitigate individual errors and enhancing overall accuracy.

4.2. Multi-Agent Experiments

A collaborative framework, where agents are assigned specific subtasks and provided with relevant contextual data exclusively for the analyst agent, yielded improved accuracy compared to individual models. This approach demonstrates a promising method for enhancing NL2SQL performance by leveraging specialized agent

contributions, as illustrated in Figure 4.3.

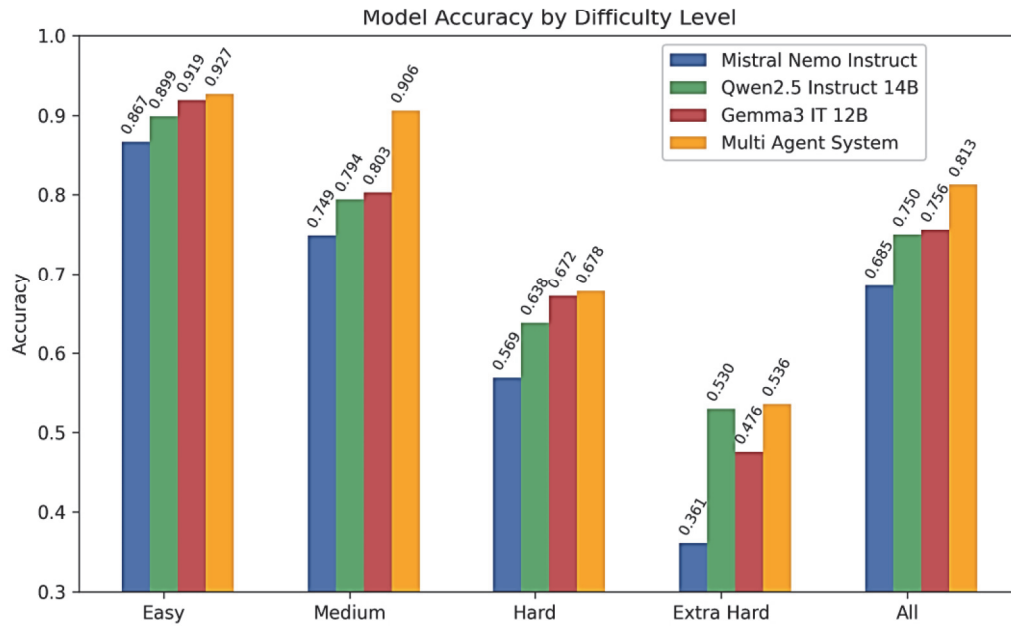


Figure 4.3. Multi agent system performance comparing to the base models

Table 4.3 quantifies this improvement, showing explicit accuracy gains across four difficulty levels and overall. The context-aware multi-agent approach boosted the performance of Qwen 2.5 Instruct by 6.3%, Gemma3-IT by 5.7% across all categories.

Table 4.3. Execution accuracy of context aware multi-agent system comparing to base models

	Easy (248)	Medium (446)	Hard (174)	Extra Hard (166)	All (1034)
Mistral Nemo	86.7%	74.9%	56.9%	36.1%	68.5%
Qwen 2.5	89.9%	79.4%	63.8%	53.0%	75.0%
Gemma3 IT	91.9%	80.3%	67.2%	47.6%	75.6%
Multi Agent System	92.7%	90.6%	67.8%	53.6%	81.3%

It is evident that the majority of the overall improvement stems from medium-labelled queries, which involve two or fewer core components and rules, and do not include any advanced components as previously defined. For queries that do include

advanced components—such as *EXCEPT*, *UNION*, *INTERSECT*, and nested structures—the multi-agent system achieved only marginal gains, surpassing the highest-performing individual model by a small margin in terms of accuracy.

Despite the observed improvements, reasoning capabilities remain insufficient for achieving higher performance on more complex queries. The observed gains can be attributed to mechanisms such as self-correction, context awareness and task decomposition. However, context awareness introduces additional challenges. First, a U-shaped relationship has been identified between context length and model effectiveness; when the data context exceeds an optimal threshold, the model may overlook the most critical elements of the prompt. Second, the inclusion of irrelevant or inaccurate contextual information may mislead the model, further degrading its performance.

On the other hand, while self-correction mechanisms can render queries executable by resolving syntactic or structural errors, they may also introduce deviations from the original intent of the question, potentially leading to failure. These challenges will be explored in greater depth in the subsequent chapters.

