CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Error and Reliability Analysis of Open-Source LLMs for Text-to-SQL Generation Across Query Complexities

Master's Thesis in Computer Science and Engineering

Mojtaba Alizade
Omar Younes

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's Thesis 2026

Error and Reliability Analysis of Open-Source LLMs for Text-to-SQL Generation Across Query Complexities

Mojtaba Alizade
Omar Younes

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Error and Reliability Analysis of Open-Source LLMs for Text-to-SQL Generation Across Query Complexities
Mojtaba Alizade & Omar Younes

Supervisor: Bengt Haraldsson & Miroslaw Staron, Department of Computer Science and Engineering
Examiner: Francisco Gomes de Oliveira Neto, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Mojtaba Alizade & Omar Younes
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Retrieving the correct data from databases quickly and accurately using Structured Query Language (SQL) is a crucial task in Software Engineering, but one that is time-consuming if done manually, and complicated to navigate if done through an application. However, asking a specific question about the database in natural language and using a Large Language Model (LLM) to generate a SQL query that retrieves the desired data is more time-efficient and intuitive for practitioners. This Text-to-SQL process is nonetheless difficult for LLMs, as it requires high-reasoning and domain-knowledge capabilities. This study aims to gain a better understanding for failures of LLMs in this context by exploring how the complexity of a Text-to-SQL task, and the active parameter count of LLMs, affect the type of failures, how often they occur, and how consistent the failure rates are. To achieve this, three randomly sampled sets of Text-to-SQL questions of different complexities from the BIRD and LiveSQLBench-Base-Lite datasets are run on the Qwen3 A3B, Qwen3 A22B and Qwen3 A35B LLMs via Amazon Bedrock using a two-step prompting strategy. The results are first analyzed descriptively and then statistically tested. The results show that selecting the correct tables to use in the generated query is the most common subcategory of failure observed across LLMs, that increases in size between the selected LLMs do not significantly affect performance on any complexity, and that none of the examined LLMs are significantly different in failure reliability across complexities. With that being said, all three LLMs achieve absolute failure consistency, in at least 76% of the Text-to-SQL questions, indicating that failures are likely systematic due to an inability to produce a correct answer.

Keywords: Empirical software engineering, Quasi-experiment, Text-to-SQL, Large Language Models, BIRD dataset, Query complexity, Statistical analysis

Acknowledgements

We would like to thank our supervisors Bengt Haraldsson and Prof. Miroslaw Staron for their support and guidance during our Master's Thesis project. We would also like to thank our examiner Prof. Francisco Gomes de Oliveira Neto for the constructive feedback and guidance regarding the best steps forward during our meeting. Finally, we express our gratitude to Chalmers University of Technology for providing the academic environment that made this research possible.

Mojtaba Alizade & Omar Younes, Gothenburg, June 2026

Contents

List of Figures

List of Tables

1

Introduction

Large Language Models (LLMs) have rapidly advanced in capability, making it feasible for organizations and inexperienced users to interact with complex data platforms using natural language instead of technical query languages [1], [2], [3]. LLMs traditionally receive a user question, interpret it based on the design of the model and use internal tools in order to formulate a coherent output. Current LLM implementations that are relevant to this study demonstrate this workflow in the context of transforming *Natural Language Queries* (NLQs) to *Structured Query Language* (SQL) in a process referred to as *Text-to-SQL* [4].

The utilization of LLMs in Text-to-SQL tasks has the potential to significantly lower the barrier to database query formulation, especially when translating NLQs into queries over complex database tables, giving them a practical use case in developers' workflows [1]. This challenge in query formulation mirrors how developers interact with SQL in practice. A study on student developers shows that converting thought-solutions into SQL code requires more effort than the initial stages of data planning and presenting the solution in natural language [5]. Additionally, these students have demonstrated better problem-solving performance when describing the solution in natural language than during the conversion step [5]. This gap between the phases suggests that adopting Text-to-SQL systems can mitigate the pressure caused by writing SQL code and allow junior developers to leverage their problem-solving skills to express their intent in the form of NLQs.

Beyond developers, Text-to-SQL leverages a familiar communication channel, i.e. natural language, and executes a technical task when requested [1], [6]. LLMs used within Text-to-SQL systems can therefore aid with the process of decision-making, and precise data analysis or retrieval, which is helpful in many daily practices for several industries that involve nontechnical users accessing essential data [1], [6].

One such use case is within the medical field. When handling digitized medical records, physicians spend up to 35% of their working hours on documentation and review of patient records [7]. The process often requires searching for specific pieces of information across multiple graphical user interfaces while keeping fragmented data in mind, and using them to either perform a procedure, or input them manually

when ordering certain medical tests [8]. This cognitive overload may contribute to diagnostic errors as Singh *et al.* [9] found that 15.3% of delayed diagnoses were caused by failure to review previous documentation. This highlights the potential benefits of integrating Text-to-SQL with medical records for patient information retrieval using natural language instead of manual navigation, especially to help mitigate risks associated with fragmented data retrieval.

The level of difficulty when translating NLQs to SQL queries varies, and is referred to as *query complexity*. Recent Text-to-SQL papers report low accuracy scores for LLMs that attempt to construct complex SQL queries from NLQs [10], [11], [12]. When model outputs are incorrect, research has identified distinct failure modes that are associated with different sources of error in the generated SQL query. Baig *et al.* [13] conducted a systematic literature review and reported *syntactic* and *semantic* errors as the two primary failure modes within Text-to-SQL. This has been supported by later Text-to-SQL experimentation work [14]. However, Peng *et al.* [15] introduce *execution* errors as an additional category that captures cases where syntactically correct queries produce unexpected results when executed. This study adopts syntactic and semantic failure modes, but renames execution errors as *hallucination* in order to more accurately reflect cases captured by the mode's redefined dimensions. The process of classification for each of the failure modes is presented later in Chapter 4. Despite advances in the Text-to-SQL field, various failure modes remain prevalent, and more research regarding the causes of these failures is required in order to mitigate them. Additionally, research on which models are the most reliable for use cases of different complexity helps align resource demands with adequately powerful models since resource demands of larger LLMs are increasingly difficult to meet.

1.1 Problem Description

Existing experimental research on Text-to-SQL highlights that the reliability of models in generating helpful SQL queries is negatively correlated with database size and complexity [11], [12], [16]. This indicates that the performance of models decreases as databases mirror real-world relational data modeling.

In order for future research to investigate effective solutions that improve the performance of LLMs in Text-to-SQL tasks, more insightful reasoning about failure modes of tasks is required. Current research does not systematically analyze failure modes in relation to the utilized models and queries. Performance is often measured solely via accuracy percentages, overemphasizing static results, and omitting other possible contributing factors or correlated variables that would aid in interpreting accuracy metrics.

In summary, understanding which conditions lead to failure or success is essential for interpreting the performance correctly. By exploring the relationship between failure modes and other variables such as query complexity and repeated-run reliability, this study supports accuracy and reliability measures for more meaningful model

comparisons, and ultimately more advanced Text-to-SQL systems.

1.2 Purpose of the Study

The study analyzes the performance and categorizes the outcomes of open-source LLMs from the same family in a systematic experimental research setting, comparing models of varying parameter sizes on databases that replicate real-world relational structures. The purpose is to:

- Draw conclusions regarding how models fail in comparison to other models through the types of failures, as well as through exploring accuracy and reliability.
- Investigate how model size impacts performance on Text-to-SQL tasks across different query complexities, and derive implications regarding model selection under relevant resource and reliability constraints.
- Gain an understanding for the reliability of open-source LLMs in Text-to-SQL contexts.

1.3 Significance of the Study

The study explores why open-source LLMs output SQL queries with certain failures when prompted. Rather than relying solely on aggregate accuracy percentages, the study investigates failure modes, model reliability, and the effects of model size and query complexity, making the results more pragmatic for interpreting performance in the real world. This is essential for both researchers and practitioners, since a model's performance can include both accuracy and reliability measures.

For researchers, the study offers a more fine-grained evaluation of Text-to-SQL systems and therefore supports future works on analyzing failures and improving the current systems. For practitioners, the results can inform model selection by presenting how the different models perform under varying level of query complexity and resource constraints.

The practical implementations of this work extend to domains where natural language offers a convenient method of access to relational databases that would improve efficiency and support decision-making. For example, domains such as healthcare, finance and education are ones where Text-to-SQL systems can help users retrieve information more intuitively, accurately and efficiently, reducing concerns regarding manual data querying and connection while improving the access of data in practice [1], [2], [17], [18].

1.4 Research Questions

RQ1: Comparative Error Analysis and Failure Subcategories
How do the most commonly occurring subcategories of failure modes differ between LLMs? This is a step that provides more details about the syntactic, semantic, or hallucination failure modes for each of the models, aiming to gain insights into how different sources of failures emerge across increasing model sizes.

RQ2: Impact of Model Size on Text-to-SQL Accuracy
For each query complexity, how do the Text-to-SQL accuracies differ between LLMs of larger and smaller sizes? This is an important question that helps with understanding the impact of model size on success rates. Specifically, investigating this effect helps determine if the use of models with larger model sizes is justified by significant performance gains, or if smaller models are adequate for various Text-to-SQL tasks.

RQ3: Impact of Query Complexity on Failure Consistency
For each of the analyzed LLMs, how are the consistencies of test case failures affected as query complexity increases? LLMs can provide varying outputs for the same input. Therefore, this question aims to investigate how this behavior changes as query complexity increases by analyzing how reliably do repeated executions of the same Text-to-SQL task lead to the same outcome.

1.5 Thesis Outline

The following table presents an overview of the structure for this study with short explanations regarding each of the chapters.

Chapter	Relevance
Background	Provides prerequisite information and terms for the rest of the study.
Related Work	Presents relevant studies and positions this study.
Methods	Presents the methods used in this study and provides an evaluation of the study's validity at the planning stage.
Results	Presents the results of the study.
Discussion	Presents a discussion of the study's results, some exploration of possible reasons why, and identified threats to the validity of the study.
Conclusion	Presents conclusions drawn from the results of the study.

2

Background

2.1 Natural Language to SQL

Natural Language Processing (NLP) enables computers to understand, interpret, and generate natural language. It has many applications in various fields such as the medical field, machine translation, text summarization, and query answering. NLP was developed to simplify user interaction with computers, in particular, by helping users communicate naturally, since not all users are proficient in machine-specific languages and may not have the time or resources to learn new programming languages or become proficient in existing ones [19].

One of the uses of NLP is to interpret NLQs. An NLQ is a query expressed in everyday natural language rather than in a formal or structured query language. The goal of NLP in this context is to understand the user's intent and translate the query into a form that a computer system can process, such as a search request or a database query [20].

Text-to-SQL systems are designed to translate NLQs into executable SQL statements [21]. From a computational perspective, turning NLQs into SQL queries means converting an informal natural language into a structured query that a database can understand and execute [22]. This requires understanding the user's intent and accurately connecting that request to the structure of the database, such as its schema and fields [23].

SQL is a database query language that provides high flexibility along with strong functionality for database systems. It is widely used in different fields, such as software development, finance, medicine, or education, in order to create, delete, update, and retrieve data. However, non-technical users often find it challenging to learn SQL. Even for technical users, writing complex and correct SQL queries in different databases and table schemas can be difficult [4].

These challenges place Text-to-SQL as a challenging semantic parsing task, where interpreting user intent and accurately generating structured and executable queries are critical [23].

2.2 LLMs for Text-to-SQL

This section examines the role of LLMs in Text-to-SQL systems, with a focus on their development, generation variability, and the impact of model size through active parameters in Mixture-of-Experts architectures.

2.2.1 Evolution of Text-to-SQL Systems

Early Text-to-SQL systems used a rule-based approach for their implementations. In this approach, grammatical rules were predefined and strategies to turn NLQs into SQL queries were designed manually [24]. Even though these systems performed well for simple cases, they struggled with handling complex queries and databases with different structures and schemas [25].

The introduction of deep neural networks greatly advanced Text-to-SQL systems through large models for the interpretation of natural language and generating SQL queries. These models became better at working with new unseen databases by learning how NLQs relate to the structure of the database, making them more flexible than earlier approaches [25].

More recently, advances in LLMs have significantly improved the performance of Text-to-SQL systems, mostly due to improved natural language understanding and code generation capabilities [21]. LLMs are pretrained on large and diverse datasets, which allows them to learn linguistic patterns, have strong reasoning abilities, and capture the complex relationships between NLQs and structured database schemas. As a result, LLM-based Text-to-SQL systems can generate accurate SQL queries and generalize to unseen database schemas with minimal task-specific training [25].

Figure 2.1 presents the process of a Text-to-SQL system. The NLQ of a user is processed by an LLM with the help of the database schema, translated into an executable SQL statement, and executed on the database to produce the final result.

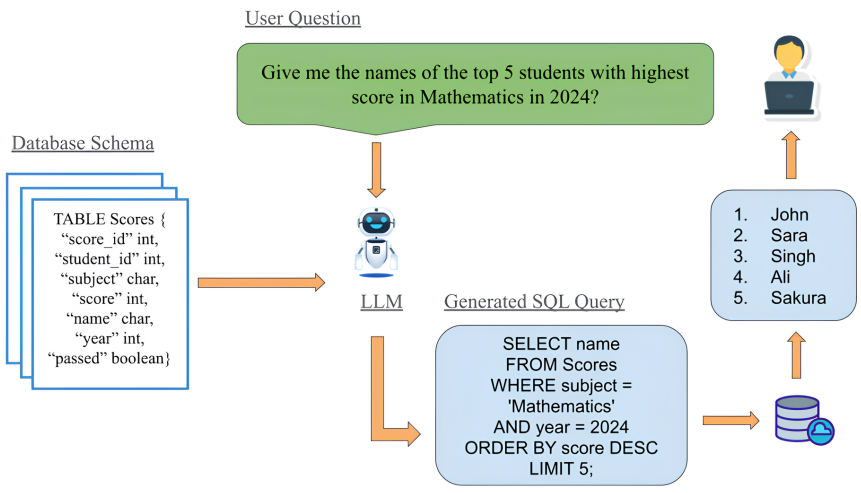

Figure 2.1: Overview of an LLM-based Text-to-SQL system.

2.2.2 Stochasticity of LLMs

Despite these advances, LLM-based Text-to-SQL systems are facing challenges. For example, LLM-based systems may be inconsistent in their responses when asked to generate SQL statements [23]. This behavior comes from the stochastic nature of LLM decoding.

During generation, LLMs determine the possibility score for various possible words and samples from this distribution [26]. The randomness level for these sampled words is determined by a *temperature* value, which is also thought to be the primary factor for randomness [27]. Low temperature values, for example ones close to zero, make the model more deterministic, which leads the model to select the word with the highest probability almost every time and hence makes the output more predictable [26]. Higher temperature values, for example close to one, increase the diversity of sampled tokens, making the model's outputs more varied and potentially more creative, but also less predictable.

However, it is important to note that even setting the temperature to zero does not guarantee identical results across different generation attempts. The internal workings of LLMs and their decision-making process are too complex and difficult to interpret [13]. Many other factors can still introduce variability. As a result, LLMs may produce different SQL queries for the same input, where some are correct, while others contain syntactic, hallucination, or semantic failures.

This randomness gives rise to significant concerns regarding the reliability of such Text-to-SQL systems using LLMs, particularly in applications where consistent and correct query generation is critical. Therefore, understanding whether model failures are systematic or stochastic is important in order to evaluate the robustness and practical usability of these systems.

2.2.3 Active Parameters in Mixture-of-Experts LLMs

Mixture-of-Experts (MoE) architectures introduce conditional computation into transformer models by activating only a subset of parameters for each input. Unlike dense models, where all parameters are used for every token, MoE layers consist of multiple expert networks and a routing mechanism that selects a small number of experts per token. This idea was introduced by Shazeer *et al.* [28], who proposed sparsely-gated MoE layers where a gating network activates only a few experts per input sample. Thus, the number of active parameters during inference is much lower than the overall number of parameters in the model.

This design creates a distinction between total parameters and active parameters. The total parameter count includes all experts in the model, while active parameters refer only to those involved in processing a given input. As shown in the Switch Transformer, this allows models to scale to very large sizes while keeping the computational cost per token approximately constant [29].

Figure 2.2 illustrates the concept of sparse activation in an MoE model. Given an

input sequence, the routing mechanism selects only a subset of available experts based on the characteristics of the input. In this example, the question "Why is my Python loop skipping the first element" is mainly related to programming syntax and logical reasoning. Therefore, the Syntax & Languages and Logic experts are activated, while less relevant experts such as Data Structure and Inheritance remain inactive.

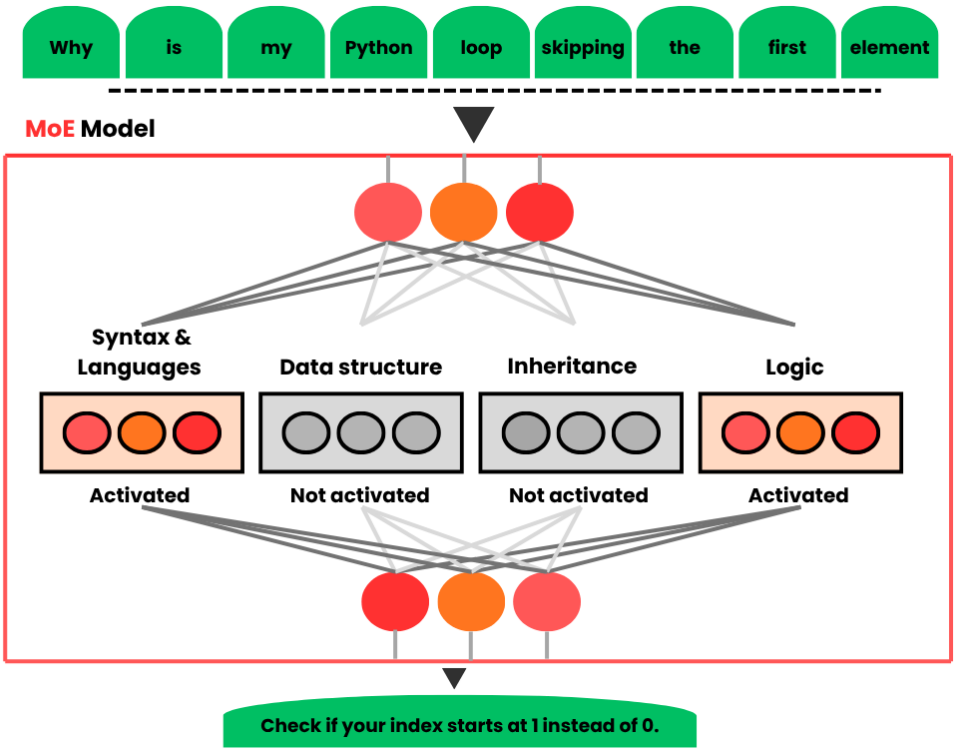

Figure 2.2: Illustration of sparse activation in a Mixture-of-Experts model.