As illustrated in Figure 4.4, the performance of the multi-agent system follows a similar trend observed across the benchmark question-query pairs, exhibiting a consistent margin of improvement. This suggests that the multi-agent system encounters difficulty with the same subset of questions that challenge the individual models.

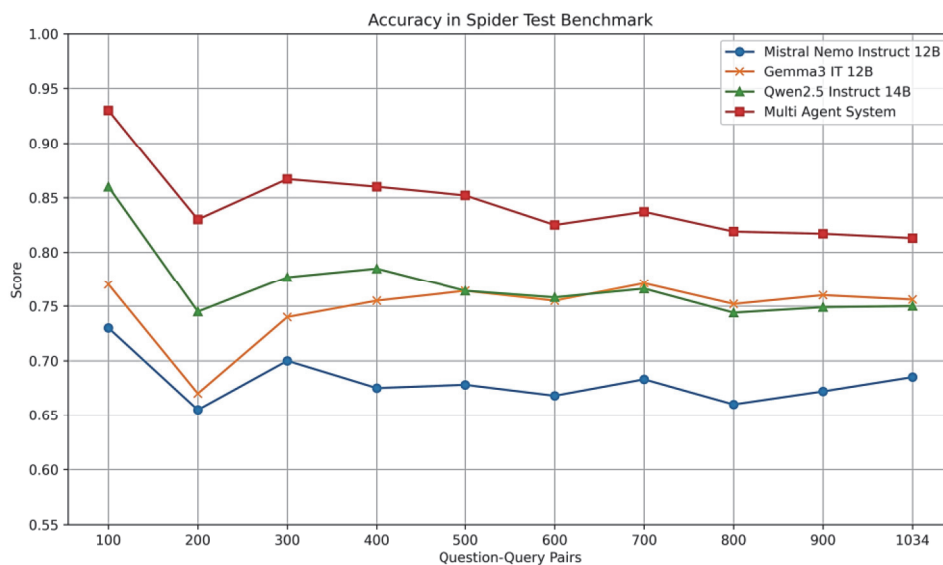


Figure 4.4. Accuracy comparison of base models comparing to multi-agent system across Spider-Dev questions

It was anticipated that the multi-agent configuration would align with the highest-performing models within the 300–400 and 400–500 question intervals. However, contrary to expectations, the accuracy of the multi-agent system declines within the 300–400 interval, despite observed performance gains by Qwen2.5 and Gemma3. This decline mirrors the trend observed in Mistral Nemo, albeit with a less pronounced slope.

Again, between the 400–500 range, the multi-agent system shows a decline in NL2SQL performance similar to Qwen2.5, though with a gentler slope, while Gemma3 and Mistral Nemo exhibit slight improvements.

The benefits of task decomposition, introducing contextual signals into the system and self-correction will now be explored.

4.2.1. Task Decomposition and Customized Prompting

The multi-agent system is composed of agents assigned to distinct subtasks: question understanding, schema linking, query building and query refinement. Each agent is prompted with instructions specifically tailored to its designated role. In addition, custom instructions were incorporated into the prompts to address issues identified in earlier experiments. To begin with, the Analyst Agent was directed to explicitly highlight the string literals required for constructing query conditions, and it was provided with contextual data to prevent incorrect conditional checks.

State string literals to be used in condition checks....

###Helpful Data (string values exactly as they appear in the reference data)

{reference_data}

Secondly, as noted in study (Pourreza and Rafiei, 2023), predicted queries can fail due to top ties—situations where multiple top-ranked rows share the same value. For example, several students might have the highest GPA, so it is worth to reason over if the question targets a single row data or multi-rows are targeted.

To address this, user intention must be carefully analysed by the Analyst Agent to determine whether a single row or all rows with the same value are expected. Therefore, the Analyst Agent was instructed to explicitly consider the expected output format (single-row or multi-row) to mitigate issues related to top ties:

State if the target data must be a single-row or multi-rows.

However, ambiguity may still persist; therefore, requesting clarification from the user—involving human into the conversational loop— can offer a definitive solution to this issue.

Thirdly, recognizing that the Spider Benchmark employs SQLite databases, a rule was added for the *Query Builder Agent* to strictly adhere to the SQLite dialect:

Always use SQLite dialect.

Finally, since deviation from the original user intent was a recurring issue observed during experiments, the *Refiner Agent* was instructed to adhere strictly to the user’s stated goals in the prompt when correcting execution errors.

#Stick to the User Goals when fixing the query execution error

- **How did task decomposition improve NL2SQL accuracy?**

Decomposing NL2SQL into subtasks and creating customized prompts enabled more effective reasoning over the details of user goals, leading to improved schema linking performance compared to individual model outputs.

As an example, both questions shown below failed in the single-agent experiments but were successfully resolved in the multi-agent setup.

1. Accurate Mapping of *name* to *Song_Name* Column:

Question: Show the *name* and the *release year of the song* by the youngest *singer*.

SINGER						
SINGER_ID	Name	Country	Song_Name	Song_Release_Year	Age	Is_male

“analyst_agent responded: ...

Target Data:

** Song Name*

** Song release year*

schema_linker responded:

Exact Tables and Columns:

** `singer.Song\_Name`*

** `singer.Song\_release\_year`”*

Predicted Query:

SELECT Song_Name, Song_release_year FROM singer

WHERE Age = (SELECT MIN(Age) FROM singer)

Query output: [(‘Love’, ‘2016’)]

2. Accurate Mapping of *template type description* to *Template_Type_Description* Column:

Question: What are the *template ids* with *template type description* "Presentation"

TEMPLATES					
Tem- plate_ID	Version_Num- ber	Tem- plate_Type_Code	Date_Effec- tive_From	Date_Effec- tive_To	Template_De- tails

REF_TEMPLATE_TYPES	
Template_Type_Code	Template_Type_Description

“analyst_agent responded:

****User Goal:**** Retrieve the `Template\_ID` values.

****Target Data:**** `Template\_ID`

****Conditions/Filtering:****

***`Ref\_Template\_Types.Template\_Type\_Description` must be equal to "Presentation".**

schema_linker responded: Here's my reasoning for linking the user's goal to the database schema:

****Exact Tables and Columns:****

*** `Templates.Template\_ID`**

*** `Ref\_Template\_Types.Template\_Type\_Description`**

Predicted Query:

SELECT Templates.Template_ID

FROM Templates JOIN Ref_Template_Types

ON Templates.Template_Type_Code = Ref_Template_Types.Template_Type_Code

WHERE Ref_Template_Types.Template_Type_Description = 'Presentation'

Query output: [(6,), (7,), (10,)]

4.2.2. Context Awareness

Throughout this work, it has been emphasized that errors often arise from misinterpretation of user intentions or lexical ambiguities in the natural language question. Therefore, improving the Analyst Agent's understanding of the natural language query is

critical for achieving better query results. To facilitate this, contextual data was provided to the Analyst Agent to assist with understanding string literals and provides greater insight into the structured data, enabling more comprehensive analysis.

The Autogen Framework's vector stores and retrievers were utilized to extract the most relevant data from the data files. Our objective was to increase the likelihood of incorporating string literals from the reference data to aid the agent's reasoning process.

The question below serves as an example where this method proves successful:

Question: Find the package choice and series name of the TV channel that has high definition TV.

Reference Context (A part of the reference data):

-- Table: TV_Channel --

<i>id</i>	<i>series_name</i>	<i>Country</i>	<i>Pixel_aspect_ratio_PAR</i>	<i>Hight_definition_TV</i>
707	MTV Dance	United Kingdom	16:9	<i>no</i>
708	MTV Music	Italy	16:9	<i>no</i>
709	MTV Live HD	Poland	16:9	<i>yes</i>

“analyst_agent responded:

****User Goals:****

* Retrieve the package choice and series name

****Conditions/Filtering:****

* `Hight\_definition\_TV` must be "yes"

****String Literals:****

* "yes" ”

Predicted Query:

```
SELECT Package_Option, series_name
FROM TV_Channel
WHERE Hight_definition_TV = 'yes'
```

In the single-agent experiments, all three models incorrectly constructed the query using the literal “Yes”, as noted in the previous sub-chapter, resulting in failed query outputs. By leveraging contextual data, the multi-agent system correctly generated the query using “yes”.

However, this approach proved ineffective for questions open to multiple interpretations. Such ambiguous cases still necessitate user clarification. For example, as

detailed in the previous chapter, all three models hallucinated non-existent property codes—such as *hse* for house and *apt* for apartment—when generating the query corresponding to the Spider question below. Although the multi-agent system successfully extracted the correct string literals—“House” and “Apartment”—from the contextual data, it still produced an incorrect query due to the inherent ambiguity in the natural language question, as explained below:

Question: What are the names of properties that are either houses or apartments with more than 1 room?