In this context, the concept of active parameters can be used to describe the number of those parameters in a model that are used while analyzing a certain input token. Even if a model contains tens of billions of parameters, the number of active parameters per token can be significantly lower, due to the sparse routing mechanism. Therefore, active parameters offer a better depiction of the computational capability and cost of the model when running.

The MoE architecture is suitable for Text-to-SQL transformations as this process involves a variety of subtasks such as natural language understanding, database schema parsing, and SQL generation. In a recent survey on LLM-based Text-to-SQL systems, MoE is discussed as a method which uses multiple experts in the transformation process, and SQL-GEN is mentioned as a framework using MoE architecture to generate multi-dialect SQL [30]. The reason behind implementing this mechanism in SQL-GEN is the difference in syntax and functions among various dialects of SQL like SQLite, PostgreSQL, and BigQuery. To solve this problem, the SQL-GEN employs MoE to merge these dialect models in a single system, where expert routing helps support different SQL dialects [31]. MoE architecture is used

in models such as DeepSeek-V3, Qwen3 MoE models, and Mistral Large 3. While many popular models, such as Llama 3, GPT-3.5, and Anthropic Claude 3 are dense models where all parameters are used at each run.

2.3 SQL Query Complexity

This section discusses SQL query complexity in Text-to-SQL tasks, including current methods of measuring complexity and its observed influence on model performance.

2.3.1 Query Complexity Types

In the context of Text-to-SQL tasks, query complexity refers to a quantification that classifies SQL queries based on how difficult they are to reason about and generate. There are different approaches for measuring this complexity. Lei *et al.* takes the simplest approach of counting a query's tokens by splitting queries on whitespace characters and assigning a complexity for various intervals of token counts [11]. This approach fails to consider any dimensions of complexity besides token count and therefore does not account for specific statements or additional semantic factors that contribute to query complexity.

Examining the internal structure of queries, Jamil *et al.* [32] recognizes queries as one of the SQL-database objects that affect the total effort required for building various relational database applications. Therefore, when writing SQL queries to combine tables or retrieve data in a database application, Jamil *et al.* defines the effort related to query metrics as a sum of the following counts, each having a specific weight: 1. Total queries written 2. Simple *select* statements 3. Nested *select* statements.

Subali and Rochimah [33] extend previous approaches that rely exclusively on counting simple or nested statements' by attaching weights to attributes within the statements themselves, such as the number of table joins. The complexity is then calculated by adding products of components and their corresponding weights [33]. Using this method, similarly long but differently structured queries are distinguished, since more relevant attributes are considered. Figure 2.3 illustrates underlying structures of the aforementioned query complexity measures, each with an SQL query example.

These methods are objective and quantifiable in terms of measurement, however, they consider the SQL query on its own, since complexity is evaluated from a human perspective. In the context of Text-to-SQL, additional dimensions aside from query complexity are intuitively relevant, such as the NLQ's intent and clarity as well as the complexity of the database schema. This broader outlook is reflected by Li *et al.* [12], who present a complexity metric for Text-to-SQL tasks in which Text-to-SQL domain experts evaluate each task and classify it as *simple*, *moderate*, or *challenging* by assigning a 1-3 score for each of the following dimensions: 1. Question understanding 2. Knowledge reasoning 3. Data complexity 4. SQL complexity. This approach can be considered a subjective categorization of complexity since ex-

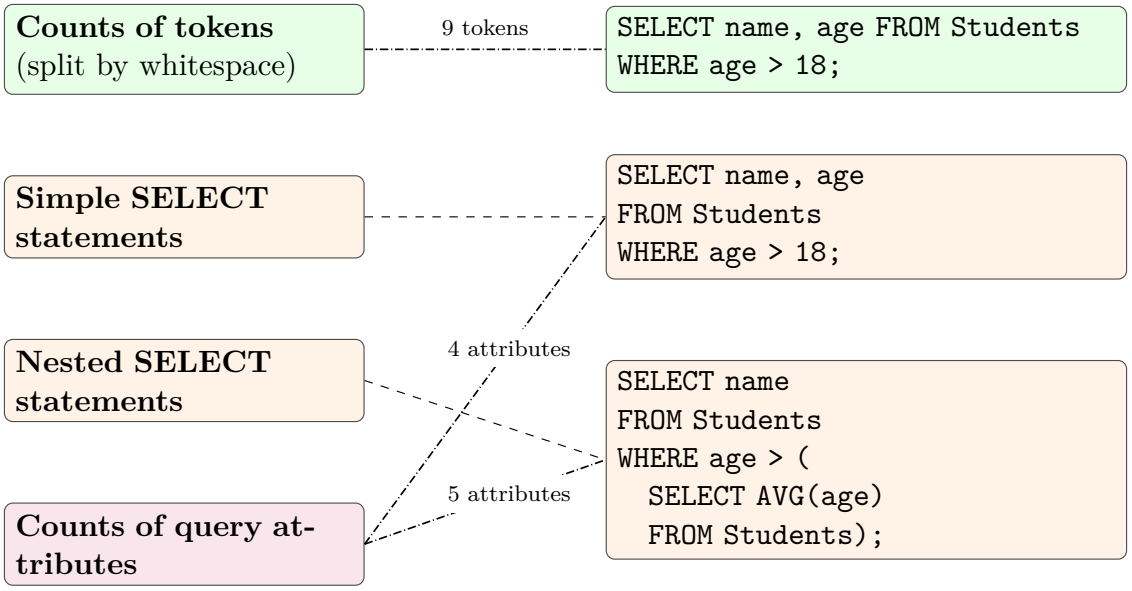

Figure 2.3: Illustration of different units used for measuring query complexity. The left column lists measurement types, while the right column provides corresponding SQL examples. Token-based complexity is computed by splitting the query on whitespace, resulting in nine tokens for the connected example. Attribute-based complexity follows the definition of output variables, input variables, nested queries, join tables, and total tables (including subqueries) outlined by Subali and Rochimah [33]. Thereby, the simple select statement contains four attributes, while the nested query contains five due to the additional subquery.

perts assigning four scores that define task difficulty limits reproducibility for other researchers. However, the inclusion of domain-relevant dimensions provides a more comprehensive categorization of complexity in Text-to-SQL tasks.

2.3.2 Impact on LLM Performance

Across many research papers that utilize schema and query complexity when benchmarking the accuracy of different models on Text-to-SQL tasks, an intuitive pattern emerges. The impact of increased difficulty is reflected on accuracy results [11], [12], [15], [17], [18], despite how complexity is measured. Therefore, schema and query complexities are shown to have significant impact on model performance. Overall, models perform better on simple schemas and flat query structures, such as single-table selections, compared to those involving large schemas, and nested subqueries [34], or arithmetic operations, **GROUP BY** clauses and **JOIN** operations [18], [35]. Although recent advances in LLMs have improved robustness, as mentioned earlier, performance degradation on increased difficulty for complex, cross-domain benchmarks indicates that schema and query complexity continue to be major factors.

2.4 Text-to-SQL Datasets

Text-to-SQL datasets are collections of **.json** objects which typically contain the following properties:

- Natural Language Question (NLQ)
- Golden SQL query (ground truth query)
- Unique question identifier
- Database identifier

In this study, each of these **.json** objects is referred to as a *test case* and each ground truth query for a test case is referred to as the *golden SQL query* for that test case. Datasets vary in size, but comprise traditionally large amounts of test cases, which are split into 3 independent sets: training, validation and test sets. One popular use case is training Text-to-SQL models on the training set, internally validating on the validation set and reserving the test set for a final evaluation or benchmarking on unseen data that datasets do not generally open-source. In this study, since LLMs are benchmarked via API calls and the analysis requires access to the golden queries, only validation sets will be utilized. More information regarding the selection of datasets and LLMs is presented in Sect. 4.2.3.

In terms of a dataset's overall difficulty, advancements in Text-to-SQL models have prompted the creation of more complicated datasets. The goal of developing newer datasets is mainly to address limitations of previous datasets by growing closer to real-world relational databases, and further challenging Text-to-SQL models. One of the first large-scale datasets is *WikiSQL* [36], but queries were limited to one table. The *Spider 1.0* dataset [37] addresses this issue by incorporating cross-domain databases with multiple tables, nested queries and NLQs that require more complicated SQL components, such as **JOIN**, or **ORDER BY**. However, Text-to-SQL models continued to improve performance on the Spider dataset, increasing the percentage of correctly answered NLQs from 53.5% [38], to 85.3% [39] within 3 years. The same model was only able to achieve 50.7% on *BIRD* [12], which featured much larger schemas in comparison to previous datasets and included abbreviated column names. Additionally, the dataset incorporated expert-annotated difficulty classifications for each test case that consider dimensions beyond the golden query's complexity, such as the NLQ's intent and database complexity, as mentioned in Sect. 2.3.1. *Spider 2.0* [11] is considered the most advanced Text-to-SQL dataset since it expands on the scale and complexity of tables per database and of keywords per golden query compared to *BIRD*, targeting more advanced LLM capabilities.

One conclusion that can be reached from the progression of the datasets mentioned is that the overall difficulty of a dataset is mainly determined by how complicated the dataset's tables, NLQs and expected golden queries are. The scale of databases' schemas and them being cross-domain, meaning encompassing different industries

such as healthcare and finance, are additional common ways to distinguish difficult relational databases that reflect enormous, real-world industry usages.

2.5 Evaluation Metrics in Text-to-SQL

The evaluation metrics used in Text-to-SQL can be divided into two groups, Content Matching-Based Metrics and Execution-Based Metrics, each containing two distinct metrics [21].

2.5.1 Content Matching-Based Metrics

The Content Matching-Based Metrics look at the structure of the SQL query generated by the model and compares the similarities to the golden query. This evaluation ensures that the model follows the correct syntax and structure of SQL, even if the model query is not fully optimized for performance or returns the correct data [21], [25].

The first metric here is *Component Matching* (CM), which measures the average number of matches between the model query and the golden query [25]. This metric looks at each component of an SQL query, such as SELECT, FROM, and WHERE, separately. A component is considered correct as long as it corresponds to the same component in the golden query, regardless of the order [21]. This method allows flexibility in the structure of the query while making sure that all necessary elements of an SQL query are properly incorporated and formatted [21], [25].

The other metric is *Exact Matching* (EM), which is a more restrictive metric than CM. This metric does not allow for any differences between the model query and the golden query, whether it is the order of the components or the structure of the query, as both must be identical [1]. However, the downside of this approach is that it penalizes queries that are semantically correct but are structured differently [25].

2.5.2 Execution-Based Metrics

The Execution-Based Metrics evaluate the performance of the model query. This evaluation looks at the result of the model query when running against the database to check the correctness of the results, as well as the efficiency of running the query [21], [25].

Execution Accuracy (EX) is one of the metrics. This metric evaluates whether the model query returns the same result as the golden query when both are executed on the same database. Unlike Exact Matching, EX does not require the generated query to have the same textual form or structure as the golden query. A query can therefore be considered correct under EX if it produces the correct output, even if it uses different SQL clauses, ordering, or an alternative but equivalent formulation [1].

Efficiency Score (ES) is the other metric. This metric compares and measures the

computational runtime and efficiency between the model query and the golden query. This is because even though the model query returns the correct result, it may have introduced unnecessary steps, which makes it inefficient. The score penalizes queries that introduce extra complexity, such as redundant subqueries or unnecessary joins, even when the results match [21], [25].

3

Related Work

3.1 Manual Error Analysis

When analyzing model outputs of failed Text-to-SQL test cases, recent studies rely on random sampling and manual inspection of results in order to complement model performance scores with observed error analysis. Benchmarking datasets [11], [12] as well as a study regarding prompting approaches [39] randomly sample a set of incorrect executions in order to conduct manual error analyses. For example, Pourreza and Rafiei [39] specify that the sampling was done from the training split of the utilized dataset and that the results were placed in manually-defined categories. In some cases, such as some of the error categories observed in Lei *et al.* [11], no subcategories are derived, meaning that the error taxonomies vary in granularity across studies.

Manually defining the categories, which the encountered errors are grouped under, is a shared approach by Lei *et al.* [11] and Li *et al.* [12]. While the results are informative, they remain limited to the specific experiment as they do not differentiate between failure modes clearly. This study addresses these inconsistencies by utilizing the failure modes as the first level in the multi-level error analysis, and placing each encountered error in a suitable subcategory under its failure mode. The process of conducting the error analyses is explained comprehensively in Sect. 4.2.6.

3.2 Consistency of LLM Outputs

Several recent approaches, such as SQL-PaLM, RESDSQL, and DIN-SQL, report strong results on benchmark datasets such as Spider, typically evaluated using EX [39], [40], [41]. However, these studies typically evaluate models using a single execution per test case and report average accuracy metrics across datasets. The consistency of model outputs under multiple generations has received less attention.

Multiple generations or other improvement strategies have been used in many works to improve reliability. For example, SQL-PaLM generates multiple SQL candidates and applies execution-based filtering and selection to improve the accuracy of generated queries [40]. Similarly, DIN-SQL introduces multi-stage reasoning with self-correction to refine model predictions, while RESDSQL improves Text-to-SQL

generation through a multi-stage pipeline that decouples schema linking and SQL skeleton parsing [39], [41]. These approaches show that multiple intermediate or alternative outputs can improve performance, which means that a single generation may be sensitive to variations in decoding processes.

Furthermore, some works explore prompting strategies that leverage database schema information and structural alignment between NLQs and SQL queries. For example, schema-aware prompting methods construct prompts that explicitly link questions, schema elements, and SQL structures to improve generation accuracy [42], while frameworks such as StructGPT enable LLMs to iteratively retrieve and reason over structured data sources when interacting with structured data systems [43]. Although these methods improve syntactic and semantic validity, there is no explicit analysis of how variability changes between multiple executions under identical conditions in the context of Text-to-SQL.

In general, existing research on Text-to-SQL systems has largely focused on consistency as a way to improve accuracy, rather than as a factor in need of systematic study. Few works carefully analyze the distribution of generated SQL queries across multiple generations or explore how stochasticity leads to unstable error patterns. This study addresses this gap by systematically analyzing multiple generations for identical inputs and exploring how consistently LLMs produce the same failure mode.

3.3 Model Size & LLM Performance

Previous studies on LLMs have indicated that increasing model size can improve LLM performance in tasks such as structured output generation and code synthesis [44], [45]. For example, increasing the size of LLMs leads to improved performance and can lead to emergent abilities such as improved reasoning and generalization of complex tasks [46], [47], which is important for tasks such as translating NLQs into correct SQL queries.

For the Text-to-SQL domain, some studies have reported results for models of different sizes, but no specific impact of model size is discussed. More recent work has made use of large pre-trained models such as PaLM-2 as the primary component of systems like SQL-PaLM. This has achieved high performance on benchmarks using few-shot prompting combined with execution-based consistency filtering, demonstrating the effectiveness of LLMs for Text-to-SQL [40]. However, in these works, the focus is generally on peak performance rather than comparisons across a range of model sizes.

Some studies specifically focus on improving performance on smaller or lightweight models. For example, SPS-SQL demonstrates how schema-aware pre-synthesized queries can improve the accuracy of small open-source LLMs on datasets like Spider, achieving performance competitive with models in the 7B parameter range [48]. Similarly, studies on small language models for Text-to-SQL test cases (e.g., 0.5B–1.5B

parameters) show that their performance can be improved through post-training and corrective approaches, allowing them to achieve competitive results compared to larger models [49]. These results demonstrate the ability of smaller models to achieve competitive results with appropriate techniques, but also highlight the lack of understanding of size effects.

Although most of the work focuses on finding the best-performing model, little attention is paid to the relationship between model size and SQL generation quality. As a result, comprehensive empirical studies that compare small- and large language models under unified experimental settings are few. This study aims to bridge this research gap by assessing Text-to-SQL performance across models of different sizes.

3.4 Prompting Strategies & Reasoning

Recent papers regarding prompting strategies have proven to enhance reasoning capabilities of LLMs. One such strategy is the use of *Chain-of-Thought* (CoT) prompting [50], which displays how providing intermediate reasoning steps improves LLM reasoning on multi-level tasks. Instead of simply requesting an answer as the output, the model is encouraged to produce step-by-step explanations. This enables more structured and interpretable outputs for the user while simultaneously guiding the model reach the correct final result. Although the paper evaluates the strategy on arithmetic and symbolic reasoning tasks, this approach has similar implicated benefits for other contexts. The improvements reflect a generalizable pattern for how reasoning of LLMs can be improved in the context of Text-to-SQL, where query generation often requires multi-step reasoning over differently complex schema elements and user intent.

Another strategy is *decomposed* prompting, a general method for handling complex tasks through separate prompts [51]. Instead of relying on a single prompt to produce the desired output, the approach defines a prompt for each distinct job such as reasoning or formatting. These intermediate outputs are later combined to produce the final result. This structure of prompts allows for more fine-grained control over the reasoning process and can improve performance on tasks that are too complex for the model to reason on its own [51].

When connecting CoT to Text-to-SQL, strategies that break up prompts into multiple steps are theoretically relevant. However, distinct steps are not beneficial for Text-to-SQL tasks since outputs from each step would be required for prompting the next step. For example, if the first step is inferring the table names that are relevant to the NLQ, the next step of formulating the query requires table names, which a sequential execution of the decomposition strategy would achieve. Therefore, Zhou *et al.* [52] introduce *least-to-most* prompting, a modification of decomposed prompting, where steps are solved sequentially in increasing order of complexity. The strategy separates a given question into multiple simpler questions, with each intermediate model output being fed into prompts of subsequent questions, leading to a step-by-step solution. This sequential strategy guides the model through a reasoning

process decided by the prompt design, reducing the risk of LLMs failing to reason correctly. For Text-to-SQL test cases, such a stepwise solution aligns well with first determining relevant tables, and then constructing a syntactically correct query with the given table names.