User goal can be interpreted in two ways:

- User asks for the names of houses with more than 1 room or apartments with more than 1 room.

Corresponding Query:

```
SELECT property_name FROM Properties
WHERE (property_type_code = 'House' OR property_type_code = 'Apartment'
) AND room_count > 1;
```

- User asks for the names of houses with any number of rooms or apartments with more than 1 room.

Corresponding Query:

```
SELECT property_name
FROM Properties WHERE property_type_code = "House" UNION
SELECT property_name FROM Properties
WHERE property_type_code = "Apartment" AND room_count > 1
```

Both the base models and our multi-agent approach interpreted the user goal differently than Spider’s expected outcome and failed.

It is believed by the authors of this study that this stems from ambiguities or incompleteness in the user's question, rather than from a lack of understanding on the part of the models. Seeking user clarification is a potential solution. Because further training on similar ambiguous question-query pairs carries the risk of introducing bias and leading to overfitting to one certain interpretation of all other ambiguous questions.

Another observation from the experiments was that increasing the length of the contextual data led to a decrease in overall accuracy. This suggests that providing more information than necessary may confuse the model, causing it to lose focus on the core

intent of the question. Therefore, the contextual data is limited to only the top retrieved document, which holds the highest similarity score.

As a final note, the retriever may occasionally fetch incorrect contextual data—for instance, the value “north america” might be stored in one table for Continent column, but the retriever could mistakenly source it from a different table where the continent information appears as “North America.”. Such discrepancies can mislead the model and lead to inaccurate query generation, as illustrated below:

<i>Retriever responded: ...</i>								
SLV	El Salvador	<u>North America</u>	Central America	21041.0	1841	6276000	69.7	...
SPM	Saint Pierre and Miquelon	<u>North America</u>	North America	242.0	NULL	7000		...
<u>Predicted Query:</u>								
<i>SELECT SUM(Population) AS Total_Population, AVG(SurfaceArea) AS Average_Area FROM country WHERE Continent = '<u>North America</u>' AND SurfaceArea > 3000</i>								
<u>Gold Query:</u>								
<i>select sum(population) , avg(surfacearea) from country where continent = "<u>north america</u>" and surfacearea > 3000</i>								

4.2.3. Self-Correction

Within the conversational framework, the *Refiner Agent* is tasked with executing predicted queries and correcting execution errors. To prevent the Refiner Agent from repeating responses and becoming stuck in a loop, the repetition penalty was increased from the default 1.0 to 1.8.

Table 4.4. Number of execution errors comparing base models with context aware multi-agent system

	Mistral Nemo	Qwen 2.5 Instruct	Gemma3-IT	MAS
#EXEC ERR	58	31	33	0

Additionally, the agent's prompt was revised to explicitly instruct it to avoid repeating the same query and to explore alternative approaches for correction. As demonstrated in Table 4.4, this successfully reduced execution errors to zero.

During the early stage of this study, the agent had been provided with the database schema, SQL query, and related error messages alone, instructing it to focus solely on correcting syntax errors. However, it was observed that fixing syntax alone would not guarantee semantic correctness or accurate results. Further refinement revealed that syntactic corrections could inadvertently alter the query's meaning, potentially deviating from the user's intended goals and leading to inaccurate results.

As an example, the 'continent = europe' condition had been removed from the query by *Refiner Agent* to ensure query execution in the scenario demonstrated below. However, this fix had altered inadvertently the semantic meaning and resulted in a query that didn't fulfill the user's goals.

Question / Spider: What are the names of all European countries with at least 3 manufacturers?

Predicted Query:

```
SELECT CountryName FROM countries
WHERE Continent = (
    SELECT ContId FROM continents
    WHERE Continent = 'europe'
) HAVING COUNT(DISTINCT Maker) >= 3
```

Execution Error: HAVING clause on a non-aggregate query

Corrected Query:

```
SELECT c.CountryName FROM countries c
JOIN car_makers cm ON c.CountryId = cm.Country
GROUP BY c.CountryName
HAVING COUNT(DISTINCT cm.Maker) >= 3;
```

To improve the Refiner Agent's output and avoid deviation from user's initial intentions, we've modified *Refiner* prompting to ensure strict adherence to user goals during query correction as detailed in Section 4.2.1. This adjustment has resulted in an approximate 1% performance gain.