These strategies have shown improved reasoning results compared to traditional user prompting [50], [51], [52]. Decomposed prompting has particularly outperformed CoT when evaluated in studies comparing it to other strategies [51], [52]. Comparisons have, however, been conducted using arithmetic or symbolic tasks, meaning that a gap exists for leveraging these strategies in the context of guiding LLMs towards solving Text-to-SQL test cases by decomposing them into separate generalizable prompts.

4

Methods

Following the research strategy categorization proposed by Stol and Fitzgerald [53], this study aligns closest to a *laboratory experiment* environment since it investigates the relationship between various variables using predetermined test cases and quantifiable outcomes that can be compared. The experiment itself is designed in such a manner that produces the desired variables via distinct repetitions of executing the same test cases on different models. Since the same set of test cases is run across all models without randomly assigning them to treatment conditions, this experiment can be classified as a *quasi-experiment*, as per the distinction made by Wohlin *et al.* [54] between true randomized experiments and quasi-experiments within *Software Engineering* (SE). Additionally, this is a *technology-oriented* experiment and no subjects are therefore involved as Wohlin *et al.* note "in technology-oriented experiments, different technical treatments are applied to objects." [54, p. 73].

In order to further structure the experimentation within SE, this study is grounded by the process described by Wohlin *et al.* [54]. Overall, the book separates an experiment into four intertwined steps: 1. *Scoping* 2. *Planning* 3. *Operation* 4. *Analysis and Interpretation.*

In the preceding sections of this thesis, the context, overall objective, RQs and high-level boundaries were outlined. In the following sections, this thesis elaborates on considerations made for each of the four steps in an SE experiment. The *Scoping* step utilizes the context and goal of the experiment to formulate a clear goal definition, as well as ensures that the experiment's setup achieves the intended goals. The *Planning* phase provides a comprehensive set of decisions regarding the selection of test cases, models, datasets, and variables per RQ. Additionally, this step includes experimental design and statistical test choices for each of the RQs.

The next phase described by Wohlin *et al.* [54] is *Operation*, which comprises: 1. Setting up an environment that does not affect the results but rather observes them based on the applied treatments. 2. Defining data that the experiment intends to collect before validating it. 3. Discarding invalid data. When valid and sufficient data is available, *Analysis and Interpretation* can be conducted based on experimental design and statistical test choices made during the *Planning* phase.

4.1 Scoping

The *Goal-Question-Metric* (GQM) template is highlighted by Wohlin *et al.* [54] as the framework used when defining the goal of the experiment since the GQM model guarantees that the evaluation process is systematic and consistent with specified goals. Therefore, the scope of the experiment conducted in this study will be determined using the GQM model as followed:

> **Analyze** the behavior of open-source LLMs
> **for the purpose of** comparison and evaluation
> **with respect to** reliability, accuracy, and failure characteristics
> **from the point of view of** researchers
> **in the context of** executing Text-to-SQL benchmark test cases published by the BIRD team on open-source LLMs.

This study evaluates the performance and behavior of open-source LLMs from the same family on Text-to-SQL test cases derived from datasets published by the BIRD team. This enables a thorough examination of how these LLMs behave while working on Text-to-SQL problems. In particular, this research focuses on discovering the reliability of the models, their level of accuracy in producing SQL queries, and the failure characteristics when executing the model query. The results are intended to support researchers in understanding the strengths and limitations of LLMs in Text-to-SQL contexts. More information regarding the selection of LLMs and test cases is presented in Sect. 4.2.3.

The experimental objects for this experiment are therefore the selected LLMs and the Text-to-SQL test cases derived from the selected datasets. These cases represent problem instances on which the performance of the models is evaluated, while the models act as the executing entities that process test cases and generate SQL queries as output.

4.2 Planning

4.2.1 Design

The experimental design of this study follows the standard design types described by Wohlin *et al.* [54]. The experimental design is described separately for each RQ, since each RQ focuses on a different variable of interest. The number of runs, which refers to the repeated executions of the same input query for a given model, is fixed at 50. The repeated runs aim to account for the stochastic nature of LLM outputs. For all test cases, prompt design, meaning the structure of the input prompt provided to the LLMs, will be kept constant. Least-to-most prompting, which is explained further in Sect. 4.2.4, is the selected prompting strategy.

4.2.2 Variables & Experimental Design

The variables are initially introduced at a general level and summarized in Table 4.1. The role of each variable is then specified separately for each research question to clarify how they contribute to the corresponding analysis.

The variables used in the experiment are the following:

- **Query Complexity**: Represents the structural difficulty of SQL queries.
- **Model Size**: Measured as the number of active parameters in each selected model.
- **Accuracy**: Measures whether the model query produces the correct execution result compared with the golden query, following the execution accuracy introduced in Sect. 2.5.2.
- **Failure Mode**: Describes the type of failure in an incorrect SQL query.
- **Consistency**: A derived percentage measure of how often repeated executions of failed test-cases result in the same failure mode for a given query.

Table 4.1: Experiment Variables and Definitions

Variable	**Scale**	**Values / Levels**
Query Complexity	Ordinal	Simple, Moderate, Challenging
Model Size	Ordinal	A3B, A22B, A35B
Accuracy	Dichotomous nominal	Incorrect, Correct
Failure Mode	Nominal	Syntactic, Hallucination, Semantic
Consistency	Ratio	0–100%

RQ1 is addressed through a manual error analysis for each model by analyzing the failure modes from the incorrect portion of the collected data. The variable of interest for this research question is *Failure Mode*

RQ2 follows a one-factor design with more than two treatments. The factor is *Model Size*, with three levels: A3B, A22B, and A35B, and the dependent variable is *Accuracy*. The prompt design and number of runs are fixed. The research question evaluates how the size of the model affects the accuracy of the model on Text-to-SQL test cases.

- **Factor:** Model Size
- **Dependent variable:** Accuracy

RQ3 follows a one-factor design with more than two treatments as well. The factor is *Query Complexity*, with three levels: simple, moderate, and challenging. The dependent variable is *Consistency*. The prompt design and number of runs are

fixed. The research question investigates the reliability of model outputs and their consistency when the same input is executed repeatedly.

- **Factor:** Query Complexity
- **Dependent variable:** Consistency

4.2.3 Object Selection

The objects in this study, as mentioned in Sect. 4.1, are the test cases derived from the selected datasets, and the open-source LLMs from the same family.

The datasets selected for the experiment are ones published by the BIRD team. This includes both the LiveSQLBench-Base-Lite dataset [55] and the development or validation set of the original BIRD dataset [12]. The combination of chosen datasets yields the distributions of test cases shown in Table 4.2. However, the LiveSQLBench-Base-Lite dataset includes *create*, *read*, *update* and *delete* (CRUD) operation test cases, and the experiment relies on EX as an essential metric. Therefore, non-read test cases are omitted. The distributions of read test cases are presented in Table 4.3. The motivations for relying on these datasets for the evaluation are:

1. The BIRD team allows access to the golden SQL queries, which is a requirement for manual evaluation of model outputs.

2. All datasets published by the BIRD team have integrated query complexity categories that acknowledge difficulties across multiple dimension of the test cases.

3. Though there have been advancements in the performance of LLMs on the BIRD dataset, accuracy percentages are far from desirable real-world performance [12], [23], [40], making the dataset viable and the results relevant to research within the Text-to-SQL field.

Table 4.2: Distributions of all test cases by complexity

Dataset	**Count per Complexity**			
	Simple	**Moderate**	**Challenging**	**All**
BIRD dev	925	464	145	1534
Base lite	91	124	55	270
Both datasets	1016	588	200	**1804**

Since both of the chosen datasets provide query complexity categorization for each test case via the *difficulty* property, and the RQs are concerned with query complexity, the experiment is best conducted with equal distributions of each complexity category. However, within each category, test cases are sampled randomly, making *Stratified Random Sampling* [54], [56] the overall strategy utilized when selecting a

Table 4.3: Distributions of read test cases by complexity

Dataset	Count per Complexity			
	Simple	**Moderate**	**Challenging**	**All**
BIRD dev	925	464	145	1534
Base lite	42	89	49	180
Both datasets	967	553	194	**1714**

set of test cases for the experiment, where complexity categories are what the sampling technique refers to as *strata*. In this case, selecting per stratum aims to gather a set of test cases representative of the entire range of query complexity while the randomization reduces selection bias within each stratum. Datasets are combined and 100 test cases are randomly sampled from each stratum as shown in Table 4.4 for an overall count of 300 test cases.

Table 4.4: Distributions of the sampled set of read test cases by complexity

Count per Complexity			
Simple	**Moderate**	**Challenging**	**All**
100	100	100	**300**

Regarding the open-source LLMs that the test cases are run on, the experiment relies on Amazon *AWS Bedrock* [57], since it provides a unified API for accessing a range of high-performing models. All three of the selected models are from the same model family, *Qwen3*, and run on the *eu-north-1* region. The motivations for choosing Qwen3 models and AWS Bedrock are as follows:

1. AWS Bedrock provides a unified API for model access, which avoids local deployment differences and simplifies model access.

2. Selecting models from the same model family helps mitigate effects of architectural differences other than parameter size.

3. The experiment's design requires three models with increasing parameter sizes, and Qwen3 offers suitable models across multiple parameter sizes. The pricing of Qwen3 models is compatible with the experiment's budget.

When selecting models from the Qwen3 family, purposive sampling [56] was utilized since it allowed the selection to be focused on increasing parameter sizes. Therefore, the models shown in Table 4.5 were selected.

4.2.4 Prompting Strategy

In order to maximize the benefits of user prompting in the context of Text-to-SQL tasks, the experiment applies least-to-most prompting strategy [52] since it splits a

Table 4.5: The three selected Qwen3 models and their active parameter counts.

Model Name	Active Parameter Count
Qwen3 A3B	3 billion
Qwen3 A22B	22 billion
Qwen3 A35B	35 billion

test case into multiple smaller prompts, utilizing the output of each sub-prompt as input for the next. This has shown significant improvements across language and mathematical reasoning domains [52]. Additionally, the strategy improves performance by managing context windows more efficiently. By decomposing a task into shorter and more concise sub-prompts, it avoids performance degradation associated with increasing context windows, even when the input remains within a model's advertised context window limits [58], [59]. However, it is important to note that from the works that this study examined, least-to-most prompting has not been applied in Text-to-SQL contexts specifically.

Since the test cases follow a consistent structure, the initial task of analyzing the NLQ remains identical, making a manual decomposition of the process appropriate. This means that the researchers decide how to split the test case into two sequential prompting steps. In the first step, the model is tasked with establishing which tables are relevant for answering the NLQ. In the second step, the model is asked to formulate a correct SQL query which satisfies the NLQ and uses the tables outputted from step one.

4.2.5 Data Collection

The experimental process is structured around a fixed trial procedure that aims to provide a systematic evaluation of the selected models. In each trial, a test case passes through a series of stages, from input preparation to query execution, evaluation, and logging relevant information.

Trial Procedure:

- Input: NLQ + Database schema.
- Processing: Python script sends input to the selected model.
- Output: The SQL query generated by the model is collected by the script.
- Execution: Model SQL query and the golden query are executed against the corresponding database.
- Evaluation & Logging: After evaluating the result of the execution step, the relevant information is logged.

During each trial, the selected model is first prompted to select the necessary ta-

bles needed to answer the NLQ. In this prompt, the model is given all tables and corresponding column names from the database related to the NLQ, as shown in Figure 4.1a. All decoding parameters are set to default in this first step of the prompting strategy except *max_tokens* which is set to 512, *temperature* is set to 0.0, and a *timeout* of 300 seconds. In the second prompting step, the tables retrieved from the first step are provided alongside their full descriptions. The selected model is asked to produce an SQL query that answers the given NLQ, as shown in Figure 4.1b. Even in this step, all decoding parameters are set to default, except *max_tokens* which is now set to 1024, in order to have a larger limit size for the model output. *Temperature* is kept as 0.0, and the *timeout* remains at 300 seconds.

```
You are a schema selection
    assistant for Text-to-
   SQL.

Return the set of tables
   needed to answer the
   question.

{all_allowed_tables}

{NLQ}

{schema_text}
```

(a) Summarized template for table selection prompt.

```
You are a Text-to-SQL
   expert.

Using the selected tables,
    schema, context and
   the question, write one
    SQLite query that
   answers the question.

{selected_tables}

{schema_text}

{context}

{NLQ}

Return only the SQL query.
```

(b) Summarized template for SQL generation prompt.

Figure 4.1: Least-to-most prompting for Text-to-SQL. (a) Step 1 selects the relevant tables, and (b) Step 2 generates the final SQL query using the selected schema and context. More detailed templates for both steps are presented in Figure B.2 and Figure B.3.

The trial procedure is repeated under the same input conditions. An overview of the trial procedure is shown in Figure 4.2.

Evaluation:

The evaluation as well as logging of the result start when the model query is collected, meaning after step 3 in the trial procedure. The evaluation process is illustrated in Figure 4.3.

The model query is executed against the database. If the model query cannot be

Figure 4.2: Trial procedure for Text-to-SQL generation and evaluation.

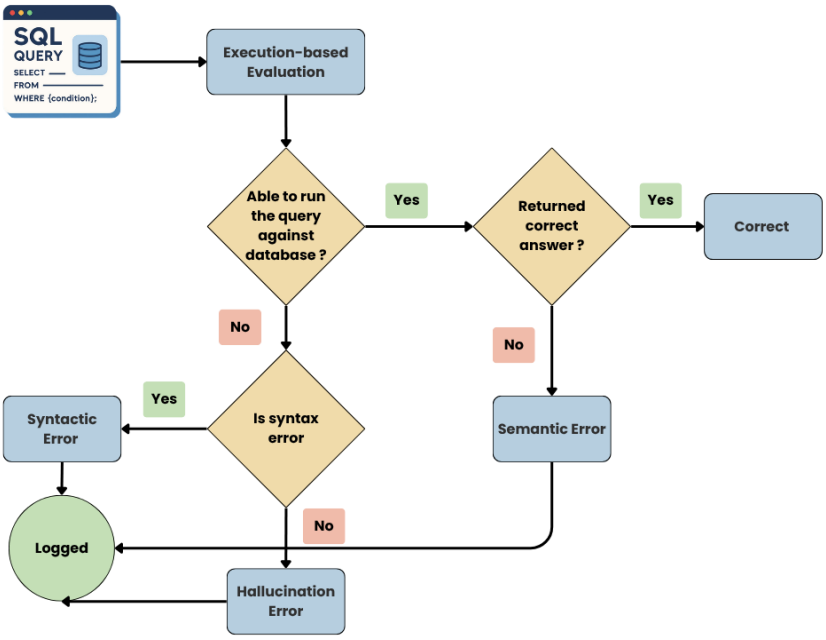

Figure 4.3: The Evaluation and logging process.

executed against the database, the error is further inspected. When the failure is caused by an invalid SQL syntax, the output is classified as a syntactic error. If syntax itself is not the main issue, the output is classified as a hallucination error. In this study, hallucination errors refer to cases where the model query includes nonexistent table or column names.

If the model query can be executed successfully, the returned result is compared with the result of the golden query. If the returned result does not match that of the golden query, the output is classified as a semantic error. This means that the model query is syntactically valid and executable, but does not return the expected answer. If both queries return the same answer, the output is classified as correct.

At the end of each trial, the result is logged. The logged information include whether the model query was correct or incorrect, the assigned failure mode if it is incorrect. A more detailed view of the collected data is illustrated in Figure B.1. This ensures that both successful and unsuccessful trials are consistently recorded for later analysis.

4.2.6 Hypothesis and Analysis Procedure

Once the experimental data is collected, various analyses are conducted in order to answer the research questions posed in Sect. 1.4. Initially, a descriptive-analysis approach will be utilized for all research questions. This allows for a simplified and summarized presentation of the collected data while simultaneously gaining an intuition for the kind of findings that are feasible for each research question. This is followed by statistical analyses for RQ2 and RQ3.

A manual error analysis is conducted for RQ1, inspired by Lei *et al.* [11], in order to draw more in-depth categorizations of the failure modes. The procedure consists of using the stratified random sampling strategy, as described in Sect. 4.2.3, to choose 50 samples from the incorrect portion of the collected data for each model. The error analysis is conducted primarily by one annotator. The purpose of the analysis is descriptive rather than to establish a validated taxonomy of Text-to-SQL errors. Each failed case is inspected using a fixed comparison procedure, the generated model SQL query, the golden SQL query, and the execution outputs are compared with respect to selected columns, tables, filters, aggregation, ordering, and output format.

For each test case, the golden query is taken as the definition of what the correct result should look like. The comparison begins by looking at the golden query and identifying selected columns, tables used, join conditions, filtering criteria, aggregation operations, sorting order, limits, and formatting. The model query is then compared to these characteristics in order to find the main structural or semantic difference. Comparison of the outputs between the generated model query and the golden query is done as well to determine whether the difference is due to missing rows, extra rows, duplicate rows, wrong column order, or extra columns.

Each error is already assigned to a failure mode during the trial procedure, semantic if the model query executes but returns an incorrect result, syntactic if the output is not a valid executable SQL query, and hallucination if the model query references a nonexistent table, column.

After this, failure reasons that describe the same underlying cause is grouped under the same subcategory name. For test cases with an ambiguous underlying cause, the classification is discussed before assigning a failure subcategory. The same subcategory name is reused whenever a new case reflects the same type of error, making the categorization consistent between test cases and suitable for grouped visualization.

In cases where the NLQ allows more than one reasonable interpretation, the golden query is still considered the reference answer that keeps the evaluation consistent.

In RQ2, the accuracy dependent variable is a dichotomous variable and is explored using two methods of classification for the 50 runs of each test case. The primary method, *majority-vote*, is classifying a test case as *correct* by majority vote if the category "correct" occurs more often than any other outcome across the 50 runs and *incorrect* otherwise, similar to the majority-vote criterion explained by Liu *et al.* [34]. The other supplementary method of classification, *at-least-one-correct*, is classifying a test case as *correct* if at least one of the 50 runs produces a "correct" result, and *incorrect* otherwise. The majority-vote criterion is the reliable measure that provides a realistic expectation of what the performance on a test case is. The at-least-one-correct criterion is a more lenient method that explores latent capabilities of the models.