Without the assistance of the Refiner Agent, the multi-agent system—comprising only three intelligent agents powered by Qwen2.5 Instruct and Gemma3-IT—would have produced 14 queries with execution errors. This indicates that the Refiner Agent contributed to correcting all 14 of these query execution errors. Therefore, an improvement in the number of execution errors would have been observed even without the Refiner Agent.

However, only half of the 14 query outputs matched the gold results, primarily those classified as simple or medium in complexity. While the *Refiner* successfully corrected the execution errors in complex queries, the executed outputs still returned incorrect results due to semantic structural errors. Thus, it can be said that execution errors were helped to be corrected by self-correction, but the contribution to overall accuracy still needs to be improved.

4.2.4. Model Bias

Single-agent experiments highlighted four aforementioned issues: execution errors, a lack of data context, challenges with schema linking and the impact of model biases. Providing the system with contextual data helped to correct incorrect conditional checks. Separating the analysis, schema linking and query building subtasks improved schema linking accuracy.

On the other hand, model biases, where the models tend to default to statistically common interpretations, were hard to mitigate or avoid. For instance, models often exhibit a tendency to automatically generate the clause *LIMIT 1* when encountering phrases such as “the least” or “the first” in the input prompt. This behavior occurs even in cases where the user's intent does not explicitly require restricting the result set to a single row.

Similarly, when performing on the aforementioned Spider Question, models had inaccurately interpreted the *average* word to *AVG()* function, high likely because in their training data *average word is strongly associated to AVG() function*. A similar misinterpretation was observed in the multi-agent experiments as well. The word, referred to the column named *average* in the *stadium* table, mistakenly interpreted it as the average of the *capacity* column by the analyst agent.

“analyst_agent responded:

User Goals:

** Determine the maximum capacity of all stadiums.*

** Calculate the average of all stadiums.*

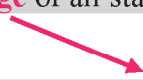
Target Data:

** Maximum capacity (single value)*

** Average capacity (single value)*

This response led to the incorrect query generation as illustrated below:

Question / Spider: What is the maximum capacity and **the average** of all stadiums?



STADIUM						
STADIUM_ID	Location	Name	Capacity	Highest	Lowest	Average

Although adding an instruction rule targeting model biases or introducing an additional agent tasked with correcting such biases was considered to mitigate these issues. However, it was ultimately not implemented due to concerns that it might adversely affect performance on other natural language to SQL tasks.

Training models on more diverse datasets represents a potential avenue for reducing these biases. Furthermore, multi-agent systems offer the advantage of incorporating human intervention to clarify user intent and resolve ambiguities within the conversation.

4.3. Baselines Comparison

Consistent with the methodology described in Chapter 3, the execution accuracy of our multi-agent setup was compared to that of two baseline studies, C3-SQL and DIN-SQL studies. Our context-aware multi-agent system achieved competitive results for Easy and Medium-level queries, demonstrating strong performance with relatively small open-source language models as presented in Table 4.5.

However, accuracy for Hard and Extra Hard-level queries remains significantly lower than that observed in DIN-SQL and C3-SQL. Further investigation is warranted to understand the reasons for this discrepancy and to enhance performance on these more complex query types. As clearly seen in the figures, complex queries remain a significant

challenge, with this study showing only a slight improvement in that area.

Dividing tasks based on additional criteria, performing semantic search not only on the natural language question but also on the most critical keywords, and incorporating customized prompts for complex queries—both hard and extra hard—may help improve accuracy. These strategies, though not fully explored in this work, warrant further investigation in future studies.

Table 4.5. Context aware multi-agent system vs DIN-SQL vs C3-SQL execution accuracy

	Easy (248)	Medium (446)	Hard (174)	Extra Hard (166)	All (1034)
DIN-SQL	92.3%	87.4%	76.4%	62.7%	82.8%
C3-SQL	92.7%	85.0%	77.6%	62.0%	81.9%
Experiment Result	92.7%	90.6%	67.8%	53.6%	81.3%



Figure 4.5. Context aware multi-agent system comparing to DIN-SQL and C3-SQL

CHAPTER 5

CONCLUSION

This study focuses on leveraging mid-sized, open-source models in a collaborative framework to address NL2SQL tasks. An aim is set to determine whether comparable performance can be achieved by this approach relative to studies and experiments utilizing larger foundation models, DIN SQL and C3 SQL.

Microsoft Autogen was leveraged as the multi-agent framework, with Gemma3-IT 12B (Google), Qwen2.5 14B Instruct (Alibaba), and Mistral Nemo Instruct 12B (Mistral AI) utilized as base models. To address the NL2SQL task, the task was decomposed into four subtasks—natural language question analysis, schema linking, query construction, and error refinement—and each subtask was assigned to an intelligent agent, each powered by a model customized with targeted prompts.

To enhance the accuracy of the multi-agent system, Autogen's Retriever Engine was utilized to perform semantic search and provide the system with highly data chunks. This approach minimizes value mismatches during data filtering and significantly improves overall accuracy.

Beyond the lack of data awareness, our initial single-agent experiments revealed three additional challenges: syntax errors, incorrect schema linking and model bias. While task decomposition proved effective in mitigating syntax and schema linking errors, further research is needed to address issues stemming from model bias.

Our Context-Aware Multi-Agent System outperformed all three base models, achieving the highest execution accuracy of 81.3%. This represents a 5.7% improvement over Gemma3 IT, 6.3% over Qwen2.5 Instruct, and 12.8% over Mistral-Nemo Instruct, demonstrating that leveraging multiple LLMs jointly can result in a performance uplift.

This study also achieved competitive results compared to C3-SQL and DIN-SQL, which was selected as baseline methods. While our execution accuracy was slightly lower—by just 0.6% compared to C3-SQL and 1.5% compared to DIN-SQL—the margin was small, indicating the effectiveness of our approach.

REFERENCES

- Bang, Seunghwan, and Hwanjun Song. 2025. "LLM-based User Profile Management for Recommender System." <https://doi.org/10.48550/arXiv.2502.14541>.
- Chang, Shuaichen, and Eric Fosler-Lussier. 2023. "How to Prompt LLMs for Text-to-SQL: A Study in Zero-shot, Single-domain, and Cross-domain Settings." <https://doi.org/10.48550/arXiv.2305.11853>.
- Databricks. 2024. "AI Functions." *Databricks Documentation*. Accessed July 6, 2025. <https://docs.databricks.com/aws/en/large-language-models/ai-functions.html>.
- Dong, Xuemci, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Lu Chen, Jinshu Lin, and Dongfang Lou. 2023. "C3: Zero-shot Text-to-SQL with ChatGPT." <https://doi.org/10.48550/arXiv.2307.07306>.
- Du, Yilun, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2023. "Improving Factuality and Reasoning in Language Models through Multiagent Debate." <https://doi.org/10.48550/arXiv.2305.14325>.
- Gao, Dawei, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. "Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation." <https://arxiv.org/abs/2308.15363>.
- Gao, Yingqi, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, et al. 2024. "XiYan-SQL: A Multi-Generator Ensemble Framework for Text-to-SQL." <https://doi.org/10.48550/arXiv.2411.08599>.
- Hong, Zijin, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2024. "Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL." <https://doi.org/10.48550/arXiv.2406.08426>.
- Li, Boyan, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025. "Alpha-SQL: Zero-Shot Text-to-SQL using Monte Carlo Tree Search." <https://doi.org/10.48550/arXiv.2502.17248>.
- Li, Fei, and H. V. Jagadish. 2014. "Constructing an Interactive Natural Language Interface for Relational Databases." In *Proceedings of the VLDB Endowment*, 73-84. <https://doi.org/10.14778/2735461.2735468>.
- Li, Jinyang, Binyuan Hui, Ge Qu, Jiayi Yang, Binhua Li, Bowen Li, Bailin Wang, et al. 2023. "Can LLM Already Serve as a Database Interface? A BIG Bench for Large-Scale Database Grounded Text-to-SQLs." <https://doi.org/10.48550/arXiv.2305.03111>.
- Lian, Jinqing, Xinyi Liu, Yingxia Shao, Yang Dong, Ming Wang, Wei Zhang, Tianqi Wan, Ming Dong, and Hailin Yan. 2024. "ChatBI: Towards Natural Language to Complex Business Intelligence SQL." <https://doi.org/10.48550/arXiv.2405.00527>.