Regarding the experimental design of RQ2, explained in Sect. 4.2.1, statistical methods of analysis suggested by Wohlin *et al.* [54] are ideally parametric ones. However, the dependent variable, accuracy, is a dichotomous nominal variable, and the measurements are repeated for each of the models, which violates the independence and continuity assumptions of parametric tests. A Non-parametric alternative, *Kruskal-Wallis* [60], is also unsuitable as it assumes independence of observations. Because of the dichotomous nominal type of response and repeated measures across model size treatments via multiple runs, a non-parametric *Cochran's Q* test [61], [62] is utilized for each of the complexity groups in order to investigate the difference between success outcomes. This is done once for each of the three complexity groups as the design of RQ2 investigates the difference between models for each complexity group separately. The null and alternative hypotheses for each of the Cochran's Q tests are:

- H_0: The success proportions are the same across the model-size treatments.

- H_1: At least one model-size treatment has a different success proportion.

Regarding RQ3, the dependent variable, consistency, is calculated once for all 50 attempts of the failed test cases. It is calculated as the count of the most commonly occurring failure mode, n_{common}, divided by the total count of runs, 50, and multiplied by 100 for percentage conversion, as seen in Eq. 4.1. The workflow that

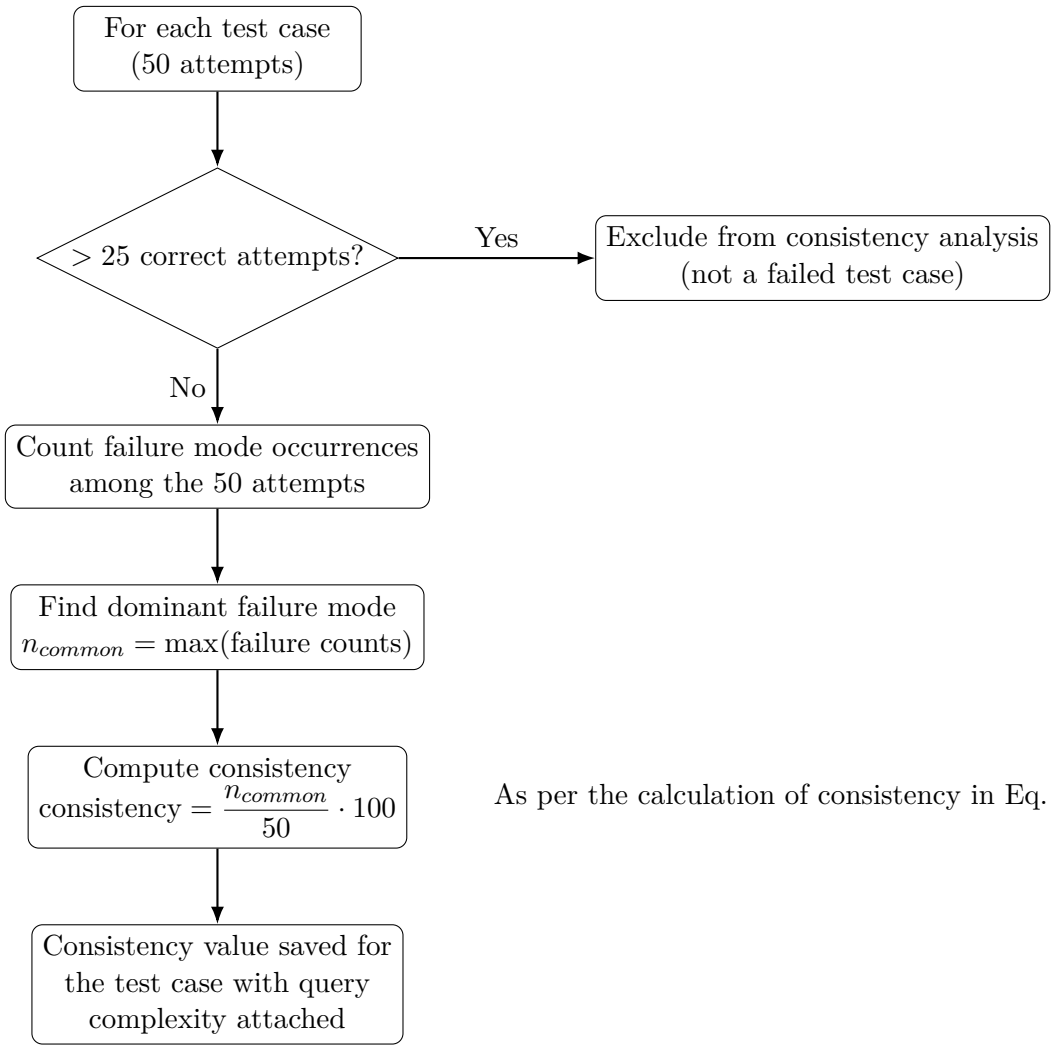

Figure 4.4: Workflow for selecting failed test cases and calculating failure consistency for RQ3. A test case is absolutely consistent if the computed consistency is equal to 100.

determines how test cases are selected for the measurement of consistency and what determines the most commonly occurring failure mode is shown in Figure 4.4. An *absolutely consistent* test case is one where n_{common} is exactly 50, meaning that the same failure mode is outputted for all 50 attempts of the test case.

$$\frac{n_{common}}{n_{total}} \cdot 100 \tag{4.1}$$

From the perspective of measurement scale, consistency does not violate the assumptions of parametric tests. Regarding normality however, parametric tests expect the dependent variable to approximately follow a normal distribution [54]. The consistency for all three models deviate significantly from a normal distribution, as indicated by the histogram plots for each of the models (see Figure C.2). Therefore, a non-parametric approach is more appropriate. For the experimental design

of RQ3, unlike RQ2, the Kruskal-Wallis test is a suitable non-parametric test that compares query complexity levels for each model separately. The null and alternative hypotheses for each model are:

- H_0: The distributions of consistency are the same across all query complexities.
- H_1: At least one query complexity level differs in its distribution of consistency.

For all statistical tests in RQ2 and RQ3, IBM's SPSS software [63] is utilized, and a significance level of $\alpha = 0.05$ is adopted as a convention widely accepted when conducting statistical analysis [54]. This means that results will be considered significant for a p-value under 0.05 ($p < 0.05$) and that there is a 5% chance that the null hypothesis is rejected falsely (Type-I-Error). For Cochran's Q analyses with significant results ($p < 0.05$), the *McNemar's* test is run as a post-hoc test with *Bonferroni correction* [64] on all pairs of model-size treatments, similar to the methodology adopted by Quintela-Pumares *et al.* [65]. This is done in order to explore which of the pairs are significantly different in success proportions more specifically. Applying the Bonferroni correction on the original significance level, $\alpha = 0.05$, gives a corrected threshold of $\alpha_1 \approx 0.0167$ as shown in Eq. 4.2. This controls the type-I-error rate at 5% between model-size treatment pairs.

$$\alpha_1 = \frac{\alpha}{n_{\text{unique pairs}}} = \frac{0.05}{3} \approx 0.0167 \tag{4.2}$$

4.2.7 Validity Evaluation

This section highlights possible validity threats that may affect the results. This is done during the planning stage in order to examine the results with greater caution regarding these aspects.

4.2.7.1 Internal Validity

One of the threats to internal validity in this experiment is prompt design. Prompting introduces potential effects to the outputs of LLMs due to the highly-sensitive nature of models to prompt design. Since both of the prompts in this experiment are designed and written by the researchers, and combined with the NLQs during execution, the specific phrasing, ordering or structure chosen during prompt design may unintentionally influence the generated SQL in ways unrelated to the treatment, thereby constituting a potential internal validity threat.

4.2.7.2 External Validity

Threats to external validity in this experiment relate to the selection of the LLMs and the test cases. The three LLMs selected for this experiment, Qwen3 A3B, Qwen3 A22B, and Qwen3 A35B belong to the same model family and are based on an MoE architecture. Thus, the results may be influenced by factors beyond model size such as training data, or characteristics specific to Qwen3 models. The

other threat that affects external validity is that of selecting test cases. All test cases were selected from datasets published by the BIRD team. Therefore, the experiment may be influenced by characteristics specific to the datasets, such as their language, question style, database schemas, SQL patterns, and annotation decisions. In addition, each test case has a predefined complexity level assigned by the BIRD team, as explained in Sect. 2.3.1. Since this definition of complexity may differ from how complexity is defined in other benchmarks or practical settings, this is acknowledged as a possible threat to external validity.

4.2.7.3 Conclusion Validity

Possible threats to conclusion validity are mainly related to statistical power, statistical test assumptions and validity of outcomes. Although non-parametric tests such as Cochran's Q and Kruskal-Wallis tests are used, they may still fail to detect real differences if the sample size is too small or if differences between models are subtle. The validity of the outcomes is also a possible threat, since the evaluation depends on the trial procedure used to compare generated model SQL query output with golden query output and to classify failure modes. Any incorrect classification or inconsistent comparison of outputs may affect the conclusions drawn from the results. Finally, test cases may vary in difficulty, database structure, ambiguity, and required SQL operations, even within the same complexity category. This random heterogeneity is therefore acknowledged as a possible threat when interpreting the experimental results.

4.2.7.4 Construct Validity

The trial procedure is a potential threat to construct validity because it compares model outputs against predetermined criteria for equivalence. These criteria may not exhaustively capture all cases in which two responses are effectively equal. For example, differently ordered lists or semantically identical phrases may be treated as non-equivalent. As a result, the measured performance may differ from the actual performance, which could affect conclusions about the relationship between the treatment and the outcome. The categorization of failure modes into syntactic, hallucination, or semantic failures may also affect construct validity since the boundaries between the failure modes are not always clear.

Another threat to construct validity is that accuracy does not account for other factors such as efficiency. This affects how to interpret the results of the study, as how accurate the generated SQL queries are does not indicate how efficient they are. Finally, another aspect of construct validity in this study is the use of *Artificial Intelligence* (AI) tools, such as ChatGPT. Considering the potential risks associated with AI tools, including hallucinations where the AI makes up false data, AI was applied with caution. In this study, ChatGPT is used to improve the formality of the language when needed. However, it was never used to create content. All information provided was verified using external papers and books. In addition, Claude is used to create boilerplate scripts. All code created by Claude is thoroughly examined and tested. The AI tools serve to automate repetitive tasks, accelerate initial

code development which allows the researchers to focus on research and analysis.

5

Results

Table 5.1 summarizes the overall outcome distribution using the majority result across 50 attempts for each test case. Across the 300 evaluated test cases per model, Qwen3 A22B achieved the highest proportion of correct outcomes with 50.3%, followed by Qwen3 A3B with 47.3% and Qwen3 A35B with 45.7%. Semantic failures were the dominant error type for all three models, ranging from 45.3% to 50.3%, while syntactic and hallucination failures occurred less frequently. This indicates that most incorrect outputs were executable queries that returned results different from the golden query, rather than failures caused by invalid SQL syntax or nonexistent schema elements.

Table 5.1: Overall classification of model outputs. Counts and percentages are reported over 300 evaluated test cases per model.

Model	Correct	Syntactic	Hallucination	Semantic
Qwen3 A3B	142 (47.3%)	5 (1.7%)	17 (5.7%)	136 (45.3%)
Qwen3 A22B	151 (50.3%)	0 (0.0%)	3 (1.0%)	146 (48.7%)
Qwen3 A35B	137 (45.7%)	6 (2.0%)	5 (1.7%)	152 (50.6%)

5.1 RQ1: Comparative Error Analysis and Failure Subcategories

The first research question investigates how the most commonly occurring subcategories of failure modes differ between LLMs. To answer this question, from the incorrect outputs of each model, 50 failed test cases were sampled and categorized, following the procedure described in Sect. 4.2.6. Therefore, 100% in Figures 5.1–5.2–5.3 corresponds to 50 analyzed failures for each model, rather than the full set of 300 evaluated test cases. The total number of failures before sampling was 158 for Qwen3 A3B, 149 for Qwen3 A22B, and 163 for Qwen3 A35B as shown in Table 5.2.

As shown in Figure 5.1, for the Qwen3 A3B model, semantic errors made up the largest category, accounting for 88% of the sampled errors (44/50), followed by hallucination errors at 10% (5/50) and syntactic errors at 2% (1/50). Among the subcategories, schema linking errors were the most common, representing 34% (17/50)

Table 5.2: Number of total failed cases for each model and the number of failed cases used for the manual error analysis.

Model	Total failures	Manually analyzed failures
Qwen3 A3B	158	50
Qwen3 A22B	149	50
Qwen3 A35B	163	50

of all errors. Other subcategories included aggregation errors 12% (6/50), wrong filter logic 10% (5/50), and wrong filter value 8% (4/50). Errors related to hallucination consisted of non-existent column errors 4% (2/50), non-existent table errors 4% (2/50), and wrong SQL dialect 2% (1/50). The syntactic category consisted of SQL syntax errors, which accounted for 2% (1/50) of the sampled errors.

As shown in Figure 5.2, for the Qwen3 A22B model, semantic errors accounted for the largest distributions of errors as well, representing 94% (47/50) of the analyzed sample, while hallucination errors accounted for 6% (3/50). No syntactic errors were observed for this model. The most common subcategories were schema linking errors and aggregation errors, each accounting for 22% (11/50) of all errors. These were followed by projection errors and duplicate errors, each at 10% (5/50). Other subcategories included calculation errors at 8% (4/50), wrong filter logic, wrong filter value, and golden query issues at 6% (3/50) each, as well as NULL handling errors at 4% (2/50). For hallucination errors, non-existent column errors at 4% (2/50), and non-existent table errors at 2% (1/50).

Lastly for the Qwen3 A35B model, as shown in Figure 5.3, semantic errors made up 96% (48/50) of the sample, followed by hallucination errors and syntactic errors, each accounting for 2% (1/50). The most common subcategory was schema linking error at 26% (13/50), followed by aggregation error at 16% (8/50), projection error at 14% (7/50), and calculation error at 12% (6/50). Additional semantic subcategories included wrong filter value, output format error, and golden query issue, each at 6% (3/50), while duplicate error accounted for 4% (2/50). Row selection error, NULL handling error, and wrong filter logic each accounted for 2% (1/50). The hallucination category consisted of non-existent column errors 2% (1/50), and the syntactic category consisted of SQL syntax errors 2% (1/50).

Figure 5.4 focuses on the three most common semantic failure subcategories for each model. Schema linking refers to errors that involve the use of incorrect table, column, or relationship selection. For example, as shown in Figure 5.5, the generated model query uses the **trans** table, while the golden query uses the **order** table. The model does not hallucinate a non-existent table, but it selects the wrong schema element for the question, which leads to an incorrect result.

Figure 5.1: Error Analysis of failures for Qwen3 A3B.

Figure 5.2: Error Analysis of failures for Qwen3 A22B.

Figure 5.3: Error Analysis of failures for Qwen3 A35B.

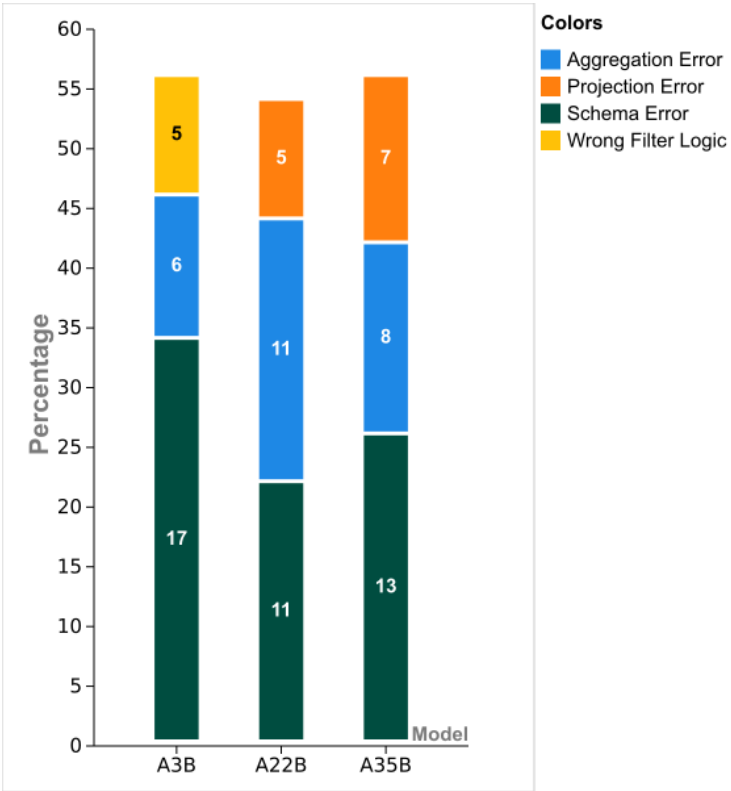

Figure 5.4: Top three most common semantic subcategories of each model. Each model is based on 50 analyzed failed cases, and the numbers inside the bars show the absolute number of cases in each subcategory. The y-axis shows the corresponding percentage of the 50 analyzed failures.

Schema linking error example

Question: How often does account number 3 request an account statement to be released? What was the aim of debiting 3539 in total?

Golden query
• Uses tables `account` + `order`

Result:
`POPLATEK MESICNE`
`POJISTNE`

Model query
• Uses tables `account` + `trans`

Result:
Empty

Main divergence: The model links the question to the wrong source table **trans**, so the generated query is executable but returns the wrong output.

Figure 5.5: Example of an schema linking error. In Appendix A, Figure A.1 illustrates a more detailed view.

Aggregation refers to incorrect use of aggregate functions or grouping. Figure 5.6 illustrates this type of error. The golden query considers the number of consumption entries each month that are greater than 1000, among customers who pay using euros. However, the model query aggregates using the condition "**HAVING SUM** (y.Consumption) > 1000" grouped by customers. Although the model query is syntactically valid, the incorrect aggregation level leads to wrong output.

Aggregation error example

Question: Among the customers who paid in euro, how many of them have a monthly consumption of over 1000?

Golden query
• Counts monthly records where `Consumption > 1000`

Result:
`1242`

Model query
• Groups results by `CustomerID`
• Counts customers through `HAVING SUM(Consumption) > 1000`

Result:
`2730`

Main divergence: The model changes the aggregation level from monthly consumption records to total consumption per customer, producing a different count.

Figure 5.6: Example of an aggregation error. In Appendix A, Figure A.2 illustrates a more detailed view.

Projection refers to incorrect selected output columns, and wrong filter logic refers to incorrect conditions in **WHERE** or **HAVING** clauses. The results show that schema linking and aggregation errors appear among the most common semantic

subcategories for all three models. Projection errors appear among the top three semantic subcategories for Qwen3 A22B and Qwen3 A35B, while wrong filter logic appears among the top three only for Qwen3 A3B. The definitions of the failure subcategories are provided in Appendix A, Table A.1.

Answer to RQ1. Overall, the results show that the evaluated models mainly differ at the failure subcategory level rather than at the broad failure mode level. Semantic failures dominate across all three models. While the most common semantic subcategories vary, schema linking errors appears as the largest or co-largest subcategory across all three models.

5.2 RQ2: Impact of Model Size on Text-to-SQL Accuracy

The second research questions asks how accuracies differ between LLMs of larger and smaller sizes for each query complexity. Because accuracy is measured per complexity and each row is based on 100 test cases, the values in Table 5.3 can be interpreted as both counts out of 100 and percentages. The accuracy was primarily evaluated using the majority-vote criterion, as explained in Sect. 4.2.6. The accuracies for the supplementary at-least-one-correct criterion are presented in parentheses. Statistical tests were run for both criteria.

The primary results show that performance is highest for *simple* queries, where all models perform nearly equally well: Qwen3 A3B achieves 52% correct outcomes, which corresponds to 52 correct test cases, while Qwen3 A22B and Qwen3 A35B achieve 51% each, meaning 51 correct test cases each. The mode is additionally *Correct* for all models in this setting.

For *moderate* queries, a minimal separation emerges. Qwen3 A22B remains above the halfway mark with 52% correct, whereas Qwen3 A3B and Qwen3 A35B both fall below 50%, with 46% and 48% respectively, which corresponds to a difference of 6 test cases between 52% and 46%. The mode values for Qwen3 A3B and Qwen3 A35B shift therefore to *Incorrect*.

Challenging queries cause all models to perform below 50%, with Qwen3 A22B at 46%, Qwen3 A3B at 44%, and Qwen3 A35B at 38%, meaning a difference 8 test cases between the Qwen3 A22B and Qwen3 A35B model. The mode values of both Qwen3 A3B and Qwen3 A35B remain *Incorrect*, and Qwen3 A22B shifts to *Incorrect* as well.

Overall, the frequency of the *Correct* mode decreased from three out of three models, to zero out of three when increasing query complexity from *simple* to *challenging*.

The overall differences between models are visualized more clearly in Figure 5.7. The figure highlights how all models drop in performance across complexities, especially when increasing the query complexity from *moderate* to *challenging* where the model

Table 5.3: Accuracy results across models for each query complexity level under the majority-vote criterion. Values for each row are out of 100 test cases per complexity level and therefore correspond numerically to counts. Values in parentheses show the accuracy results of the at-least-one-correct criterion.

Complexity	Model	Correct (%)	Incorrect (%)	Mode
Simple	Qwen3 A3B	**52** (53)	**48** (47)	Correct
	Qwen3 A22B	51 (**57**)	49 (**43**)	Correct
	Qwen3 A35B	51 (53)	49 (47)	Correct
Moderate	Qwen3 A3B	46 (49)	54 (51)	Incorrect
	Qwen3 A22B	**52** (**58**)	**48** (**42**)	Correct
	Qwen3 A35B	48 (51)	52 (49)	Incorrect
Challenging	Qwen3 A3B	44 (46)	56 (54)	Incorrect
	Qwen3 A22B	**46** (**56**)	**54** (**44**)	Incorrect
	Qwen3 A35B	38 (40)	62 (60)	Incorrect

with the highest active parameter count, Qwen3 A35B, succeeds in 10 less test cases while the best performing model Qwen3 A22B succeeds in 6 less test cases, meaning a difference of 4 test cases only.

When examining the per-complexity differences between models on the same sets of 100 test cases for each query complexity, there were no significant differences between the success proportions of the models on the *simple*, *moderate*, or *challenging* queries, as seen in the results of Cochran's Q test in Table 5.4. Therefore, when utilizing the majority-vote criterion for evaluation, the null hypothesis that states "The success proportions are the same across the model-size treatments" fails to be rejected for all query complexities.

When the threshold for classifying a test case as *Correct* is defined more leniently under the at-least-one-correct criterion, as explained in Sect. 4.2.6, success rates increase marginally across all complexity levels, as shown in the accuracies in parentheses in Table 5.3.

Table 5.4: Results of Cochran's Q tests on each query complexity under the majority-vote criterion.

Query complexity	Q	p-value
Simple	0.091	0.956
Moderate	2.240	0.326
Challenging	3.355	0.187

Statistically significant differences emerge on *challenging* queries between models when running the Cochran's Q test under the at-least-one-correct criterion (Q =

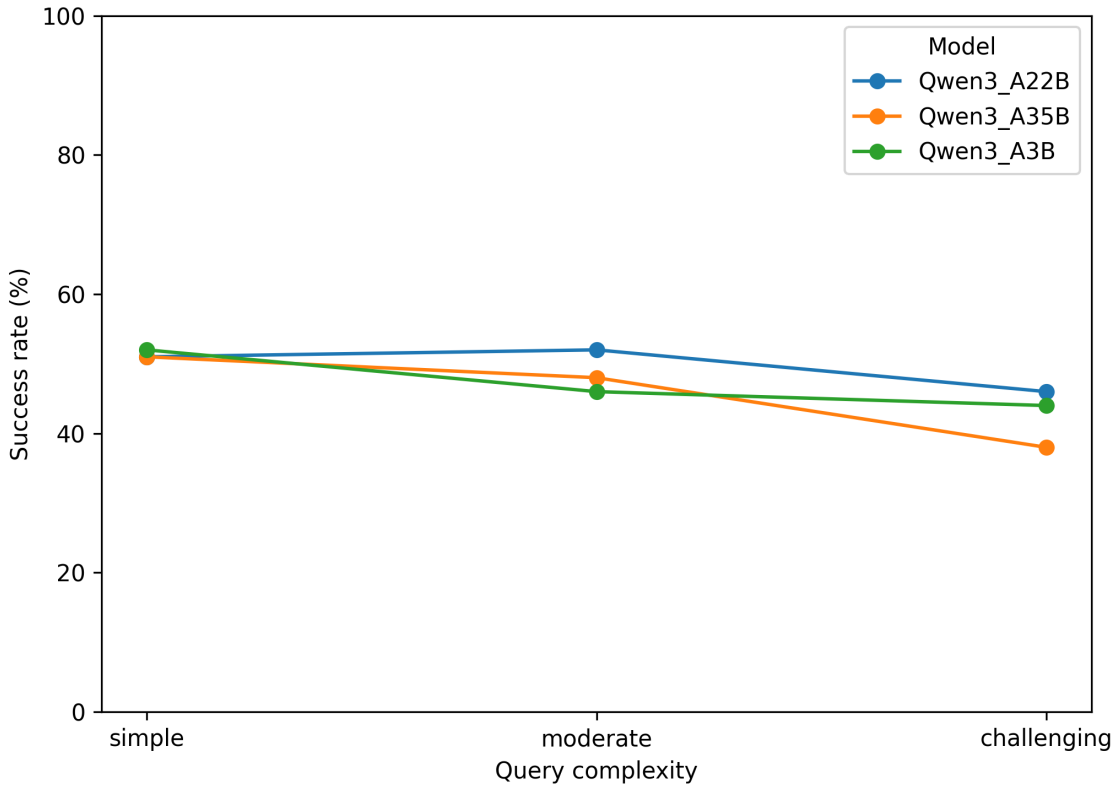

Figure 5.7: Line plot of models' success rates for each of the query complexity levels under the majority-vote criterion.

11.879, p = 0.003), as shown in Table 5.5. The Qwen3 A22B and Qwen3 A35B pair of models is however the only pair with significantly different success proportions (p = 0.002), as per the results of a post-hoc pairwise McNemar test on *challenging* queries, presented in Table 5.6. This p-value is below the Bonferroni corrected alpha threshold ($\alpha_1 < 0.0167$) and is therefore significant. The *simple* and *moderate* query complexities did not show statistical significance however (see Table 5.5). Therefore, when utilizing the at-least-one-correct criterion for evaluation, the null hypothesis that states "The success proportions are the same across the model-size treatments" is rejected for *challenging* queries, but cannot be rejected for *simple* and *moderate* queries.

Table 5.5: Results of Cochran's Q tests on each query complexity separately under the at-least-one-correct criterion.

Query complexity	Q	p-value
Simple	1.684	0.431
Moderate	4.467	0.107
Challenging	11.879	**0.003**

Results for the matched set of 100 *challenging* test cases on which both models were evaluated are presented in the contingency table in Table 5.7. The table shows that

Table 5.6: Results of pairwise McNemar tests across model-pairs on the *challenging* query complexity under the at-least-one-correct criterion.

	Qwen3 A3B & Qwen3 A22B	**Qwen3 A3B & Qwen3 A35B**	**Qwen3 A22B & Qwen3 A35B**
p-value	0.064	0.238	**0.002**

the pair produced the same accuracy on 76% of the test cases: 40% where both were *Incorrect* and 36% where both were *Correct*, meaning that the pair produced a *Correct* result in at least one attempt for 36 test cases, and *Incorrect* for 40 test cases. The remaining 24%, or 24 test cases, are discordant pairs: 20% of the test cases where only the Qwen3 A22B model was *Correct*, while test cases where only the Qwen3 A35B model produced a *Correct* SQL query are the remaining 4%, which correspond to a difference of 16 more test cases that the Qwen3 A22B model produced a *Correct* SQL query for.

Table 5.7: Contingency table for Qwen3 A22B and Qwen3 A35B on the set of 100 challenging test cases. Values represent percentages but correspond numerically to counts as well since the each query complexity has exactly 100 test cases.

	A35B Correct (%)	**A35B Incorrect (%)**
A22B Correct (%)	36	20
A22B Incorrect (%)	4	40

Answer to RQ2. Overall, model size increases do not show a consistent advantage across query complexities. For *simple* and *moderate* queries, Qwen3 A22B performs best, while Qwen A35B is the worst performing model on *challenging* queries, showing that the larger model is not necessarily more accurate. Statistically, no significant differences are found between models when accuracy is defined by the majority-correct criterion. Under the supplementary at-least-one-correct criterion, a significant difference appears only for *challenging* queries. More specifically, Qwen A35B is found to perform significantly worse than the smaller Qwen A22B. In practice, this suggests that for Text-to-SQL systems that sample multiple SQL queries, testing which model works best for the expected difficulty of Text-to-SQL tasks is more relevant than selecting the largest model. More on the implications of these results is presented in Chapter 6.

5.3 RQ3: Impact of Query Complexity on Failure Consistency

The third RQ investigates how the consistencies of test case failures are affected by the various query complexity levels. The results of exploring those consistencies for each model are presented in Table 5.8. Since RQ3 is only concerned with failed test cases, the number of cases analyzed for each model is reduced from the total count of test cases to 158 failed test cases for the Qwen3 A3B model (48 + 54 + 56), 151

for the Qwen3 A22B (49 + 48 + 54), and 163 for the Qwen3 A35B (49 + 52 + 62) when going through the workflow shown in Figure 4.4. Since consistency values are presented as percentages, the median being at 100% across all query complexities for all models is interpreted as reaching the maximum possible value of failure mode consistency at the 50^{th} percentile, meaning that at least half of the failed test cases are absolutely consistent. Absolutely consistent test cases are defined in Sect. 4.2.6 and Figure 4.4. The percentage of absolutely consistent cases in the next column presents a clearer picture since it represents the amount of failed test cases in which all 50 runs yielded the same failure mode. Finally, the minimum and maximum consistencies allow for understanding of values at the lower and upper bounds of consistency as they show the consistency of the least and most consistent failed test cases.

Table 5.8: Descriptive statistics of consistency for failed test cases, grouped by model and query complexity. Consistency values are multiplied by 100 and reported as percentages, as explained in Sect. 4.2.6.

Model	Complexity	N	Median	Absolute consistency (%)	Min, Max
Qwen3 A3B	Simple	48	100	94	(98, 100)
	Moderate	54	100	91	(96, 100)
	Challenging	56	100	89	(94, 100)
Qwen3 A22B	Simple	49	100	78	(56, 100)
	Moderate	48	100	81	(36, 100)
	Challenging	54	100	76	(34, 100)
Qwen3 A35B	Simple	49	100	92	(56, 100)
	Moderate	52	100	92	(52, 100)
	Challenging	62	100	92	(50, 100)

For the Qwen3 A3B model, the table shows that the amount of failed test cases increases with relatively similar amounts when increasing the query complexity. The percentage of absolutely consistent failures decreases with increased complexity as well, meaning that failure modes are less often the exact same on *challenging* test cases than on *moderate* and *simple* ones across all 50 runs. Examining the least consistent failed test cases for the same model shows that the minimum consistency does decrease when increasing complexity as well, but it does not drop below 94% for *challenging* test cases.

As mentioned in Sect. 5.2, in the case of the Qwen3 A22B model, there is no notable change in the amount of failed test cases between *moderate* and *simple* query complexities, though the *challenging* complexity displays an increase in the amount of failed cases. Inspecting percentages of cases with absolute failure mode consistency shows lower values across all query complexities for this model. Though no trend of decline between the absolute consistency of *simple* and *moderate* queries is observed,

the *challenging* queries have a smaller proportion of absolutely consistent test cases failures. This is reflected in the minimum and maximum consistencies of the model as the minimum goes as low as 34% for the least consistent *challenging* test case, and closely inconsistent on the *moderate* case with 36%, while the least consistent *simple* test case is still above 50% of consistency with 56%.

The biggest model in terms of model size, Qwen3 A35B has the largest number of failed test cases in total at 163, notably failing more often on *challenging* test cases than the other query complexities, though failures occur more often for *moderate* test cases than *simple* ones as well, with an increase of 49 to 52. Despite the difference in the amount of failed cases, the percentage of ones that the model achieved absolute consistency of failure mode outcomes on are the exact same at 92% across all three query complexities. The minimum consistency values decrease for each step of increase in query complexity, though all hover around and above the 50% mark, meaning that for the least consistent failed test cases, half or more of the attempts produce the same failure mode when evaluating model outputs.

No statistical significance was found between consistency value groups of the three query complexities for any of the models, as presented in Table 5.9. Therefore, the null hypothesis that states "The distributions of consistency are the same across all query complexities" cannot be rejected for any of the models.

Table 5.9: Results of Kruskal-Wallis tests across difficulty groups for each model.

Model	**H**	**p-value**
Qwen3 A3B	0.718	0.698
Qwen3 A22B	0.604	0.739
Qwen3 A35B	0.014	0.993

Answer to RQ3. The results show small variability patterns in consistency across models. Qwen3 A3B and Qwen3 A35B show somewhat lower minimum consistencies as complexity increases, while Qwen3 A22B has the lowest absolute-consistency rates overall and the lowest minimum consistencies. Statistical results for all three models do not suggest that increasing query complexity changes the consistency of failed test cases for any of the analyzed models as no significant difference was found between complexity levels. However, the high baseline of absolute consistency across the models is an important finding despite the statistical results. It illustrates that when these models fail, they systematically produce the exact same failure mode across all 50 attempts. More discussion regarding these findings is presented in Chapter 6.

6

Discussion

RQ1: How do the most commonly occurring subcategories of failure modes differ between LLMs?

Within the manual error analysis of the failed test cases, the findings indicate that while the models managed to generate executable SQL queries, they did not succeed in generating SQL queries that were semantically aligned with golden SQL queries. Thus, it can be stated that the challenge was not primarily SQL syntax or hallucination, but rather correct understanding of the intent of the NLQs and matching it with database schemas.

Schema linking appears to be a particularly important bottleneck. This connects to the definition of Text-to-SQL as a task that requires both understanding the intent of the NLQ and linking that to the correct database schema. The observed schema linking failures show that models can generate executable SQL queries while still selecting the wrong table, or schema path. This supports the motivation behind schema-aware Text-to-SQL approaches [41], where schema linking is treated as a separate and important part of the generation process. This suggests that improving the ability of the model to write valid SQL is not sufficient if the model still fails to select the correct tables, columns, or schema relationships needed to answer the NLQ.

The findings also help explain why multi-stage systems such as SQL-PaLM, DIN-SQL, and RESDSQL can be useful. These approaches use mechanisms such as candidate generation, execution filtering, self-correction, or explicit schema reasoning to improve reliability [39], [40], [41]. The RQ1 findings support the need for such mechanisms because many errors are not invalid SQL queries, but syntactically valid queries that retrieve the wrong result.

The comparison between models also connects to the model size discussion in prior work. Earlier studies [46], [47] suggest that larger models can improve reasoning and structured generation. The RQ1 findings indicate that model scaling alone may not remove the main Text-to-SQL bottlenecks. Even when some surface-level failures appear reduced, the same types of deeper semantic failure remain visible. This suggests that larger active parameter count may improve some aspects of generation, but schema linking and aggregation-related problems still require targeted

mechanisms.

Finally, these findings should be interpreted in relation to the evaluation method. Execution-based evaluation allows structurally different SQL queries to be accepted when they return the same result, but the result from executing golden query still defines the expected answer. This made the evaluation systematic and reproducible, but as a result, some semantic failures may reflect ambiguity in the NLQ or a valid alternative interpretation that differs from the golden query. In Appendix A, Figure A.3 shows such a case. The question asks whether molecule TR124 is carcinogenic. The golden SQL returns the raw database label "-", while the model SQL maps this label to the more explicit answer "No". Under the evaluation procedure, this is counted as incorrect because the result does not exactly match the golden output. However, from the perspective of the NLQ, "No" can be considered a valid answer.

> *RQ2: For each query complexity, how do the Text-to-SQL accuracies differ between LLMs of larger and smaller sizes?*

The very similar accuracies on simple queries across all three models under the primary majority-vote criterion, together with the statistical results not showing any significant difference between them, indicate that the increases in model size between the selected models may not have been large enough to produce clear variation in accuracy performance. The first increase, from Qwen3 A3B to Qwen3 A22B, corresponds to a sevenfold increase in active parameter count, while the increase from Qwen3 A22B to Qwen3 A35B is approximately one-and-a-half times. An alternative explanation is that performance on simple queries may already be close to a ceiling for these models, making the increase in active parameter count not sufficient to detect a change in accuracies, similar to the findings of Song *et al.* [18] on simpler test cases. For moderate and challenging queries, the observed accuracies vary more noticeably and Qwen3 A22B achieves the highest accuracy, but the statistical results still indicate that these differences are likely due to random chance.