- Liu, Xinyu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2024. "A Survey of NL2SQL with Large Language Models: Where are we, and where are we going?" <https://arxiv.org/abs/2408.05109>.
- Mahmud, Tanzim, Azharul Hasan, Mahtab Ahmed, and Ching Chak. 2015. "A Rule Based Approach for NLP Based Query Processing." In *2015 International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*, 78–82. <https://ieeexplore.ieee.org/document/7391926>.
- Mavroudis, Vasilios. 2024. "LangChain". November. <https://doi.org/10.20944/preprints202411.0566.v1>.
- Pourreza, Mohammadreza, and Davood Rafiei. 2023. "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction." *arXiv preprint arXiv:2304.11015*. <https://arxiv.org/pdf/2304.11015>.
- Pourreza, Mohammadreza, and Davood Rafiei. 2023. "Evaluating Cross-Domain Text-to-SQL Models and Benchmarks." <https://doi.org/10.48550/arXiv.2310.18538>.
- Pourreza, Mohammadreza, Ruoxi Sun, Hailong Li, Lesly Miculicich, Tomas Pfister, and Sercan Ö. Arik. 2024. "SQL-GEN: Bridging the Dialect Gap for Text-to-SQL Via Synthetic Data And Model Merging." *arXiv preprint arXiv:2408.12733*. <https://doi.org/10.48550/arXiv.2408.12733>.
- Qi, Jiexing, Jingyao Tang, Ziwei He, Xiangpeng Wan, Yu Cheng, Chenghu Zhou, Xinbing Wang, Quanshi Zhang, and Zhouhan Lin. 2022. "RASAT: Integrating Relational Structures into Pretrained Seq2Seq Model for Text-to-SQL." <https://arxiv.org/pdf/2205.06983>.
- Rajkumar, Nitarshan, Raymond Li, and Dzmitry Bahdanau. 2022. "Evaluating the Text-to-SQL Capabilities of Large Language Models". <https://arxiv.org/pdf/2204.00498>.
- Salesforce. 2020. *Introducing Salesforce Einstein Automate*. December 2, 2020. <https://www.salesforce.com/news/press-releases/2020/12/02/introducing-salesforce-einstein-automate/>.
- Shao, Zhihui, Shubin Cai, Rongsheng Lin, and Zhong Ming. 2025. "Enhancing Text-to-SQL with Question Classification and Multi-Agent Collaboration." <https://doi.org/10.18653/v1/2025.findings-naacl.245>.
- StackOverflow. 2024. "Stack Overflow Developer Survey 2024." *Stack Overflow*. Accessed July 6, 2025. <https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages>.
- Suhr, Alane, Ming-Wei Chang, Peter Shaw, and Kenton Lee. 2020. *Exploring Unexplored Generalization Challenges for Cross-Database Semantic Parsing*. Association for Computational Linguistics, 8372-8388. <https://aclanthology.org/2020.acl-main.742.pdf>.

- Topsakal, Oguzhan, and T. Cetin Akinci. 2023. "Creating Large Language Model Applications Utilizing LangChain: A Primer on Developing LLM Apps Fast." In *International Conference on Applied Engineering and Natural Sciences*, 1050–56.
- Wang, Bing, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, et al. 2025. "MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL." <https://doi.org/10.48550/arXiv.2312.11242>.
- Woods, W. A. 1973. "Progress in Natural Language Understanding." *National Computer Conference and Exposition on - AFIPS 73*.
- Wu, Qingyun, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, et al. 2023. "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation." <https://doi.org/10.48550/arXiv.2308.08155>.
- Xiao, Chunyang, Marc Dymetman, and Claire Gardent. 2016. "Sequence-based Structured Prediction for Semantic Parsing." 1341-1350. <https://aclanthology.org/P16-1127.pdf>.
- Yu, Tao, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, et al. 2018. "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task." *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics. 3911–3921. <https://doi.org/10.18653/v1/D18-1425>.
- Zhong, Ruiqi, Tao Yu, and Dan Klein. 2020. "Semantic Evaluation for Text-to-SQL with Distilled Test Suites." <https://doi.org/10.18653/v1/2020.emnlp-main.29>.
- Zhu, Xiaohu, Qian Li, Lizhen Cui, and Yongkang Liu. 2024. "Large Language Model Enhanced Text-to-SQL Generation: A Survey." <https://arxiv.org/abs/2410.06011>.