Overall, across query complexities, a pattern of accuracy degradation is observed when increasing the difficulty of test cases, which is consistent with expectations [11], [12], [15], [17]. These observed differences are descriptive only, as no cross-complexity test was performed. However, such cross-complexity patterns remain interesting because they suggest that the complexity of test cases that practitioners intend to execute, may help identify an LLM choice that achieves a certain accuracy threshold while minimizing diminishing returns. This supports the benefit of utilizing multi-model approaches, where test cases are routed to appropriate models based on query complexity [34].

The supplementary at-least-one-correct criterion improved accuracies across all models, indicating that the models exhibit latent capability for producing correct SQL queries under repeated sampling. However, statistical tests show no significant differences for the simple and moderate query complexities. For challenging queries, in contrast, there is a significant difference in accuracies between Qwen3 A22B and

Qwen3 A35B, with Qwen3 A22B including 16 more correct test cases out of 100. This suggests that Qwen3 A22B has the strongest latent capability, or at least the highest observed ceiling on challenging Text-to-SQL test cases.

> *RQ3: For each of the analyzed LLMs, how are the consistencies of test-case failures affected as query complexity increases?*

From the analysis of failure mode consistency, additional context and understanding for the accuracy findings of RQ2 can be concluded. Although accuracy results, especially under the at-least-one-correct criterion highlight Qwen3 A22B as the best performing model, the consistency values indicate that its failed test cases are less stable than failures of the other models. More specifically, Qwen3 A22B has the lowest minimum consistency values and the lowest proportions of absolutely consistent failed test cases across the models, which suggests that its failures are more sensitive to repeated sampling, and therefore less reliable. This may help explain why the model benefits more from the supplementary accuracy criterion on challenging queries as it is more likely to occasionally produce a correct result. Qwen3 A3B in contrast, shows high failure consistency, with a minimum value of 94%, indicating that its failures are more systematic and less affected by sampling noise.

Generally, the median consistency of 100% across all models, as well as the high proportion of absolutely consistent failed test cases, suggests that many failures are systematic rather than random. This means that when a model fails to solve a test case initially, the model is unlikely to produce a correct result for the given decoding parameters, even after repeated sampling. Within each model, having no statistically significant differences across query complexity levels in the consistency results indicates that small differences in the proportion of absolutely consistent failed test cases are likely due to random chance. Therefore, the observed systematic failure patterns are not clearly caused or affected by query complexity within each model.

6.1 Threats to Validity

The manual error analysis used for RQ1 introduces a threat to construct validity. The fine-grained failure subgroups are based on manual inspection of 50 failed test cases for each model. Although this supports a more detailed analysis of failure patterns, some cases may contain multiple possible sources of error, and assigning one dominant subgroup can simplify the cause of the failure. The subgroup results should therefore be interpreted as descriptive patterns rather than complete explanations of all model failures.

Another threat to construct validity is the high proportion of semantic failures which may have been partially influenced by the evaluation procedure. Since the golden query is treated as the reference for the definition of a correct answer, generated queries are counted incorrect whenever their execution result differs from the golden result. This provides a consistent evaluation procedure, but it can also label valid

alternative interpretations of an NLQ as a semantic failure. Therefore, the semantic failure rate should be understood as failure relative to the golden query reference, rather than as an absolute measure of invalid SQL reasoning.

External validity is limited by the selected datasets and models. The experiment uses test cases published by the BIRD team and three Qwen3 models that can be accessed using the same interface. While this enhances internal consistency in the experimental design, the conclusions cannot be directly generalized to different datasets and families of models. In addition, the labeling of test cases as simple, moderate, or challenging is derived from the underlying dataset, implying that conclusions will depend largely on how the dataset developers classified the queries. However, under a similar selection of objects, the results should be valid.

Other threats to the validity in Sect 4.2.7 remain relevant but are not changed by the observed results. They are therefore treated as general limitations of the experimental design rather than result specific threats.

6.2 Implications & Lessons Learned

The results and interpretation regarding the first research question highlight the need for researchers to include safeguards such as intent clarification, schema verification, or user confirmation when designing Text-to-SQL systems. Additionally, they support that future research improvements may need to focus more on validation mechanisms that check whether the model query matches the NLQ intent, not only whether it executes successfully.

For researchers and practitioners, the interpretations of the second research question's results imply that when utilizing LLMs in Text-to-SQL contexts, small differences in active parameter count are likely not worth the extra costs in budget and resources for larger LLMs, in terms of model size, if that is a concern when selecting an LLM. This aligns well with recommendations of smaller models for small businesses and research purposes [18].

Furthermore, the findings are interesting since they indicate that, depending on test-case complexity, the best model for researchers looking to explore a retry loop or other Text-to-SQL settings that benefit from multiple sampled outputs may not be the largest available model in terms of model size.

The findings made by the results of the third research question suggest that accuracy alone is insufficient as a standalone measure of Text-to-SQL performance. For researchers, the results indicate that future evaluations should complement accuracy with reliability measures in order to gain additional understanding of failure reliability under repeated sampling. This is important for distinguishing between systematic and stochastic failures. For practitioners, the findings suggest that model selection should consider both accuracy and reliability. Even though LLMs do not deliver production-level accuracies at the current stage of research, access to reliability measures becomes increasingly essential for practitioners as Text-to-SQL systems

advance, especially in client-facing deployments where reliability is expected.

From conducting this study, the following lessons learned can be concluded:

- **The choice of the evaluation method and criterion should be context-dependent.** The accuracy in this study is defined as execution accuracy (EX), and both the majority-vote and at-least-one-correct criteria led to different statistical outcomes, specifically for challenging queries. This suggests that Text-to-SQL evaluation criteria should be selected based on the intended deployment architecture or the research purpose rather than adopting the setup from a certain study blindly. Additionally, studies reporting a single criterion or relying on a single attempt may not be exploring true model capabilities.

- **Following an established experimental framework structures decision-making before the execution.** Grounding the study's methods in Wohlin et al. [54] guidelines ensured that design choices, validity threats, and statistical test selections were addressed during the planning phase. This prevented reactive decisions and determined the analysis procedure independently of the results, which strengthens the credibility of the study.

- **Manual error analysis benefits from a predefined comparison procedure.** Defining the process of examining selected columns, tables, filters, aggregation, and output format before assigning a subcategory, helped keep the categorization consistent across the 150 analyzed test cases and three models.

- **Model selection required balancing research value and feasibility.** The selected models needed to be relevant to the research goal, available through the same platform, and possible to run within the available budget. This showed that experimental design is not only a theoretical choice, but also depends on practical constraints.

- **Logging more information than initially expected was valuable.** Storing only the final correctness outcome would not be sufficient for later analysis. In order to understand and verify the results, it was useful to log additional information such as the natural language question, selected tables, generated SQL query, golden SQL query, repetition number and model name. This made it possible to trace individual cases and inspect unexpected outcomes. Therefore, a key lesson was that experimental logging should be designed not only for the planned analysis but also for debugging, validation, and reproducibility.

6.3 Limitations & Delimitations

Limitations:

- It was not feasible to run entire Text-to-SQL datasets on the LLMs. Resources were limited to personal contributions on the AWS computing platform. Since

datasets include thousands of data points, API call costs would have become unmanageable if the entirety of both datasets were run on each model.

- Since available Text-to-SQL datasets are designed for academic testing and benchmarking, they follow naming conventions and high clarity level, which industry-level databases in the real-world might not, therefore worsening LLM results in real-world use cases.

- Prompt engineering may be a factor in the results. LLMs can be sensitive to small prompt variations, meaning that unnoticeable formatting or unintentional language patterns when prompting could have affected the results.

Delimitations:

- This study does not provide solutions that improve accuracy or reliability results. Rather, the study focuses on observing relationships between data and making reasonable conclusions regarding failures that can aid future solutions.

- NLQs, databases, and more generally, test cases, are limited to English only content due to the selected datasets. The goal of the study is not to test the translation or language knowledge of the LLMs, rendering this variable irrelevant.

- Model queries, as well as golden queries, are limited to the SQLite dialect due to the selected datasets. The cross-dialect results of LLMs on SQL queries are therefore not explored.

- Differences between the selected LLMs beyond active parameter count are disregarded. Since the variable of interest between the changing LLMs is model size, other architectural changes can cause confusions that make the scope of the study wider than the study's intended topic, and are thereby disregarded but acknowledged.

7

Conclusion

This study presents research that explores the failure modes, accuracy, and reliability of open-source LLMs across different query complexities. The study evaluated three Qwen3 models with different active parameter counts: Qwen3 A3B, Qwen3 A22B, and Qwen3 A35B, on Text-to-SQL test cases derived from the BIRD and LiveSQLBench-Base-Lite datasets. The selected test cases were divided into three query complexity levels: simple, moderate, and challenging. For each test case, the models were prompted to generate an SQL query based on the given NLQ and the database schema. The generated model query was then executed and compared with the corresponding golden query. Each such trial was repeated multiple times to account for the stochastic nature of the LLM outputs and to analyze the consistency of failures.

In order to investigate failure modes in depth, a manual error analysis was performed. The results show that the evaluated models were generally able to produce executable SQL queries, while the main challenge was producing SQL queries that are semantically aligned with the intended meaning of the NLQ and database schema. Semantic failures were the dominant failure mode in all three models. Schema linking and aggregation-related errors were among the most common sources of semantic failures. The findings highlight some key areas where future Text-to-SQL models can be improved.

The accuracy analysis also provides useful insights for model selection. Under the primary majority-vote criterion, the differences between the selected models were not statistically significant. This suggests that increasing the active parameter count alone may not guarantee better Text-to-SQL performance. This indicates that larger models may not always justify their additional computational cost when the accuracy gains are small or uncertain. Under the supplementary at-least-one-correct criterion, the models achieved higher accuracy, and a significant difference was observed for challenging queries between Qwen3 A22B and Qwen3 A35B. This shows that repeated sampling can reveal latent model capability, especially for more difficult Text-to-SQL test cases.

The consistency analysis shows that the outcomes of many failed test cases were highly consistent across repeated attempts, indicating that failures were often systematic rather than purely random. The absence of statistically significant differ-

ences in consistency across query complexity levels suggests that failure reliability cannot be explained only by the assigned complexity level and that other factors may play an important role.

Overall, the findings are useful for researchers aiming to improve Text-to-SQL systems and for practitioners who need to select models under resource constraints. Future work could extend this study by analyzing a broader range of LLM families, datasets and prompting strategies. In addition, evaluation methods that better account for semantically equivalent SQL output and valid alternative interpretations of the NLQ could be developed. Future studies may also test whether stronger schema linking and improved aggregation handling reduce the dominant semantic failure patterns observed in this study. Such work could contribute to a more reliable understanding of when Text-to-SQL systems can be used in practice, and what kind of improvements are needed before they can be trusted in high-stakes or complex database environments.

7.1 Future Work

During this study, the following possible areas of future work were identified:

Standardized Complexity Metric. As query complexity is an essential part of the methodology in this study, the collected test cases were limited to ones published by the BIRD team and based on the complexity levels assigned by their annotators. In order to enable more consistent evaluations of query complexity, a standardized tool that measures query complexity entire test cases, including the NLQ, database and expected query on a reasonable categorical or numerical metric would be beneficial. This would additionally allow researchers to measure complexity on existing datasets, making the available pool of datasets with complexity much more diverse.

Model Scaling and Architecture Choices. From the findings of this study, one unexpected results was the insignificant change in reliable performance from smaller to larger models which was possibly due to the small increases in active parameter counts. It would therefore be an interesting area of study to replicate the study with a set of models from the same family, though with larger and clearer model size differences. Another alternative is to select models with relatively similar increases in parameter count, though with dense models instead of MoE models, in order to observe how reliability and performance are affected between the architectures, in the context of Text-to-SQL.

Explaining Failures. The results of this study could not statistically show that changes in dichotomous accuracy results are caused by changes in model size. Therefore, statistically exploring other possible factors that contribute to accuracy, such as prompting strategies and prompt wording would be a helpful future work that would aid in improving Text-to-SQL models.

Bibliography

[1] A. Singh, A. Shetty, A. Ehtesham, S. Kumar, and T. T. Khoei, "A survey of large language model-based generative ai for text-to-sql: Benchmarks, applications, use cases, and challenges," in *2025 IEEE 15th Annual Computing and Communication Workshop and Conference (CCWC)*, 2025, pp. 00 015–00 021. DOI: 10.1109/CCWC62904.2025.10903689.

[2] I. K. Chadha, A. Gupta, S. Sarkar, M. Tomer, and T. Rathee, "Performance evaluation of open-source llms for text-to-sql conversion in healthcare data," in *2025 International Conference on Pervasive Computational Technologies (ICPCT)*, 2025, pp. 606–609. DOI: 10.1109/ICPCT64145.2025.10941238.

[3] A. Tripathi et al., "End-to-end text-to-sql with dataset selection: Leveraging llms for adaptive query generation," in *2025 International Joint Conference on Neural Networks (IJCNN)*, 2025, pp. 1–9. DOI: 10.1109/IJCNN64981.2025.11227747.

[4] M. Visperas, A. J. Adoptante, C. J. Borjal, M. T. Abia, J. K. Catapang, and E. Peramo, "On modern text-to-sql semantic parsing methodologies for natural language interface to databases: A comparative study," in *2023 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, 2023, pp. 390–396. DOI: 10.1109/ICAIIC57133.2023.10067134.

[5] S.-S. Shin, "Effect of semantic distance on learning structured query language: An empirical study," *Frontiers in Psychology*, vol. Volume 13 - 2022, 2022, ISSN: 1664-1078. DOI: 10.3389/fpsyg.2022.996363. [Online]. Available: https://www.frontiersin.org/journals/psychology/articles/10.3389/fpsyg.2022.996363.

[6] Z. Shaikhiyeva, M. Mansurova, and G. Amirkhanova, "Text to sql transformation using llm: A comparative research of t5, seq2seq, and sqlnet models," in *2024 9th International Conference on Computer Science and Engineering (UBMK)*, 2024, pp. 938–943. DOI: 10.1109/UBMK63289.2024.10773579.

[7] C. Sinsky et al., "Allocation of physician time in ambulatory practice: A time and motion study in 4 specialties," *Annals of Internal Medicine*, vol. 165, no. 11, pp. 753–760, 2016. DOI: 10.7326/M16-0961.

[8] S. Makhni et al., "Meeting the challenges of electronic health record (ehr) optimization," *npj Digital Medicine*, vol. 9, no. 1, p. 8, 2025. DOI: 10.1038/s41746-025-02178-w.

[9] H. Singh, T. D. Giardina, A. N. D. Meyer, S. N. Forjuoh, R. M. Reis, and E. J. Thomas, "Types and origins of diagnostic errors in primary care settings," *JAMA Internal Medicine*, vol. 173, no. 6, pp. 418–425, 2013. DOI: 10.1001/jamainternmed.2013.2777.

[10] A. K. Sharma, S. C. Kanumuri, P. A R, and S. S. Sharath, "Advancing natural language to sql: A comparative study of open source llms on benchmark datasets," in *2025 IEEE Symposium on Computational Intelligence in Natural Language Processing and Social Media (CI-NLPSoMe Companion)*, 2025, pp. 1–5. DOI: 10.1109/CI-NLPSoMeCompanion65206.2025.10977851.

[11] F. Lei et al., *Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows*, 2025. DOI: 10.48550/arXiv.2411.07763.

[12] J. Li et al., *Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls*, 2023. DOI: 10.48550/arXiv.2305.03111.

[13] M. S. Baig, T. Sher, A. Rehman, and S. Sheikh, "A systematic literature review of text-to-sql: Performance, challenges, and limitations," *ICCK Transactions on Advanced Computing and Systems*, vol. 2, no. 1, pp. 1–24, 2025, ISSN: 3068-7969. DOI: 10.62762/TACS.2025.497935.

[14] G. Jiang and C. Ma, "Exploring large language models for text-to-sql error correction with lora fine-tuning," in *2025 8th International Conference on Advanced Algorithms and Control Engineering (ICAACE)*, 2025, pp. 2670–2674. DOI: 10.1109/ICAACE65325.2025.11018984.

[15] Z. Peng, T. Zhu, C. Liu, Y. Chen, Z. Zhang, and Z. Ye, "Sage: Synthetic-augmented generation with error feedback for text-to-sql," in *2025 IEEE 3rd International Conference on Sensors, Electronics and Computer Engineering (ICSECE)*, 2025, pp. 656–661. DOI: 10.1109/ICSECE65727.2025.11256835.

[16] M. Ganti, L. Orr, and S. Wu, "Evaluating text-to-sql model failures on real-world data," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, 2024, pp. 1–1. DOI: 10.1109/ICDE60146.2024.00456.

[17] Q. Li, T. You, J. Chen, Y. Zhang, and C. Du, "Li-emrsql: Linking information enhanced text2sql parsing on complex electronic medical records," *IEEE Transactions on Reliability*, vol. 73, no. 2, pp. 1280–1290, 2024. DOI: 10.1109/TR.2023.3336330.

[18] Y. Song et al., "Enhancing text-to-sql translation for financial system design," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '24, Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 252–262, ISBN: 9798400705014. DOI: 10.1145/3639477.3639732.

[19] D. Khurana, A. Koli, K. Khatter, and S. Singh, "Natural language processing: State of the art, current trends and challenges," *Multimedia Tools and Applications*, vol. 82, no. 3, pp. 3713–3744, Jul. 2022, ISSN: 1573-7721. DOI: 10.1007/s11042-022-13428-4.

[20] A. Quamar, V. Efthymiou, and C. Lei, "Natural language interfaces to data," *Foundations and Trends in Databases*, vol. 11, no. 4, pp. 319–414, May 2022, ISSN: 1931-7891. DOI: 10.1561/1900000078.

[21] Z. Hong et al., "Next-generation database interfaces: A survey of llm-based text-to-sql," *IEEE Transactions on Knowledge and Data Engineering*, vol. 37, no. 12, pp. 7328–7345, 2025. DOI: 10.1109/TKDE.2025.3609486.

[22] J. Sarker, M. Billah, and M. A. Mamun, "Textual question answering for semantic parsing in natural language processing," in *2019 1st International Conference on Advances in Science, Engineering and Robotics Technology (ICASERT)*, 2019, pp. 1–5. DOI: 10.1109/ICASERT.2019.8934734.

[23] Y. Huang et al., *Exploring the landscape of text-to-sql with large language models: Progresses, challenges and opportunities*, 2025. DOI: 10.48550/arXiv.2505.23838.

[24] T. Mahmud, K. M. Azharul Hasan, M. Ahmed, and T. H. C. Chak, "A rule based approach for nlp based query processing," in *2015 2nd International Conference on Electrical Information and Communication Technologies (EICT)*, 2015, pp. 78–82. DOI: 10.1109/EICT.2015.7391926.

[25] A. Mohammadjafari, A. S. Maida, and R. Gottumukkala, *From natural language to sql: Review of llm-based text-to-sql systems*, 2025. DOI: 10.48550/arXiv.2410.01066.

[26] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, *The curious case of neural text degeneration*, 2020. DOI: 10.48550/arXiv.1904.09751.

[27] M. Peeperkorn, T. Kouwenhoven, D. Brown, and A. Jordanous, *Is temperature the creativity parameter of large language models?* 2024. DOI: 10.48550/arXiv.2405.00492.

[28] N. Shazeer et al., "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *arXiv preprint arXiv:1701.06538*, 2017. DOI: https://doi.org/10.48550/arXiv.1701.06538.

[29] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022.

[30] X. Zhu, Q. Li, L. Cui, and Y. Liu, *Large language model enhanced text-to-sql generation: A survey*, 2024. arXiv: 2410.06011 [cs.DB]. [Online]. Available: https://arxiv.org/abs/2410.06011.

[31] M. Pourreza, R. Sun, H. Li, L. Miculicich, T. Pfister, and S. O. Arik, *Sql-gen: Bridging the dialect gap for text-to-sql via synthetic data and model merging*, 2024. arXiv: 2408.12733 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2408.12733.

[32] B. Jamil, J. Ferzund, A. Batool, and S. Ghafoor, "Empirical validation of relational database metrics for effort estimation," in *INC2010: 6th International Conference on Networked Computing*, 2010, pp. 1–5. [Online]. Available: https://ieeexplore.ieee.org/document/5484854.

[33] M. A. P. Subali and S. Rochimah, "A new model for measuring the complexity of sql commands," in *2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE)*, 2018, pp. 1–5. DOI: 10.1109/ICITEED.2018.8534782.

[34] X. Liu et al., "A survey of text-to-sql in the era of llms: Where are we, and where are we going?" *IEEE Transactions on Knowledge and Data Engineering*, vol. 37, no. 10, pp. 5735–5754, 2025. DOI: 10.1109/TKDE.2025.3592032.

[35] C. Finegan-Dollak et al., "Improving text-to-SQL evaluation methodology," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, I. Gurevych and Y. Miyao, Eds., Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 351–360. DOI: 10.18653/v1/P18-1033.

[36] V. Zhong, C. Xiong, and R. Socher, *Seq2sql: Generating structured queries from natural language using reinforcement learning*, 2017. DOI: 10.48550/arXiv.1709.00103.

[37] T. Yu et al., "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds., Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 3911–3921. DOI: 10.18653/v1/D18-1425.

[38] V. Zhong, M. Lewis, S. I. Wang, and L. Zettlemoyer, "Grounded adaptation for zero-shot executable semantic parsing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds., Online: Association for Computational Linguistics, Nov. 2020, pp. 6869–6882. DOI: 10.18653/v1/2020.emnlp-main.558.

[39] M. Pourreza and D. Rafiei, *Din-sql: Decomposed in-context learning of text-to-sql with self-correction*, 2023. DOI: 10.48550/arXiv.2304.11015.

[40] R. Sun et al., *Sql-palm: Improved large language model adaptation for text-to-sql (extended)*, 2024. DOI: 10.48550/arXiv.2306.00739.

[41] H. Li, J. Zhang, C. Li, and H. Chen, *Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql*, 2023. DOI: 10.48550/arXiv.2302.05965.

[42] Z. Zhang, Y. Chen, C. Guo, J. Song, and G. He, "Enhancing text-to-sql generation with language sequential consistency," *Neurocomputing*, vol. 658, p. 131 721, 2025. DOI: 10.1016/j.neucom.2025.131721.

[43] J. Jiang, K. Zhou, Z. Dong, K. Ye, W. X. Zhao, and J.-R. Wen, *Structgpt: A general framework for large language model to reason over structured data*, 2023. DOI: 10.48550/arXiv.2305.09645.

[44] M. O. Derin, E. Uçar, B. Yergesh, Y. Shimada, Y. Hong, and X.-Y. Lin, "Evaluating the impact of model size on multilingual json structuring for knowledge graphs with recent llms," in *2024 IEEE 3rd International Conference on Problems of Informatics, Electronics and Radio Engineering (PIERE)*, 2024, pp. 1890–1895. DOI: 10.1109/PIERE62470.2024.10805070.

[45] X. Luo et al., *Scaling laws for code: A more data-hungry regime*, 2025. DOI: 10.48550/arXiv.2510.08702.

[46] J. Wei et al., *Emergent abilities of large language models*, 2022. DOI: 10.48550/arXiv.2206.07682.

[47] J. Kaplan et al., *Scaling laws for neural language models*, 2020. DOI: 10.48550/arXiv.2001.08361.

[48] L. Yan, Q. Wan, C. Liu, S. Duan, P. Han, and Y. Xu, "Sps-sql: Enhancing text-to-sql generation on small-scale llms with pre-synthesized queries," *Pattern*

Recognition Letters, vol. 196, pp. 45–51, 2025, ISSN: 0167-8655. DOI: 10.1016/j.patrec.2025.04.016.

[49] L. Sheng and X. S. Shuai, "SLM-SQL: An exploration of small language models for text-to-SQL," in *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, K. Inui et al., Eds., Mumbai, India: The Asian Federation of Natural Language Processing and The Association for Computational Linguistics, Dec. 2025, pp. 1497–1512, ISBN: 979-8-89176-303-6. DOI: 10.18653/v1/2025.findings-ijcnlp.92.

[50] J. Wei et al., *Chain-of-thought prompting elicits reasoning in large language models*, 2023. DOI: 10.48550/arXiv.2201.11903.

[51] T. Khot et al., *Decomposed prompting: A modular approach for solving complex tasks*, 2023. DOI: 10.48550/arXiv.2210.02406.

[52] D. Zhou et al., *Least-to-most prompting enables complex reasoning in large language models*, 2023. DOI: 10.48550/arXiv.2205.10625.

[53] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 3, pp. 1–51, Sep. 2018, ISSN: 1049-331X. DOI: 10.1145/3241743.

[54] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, 2nd ed. Berlin, Heidelberg: Springer, 2024, p. 274, ISBN: 978-3-662-69305-6. DOI: 10.1007/978-3-662-69306-3.

[55] BIRD Team, *Livesqlbench-base-lite-sqlite: A dynamic benchmark for text-to-sql evaluation*, Hugging Face dataset, 2025. [Online]. Available: https://huggingface.co/datasets/birdsql/livesqlbench-base-lite-sqlite.

[56] S. Baltes and P. Ralph, "Sampling in software engineering research: A critical review and guidelines," *Empirical Software Engineering*, vol. 27, no. 4, p. 94, 2022, ISSN: 1573-7616. DOI: 10.1007/s10664-021-10072-8.

[57] Amazon Web Services, *Amazon bedrock documentation*, https://aws.amazon.com/documentation-overview/bedrock/, Accessed: 2026-04-29, 2026.

[58] C.-P. Hsieh et al., "RULER: What's the real context size of your long-context language models?" In *First Conference on Language Modeling*, 2024. [Online]. Available: https://openreview.net/forum?id=kIoBbc76Sy.

[59] Y. Du et al., "Context length alone hurts LLM performance despite perfect retrieval," in *Findings of the Association for Computational Linguistics: EMNLP 2025*, C. Christodoulopoulos, T. Chakraborty, C. Rose, and V. Peng, Eds., Suzhou, China: Association for Computational Linguistics, Nov. 2025, pp. 23 281–23 298, ISBN: 979-8-89176-335-7. DOI: 10.18653/v1/2025.findings-emnlp.1264.

[60] E. Ostertagová, O. Ostertag, and J. Kováč, "Methodology and application of the kruskal-wallis test," *Applied Mechanics and Materials*, vol. 611, pp. 115–120, Oct. 2014. DOI: 10.4028/www.scientific.net/AMM.611.115.

[61] P. Mishra, C. M. Pandey, U. Singh, A. Keshri, and M. Sabaretnam, "Selection of appropriate statistical methods for data analysis," *Annals of Cardiac Anaesthesia*, vol. 22, no. 3, pp. 297–301, 2019. DOI: 10.4103/aca.ACA_248_18.

[62] NCSS, LLC., *Cochran's q test*, Accessed 2026-05-18, NCSS Statistical Software, ch. 521. [Online]. Available: https://www.ncss.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Cochrans_Q_Test.pdf.

[63] IBM Corp., *Ibm spss statistics*, version 28.0.1.1, 2021. [Online]. Available: https://www.ibm.com/products/spss-statistics.

[64] J. M. Bland and D. G. Altman, "Multiple significance tests: The bonferroni method," *BMJ*, vol. 310, no. 6973, p. 170, 1995, ISSN: 0959-8138. DOI: 10.1136/bmj.310.6973.170.

[65] M. Quintela-Pumares, D. Fernández-Lanvin, and A.-M. Fernandez-Alvarez, "Heuristic-based incremental local domain model generation," *Information and Software Technology*, vol. 186, p. 107 817, 2025, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2025.107817.

A

Error Analysis

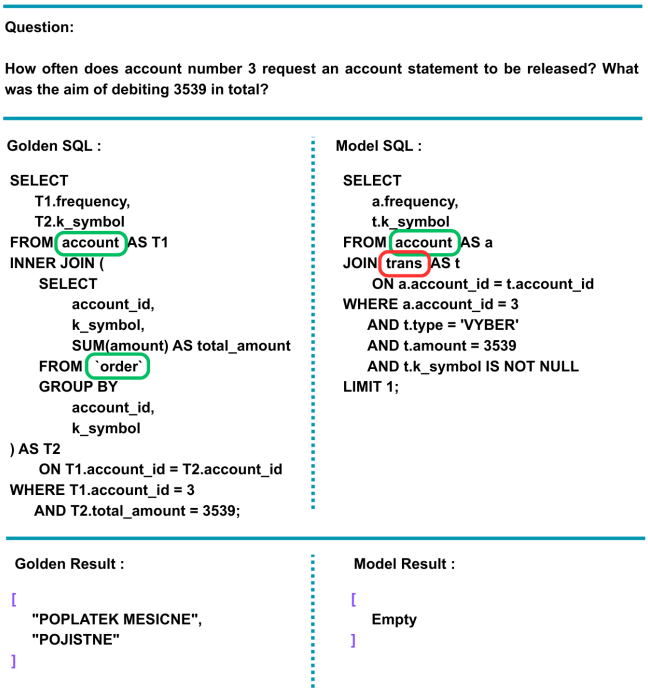

Figure A.1: Example of a schema linking error. The model query uses a different source table than the golden query, resulting in an incorrect output.

Table A.1: Failure subcategory definitions used in the manual error analysis.

Failure subcategories	Definition	Example
Schema Linking Error	The model query uses an incorrect but existing table, column, or join path.	Fig A.1
Aggregation Error	The model query uses incorrect counting, grouping, aggregation scope, or aggregation level.	Fig A.2
Wrong Filter Logic	The model query uses an incorrect structure, such as an extra filter, missing filter, or wrong logical operator.	Fig A.4

Failure subcategories	**Definition**	**Example**
Wrong Filter Value	The model query uses an incorrect literal value in a filter, even when the general condition is relevant.	Fig A.5
Calculation Error	The model query uses an incorrect formula, arithmetic operation, percentage conversion, or derived metric calculation.	Fig A.6
Projection Error	The model query selects incorrect, missing, or unnecessary output columns compared with the expected answer.	Fig A.8
Output Format Error	The model query returns mostly relevant content but in the wrong representation, encoding, or output shape.	Fig A.3
Duplicate Error	The model query incorrectly keeps or removes duplicate rows, often due to incorrect use or omission of `DISTINCT`.	Fig A.7
Row Selection Error	The model query selects the wrong specific row or subset of rows, often due to incorrect ranking, ordering, or `LIMIT`.	Fig A.9
NULL Handling Error	The model query handles `NULL` values incorrectly, either by including or excluding them in a way that changes the result.	Fig A.10
Golden Query Issue	The observed failure appears to result from a problem, ambiguity, or validity issue in the golden query rather than from the generated model query.	
Nonexistent Column	The model referenced a nonexistent column name.	
Nonexistent Table	The model referenced a nonexistent table name.	
SQL Syntax	The model query is not an executable SQL query due to syntax errors, or invalid SQL structure.	

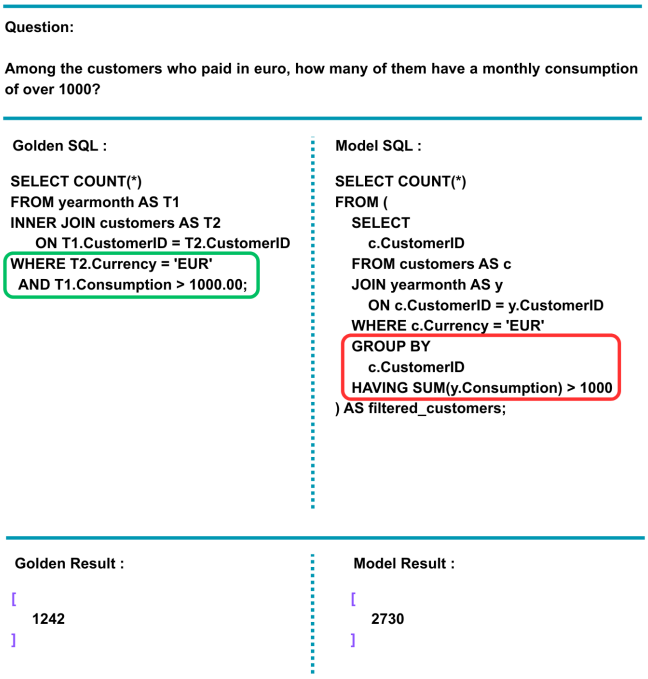

Figure A.2: Example of an aggregation error. The model query aggregates consumption at the customer level, while the golden query filters monthly consumption records directly.

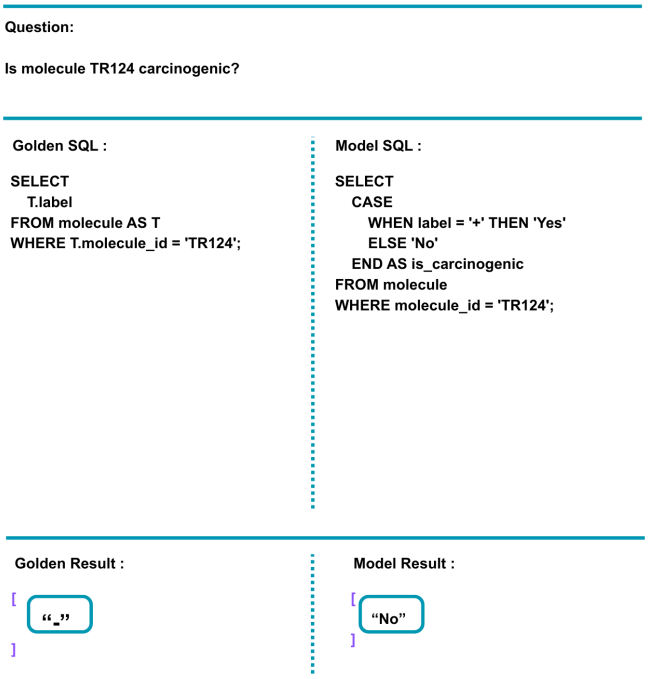

Figure A.3: Example of an output format error. The model's result is presented differently from the golden result, but it may still be considered a valid interpretation of the NLQ.

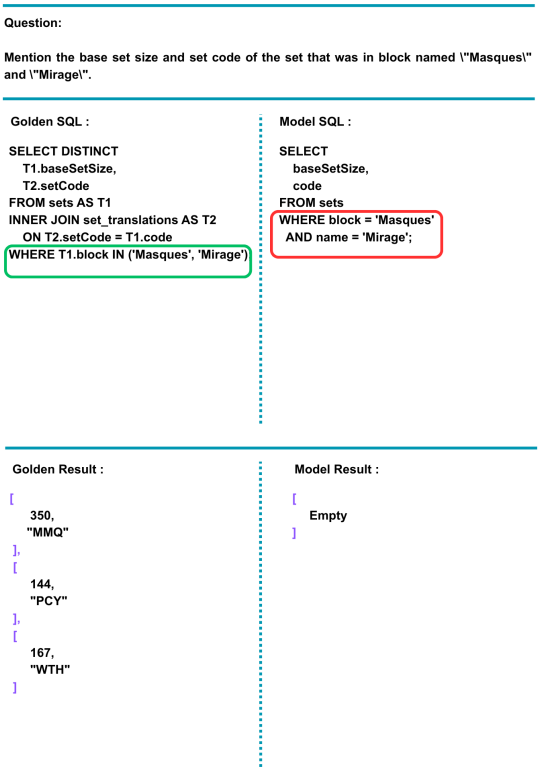

Figure A.4: Example of a wrong filter logic. The golden query interprets block named "Masques" and "Mirage" as sets whose block is either Masques or Mirage. The model query treats "Masques" as the block value, but treats "Mirage" as a set name.

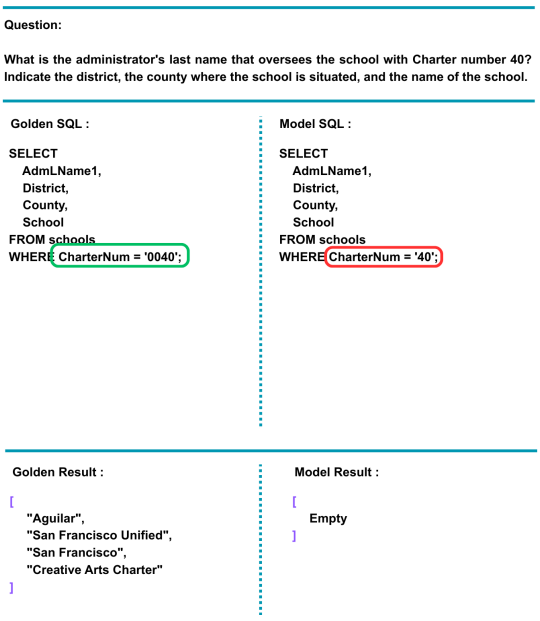

Figure A.5: Example of a wrong filter value. The model query uses the literal value '40', while the database stores the matching charter number as '0040', causing the filter to select the wrong records.

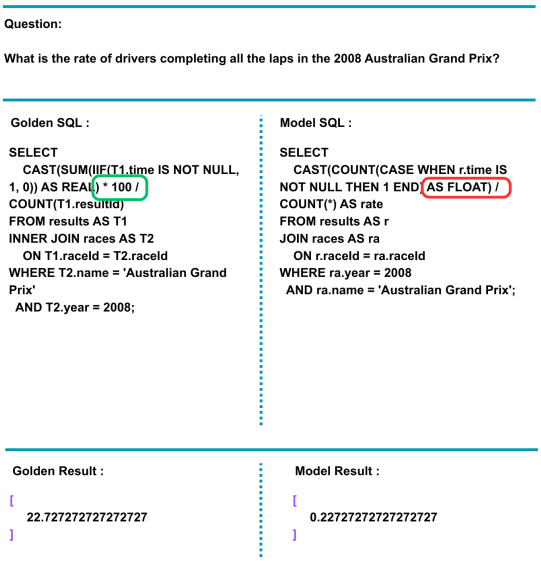

Figure A.6: Example of a calculation Error. The model query returns the result as a decimal proportion, whereas the golden query converts the value to a percentage. The error is therefore caused by an incorrect arithmetic transformation, specifically the missing percentage conversion.

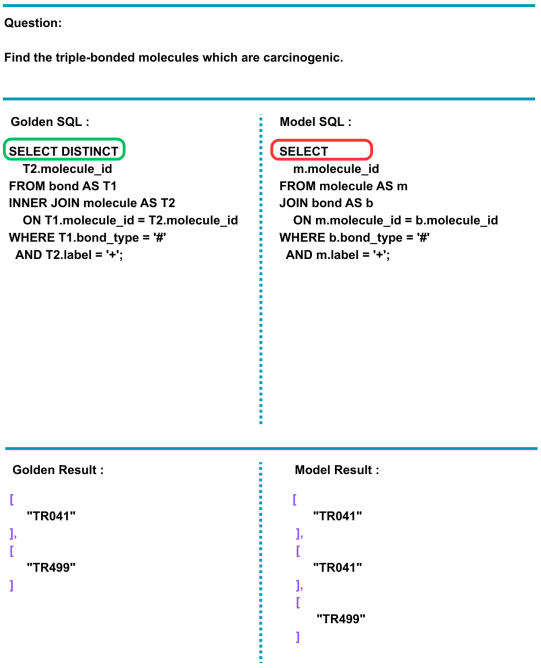

Figure A.7: Example of a duplicate handling error. The model query omits DISTINCT, causing the same molecule ID to appear multiple times when several matching bonds are found.

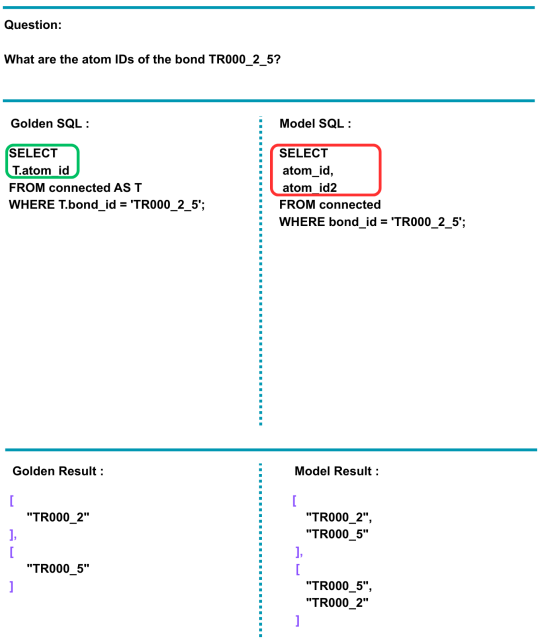

Figure A.8: Example of a projection error. The model query returns two endpoint columns, while the golden query expects one column containing atom IDs.

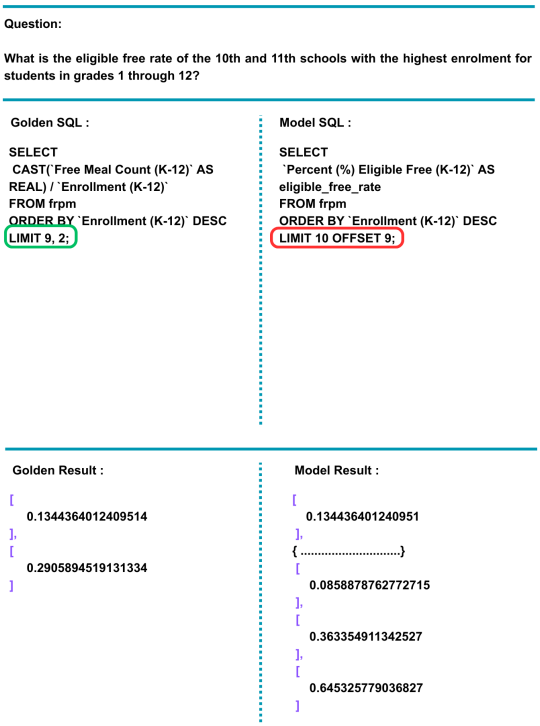

Figure A.9: Example of a row selection error. The model query uses an incorrect LIMIT/OFFSET combination and returns more ranked rows than the specific rows expected by the golden query.

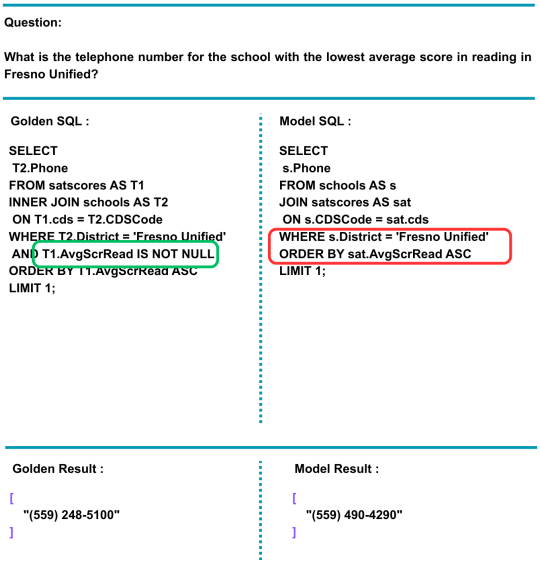

Figure A.10: Example of a NULL handling error. The model query omits the `AvgScrRead IS NOT NULL` condition, allowing rows with missing values to affect the result selection.

B

Reproducibility Elements

```
{
    Name of the model: "...",
    Question id: "...",
    Complexity level: "...",
    Golden SQL query: "...",
    Model SQL query: "...",
    Tables used by the model: ["..."],
    Output of executing model SQL query: [...],
    Output of executing golden SQL query: [...],
    Result of comparing the outputs: "...",
    Repetition number: "..."
}
```

Figure B.1: Structure of a logged record for a trial.

```
You are a schema selection assistant for Text-to-SQL.

Select the smallest set of tables needed to answer the
question. Include JOIN table names if needed. If unsure,
include the table name.

Rules:
- Output ONLY valid JSON.
- Format: {{"tables": ["table1", "table2"]}}
- Use ONLY table names from the allowed list.
- Do NOT include explanations or extra text.
- Do NOT include parentheses or anything besides the JSON.

Valid examples:
{{"tables": ["students"]}}
{{"tables": ["orders", "customers"]}}

Allowed tables:
{allowed_tables}

Question:
{question}

Schema:
{schema_text}
```

Figure B.2: Prompt template for table selection (Step 1 of least-to-most prompting).

```
You are a Text-to-SQL expert.

Using the selected tables below, the database
   clarifications, and the natural language question, write
    one SQLite SQL query that answers the question.

Selected Tables:
{', '.join(selected_tables)}

Database Schema for Selected Tables:
{schema_text}

Clarifications:
{case.get("evidence", "")}

Natural Language Question:
{case["question"]}

Return only the SQL query.
```

Figure B.3: Prompt template for SQL generation (Step 2 of least-to-most prompting).

Query Complexity	Simple	Moderate	Challenging
Question ids	82, 831, 554, 542, 695, 998, 799, 1134, 223, 69, 202, 422, 1410, 929, 546, 86, 950, 400, 299, 301, 1665, 844, 200, 318, 1505, 1314, 1512, 1361, 476, 748, 313, 389, 181, 614, 132, 1585, 61, 464, 496, 1129, 1673, 84, 1131, 502, 953, 54, 1004, 1575, 549, 1221, 756, 1313, 1363, 1045, 1145, 428, 1157, 13, 190, 1619, 1517, 1508, 143, 599, 225, 858, 809, 742, 103, 878, 900, 540, 1522, 1153, 396, 1133, 88, 311, 873, 713, 312, 902, 442, 697, 161, 1713, 564, 841, 580, 722, 10, 1197, 679, 430, 1536, 577, 323, 393, 369, 725	1506, 906, 222, 1106, 68, 1240, 970, 360, 761, 1182, 1375, 723, 1158, 100, 402, 466, 943, 1127, 1501, 766, 31, 615, 397, 1112, 1560, 1200, 185, 93, 958, 1316, 459, 408, 1291, 1430, 407, 1602, 40, 1475, 1327, 1225, 427, 732, 1098, 1440, 1146, 1395, 1121, 1548, 1529, 270, 1164, 1101, 508, 522, 822, 1474, 1229, 1162, 213, 1103, 236, 814, 1684, 255, 683, 1104, 1691, 894, 35, 1454, 1152, 152, 298, 511, 1289, 450, 188, 1252, 1264, 446, 391, 135, 175, 1170, 637, 1095, 899, 1274, 327, 1439, 753, 1125, 1388, 1047, 1093, 657, 272, 1401, 81, 1167	73, 173, 1242, 169, 1247, 1236, 730, 125, 818, 1190, 198, 1694, 268, 249, 319, 328, 1231, 1232, 1171, 1223, 1481, 62, 1037, 1161, 1339, 306, 1307, 834, 1457, 277, 772, 1239, 231, 371, 1295, 431, 1270, 634, 1115, 1183, 247, 1076, 835, 1359, 263, 1460, 28, 1241, 1041, 701, 1202, 1001, 994, 896, 1028, 1549, 1191, 1169, 1552, 744, 962, 290, 1114, 1302, 206, 1257, 477, 1036, 1192, 1168, 1437, 521, 212, 1476, 302, 1429, 1071, 307, 87, 1686, 819, 322, 506, 149, 775, 253, 1243, 1292, 639, 829, 334, 285, 337, 523, 513, 416, 1448, 94, 743, 1194

Table B.1: A full list of the 300 question identifiers for test cases sampled for this study, grouped by query complexity.

```
INPUT:
  - Dataset of cases
  - SQLite databases
  - LLM model
  - number of repetitions

OUTPUT:
  - JSONL file with predictions and evaluation results


FOR each selected case in the dataset DO
  (tables, schema) <-- ExtractDatabaseSchema(case)

  Prompt 1: ask the model to SelectRelevantTables(question
     , tables, schema)
  selected_tables <-- ParseTableList(model response)

  FOR each repetition from 1 to repetitions DO
    Prompt 2: ask the model to GenerateSQL(question,
       context, selected_tables)
    sql <-- CleanSQL(model response)

    TRY
      predicted_result <-- ExecuteSQL(case.db_id, sql)
    CATCH SQL error
      SaveFailureRecord(case, sql, selected_tables, "
         Hallucination" or "Syntactic")
      CONTINUE
    END TRY

    gold_result <-- ExecuteSQL(case.db_id, case.SQL)

    IF CompareResults(predicted_result, gold_result) =
       correct THEN
      SaveSuccessRecord(case, sql, selected_tables, "
         Correct", predicted_result, gold_result)
    ELSE
      SaveFailureRecord(case, sql, selected_tables, "
         Semantic", predicted_result, gold_result)
    END IF
  END FOR
END FOR
```

Figure B.4: Pseudocode of the experiment's script, executed once for each selected model.

C

Supplementary Statistical Results

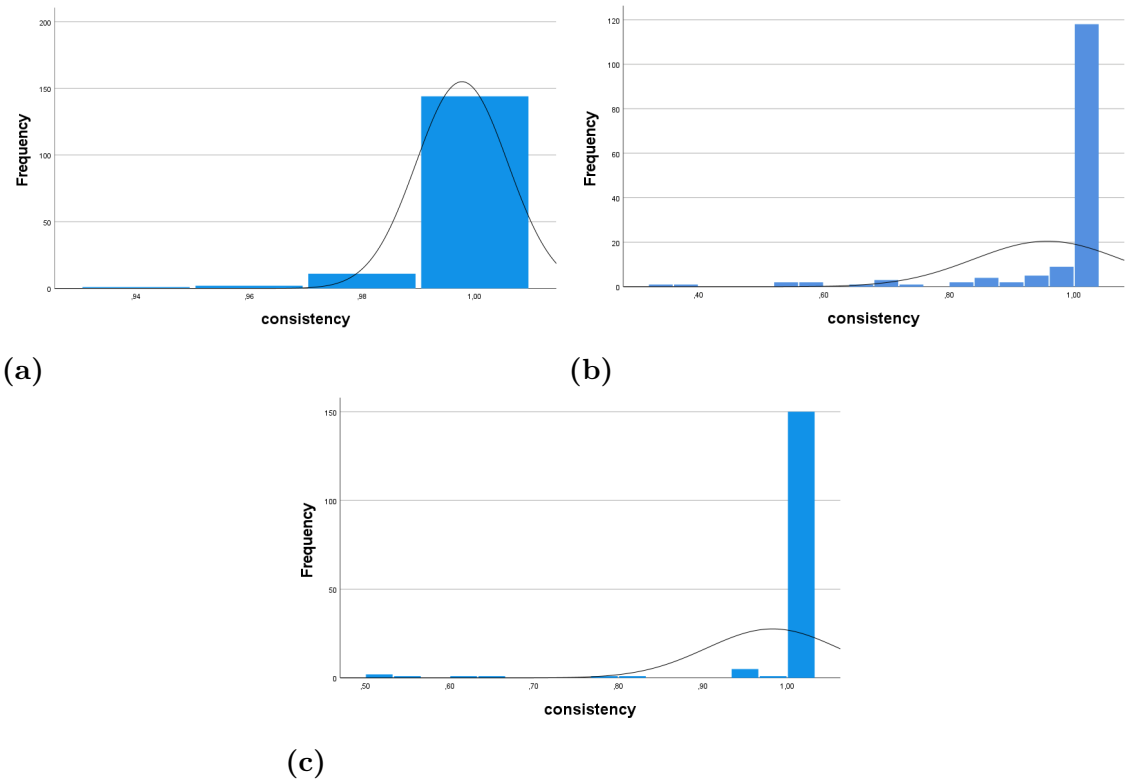

Figure C.1: Frequencies of consistency values for: (a) Qwen3 A3B; (b) Qwen3 A22B; (c) Qwen3 A35B.

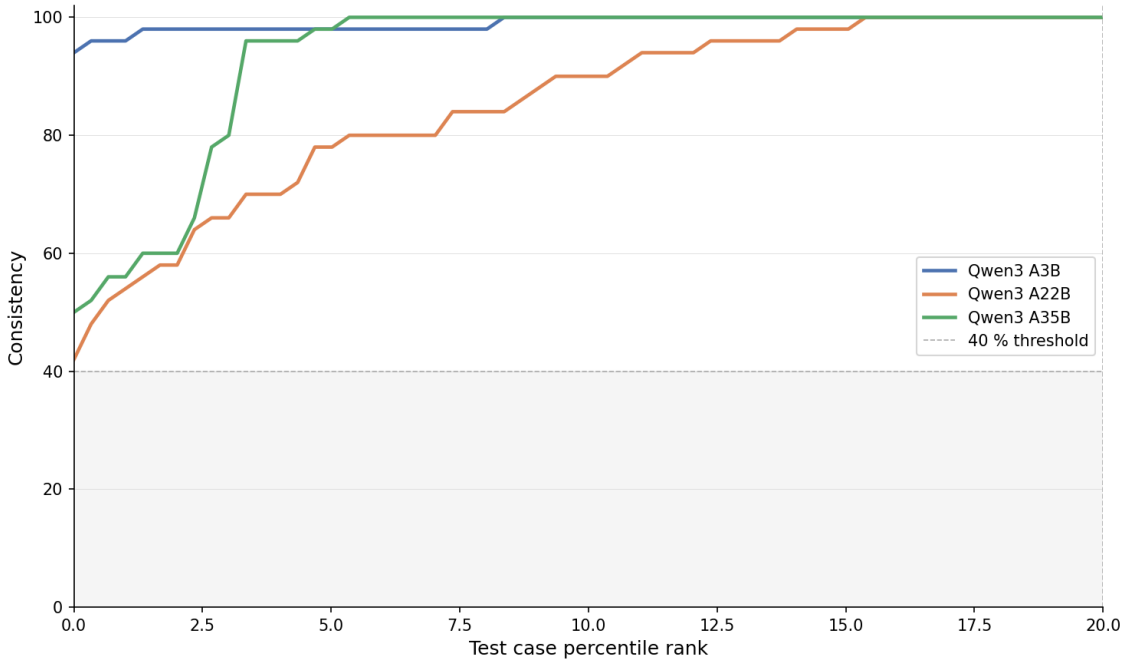

Figure C.2: Per-test-case consistency for the bottom 20% of test cases ranked by consistency.