

Evaluating Small and Large Language Models in a Self-Correcting Text-to-SQL Agent for Enterprise ERP Reporting

Sanjib Shah

Department of IT Engineering

Nepal College of Information Technology

Lalitpur, Nepal

sanjib.211533@ncit.edu.np

Aman Sheikh

Department of IT Engineering

Nepal College of Information Technology

Lalitpur, Nepal

aman.211506@ncit.edu.np

Sunil Nath

Department of IT Engineering

Nepal College of Information Technology

Lalitpur, Nepal

sunil.211443@ncit.edu.np

Nabin Paudel

Department of IT Engineering

Nepal College of Information Technology

Lalitpur, Nepal

nabin.211523@ncit.edu.np

Suyog Lamsal

Department of IT Engineering

Nepal College of Information Technology

Lalitpur, Nepal

suyog.211446@ncit.edu.np

Abstract—Retrieving data from enterprise relational databases requires SQL. Many business users do not have SQL skills, which drives the need for natural language to SQL systems. Most leading systems depends on large, cloud-hosted LLMs, which sits poorly with commercially sensitive ERP data and their surrounding retrieval and validation pipelines are rarely evaluated separately from the generation model itself. This article investigates whether a small locally deployed LLM can compete as a generation model in the same retrieval augmented, rule validated agent pipeline as larger models, against which the retrieval and validation pipelines also evaluated with the data. We built an agent for an on-premise Oracle-based Synergy ERP schema, using BGE-M3 embeddings over Qdrant vector database to index schema and example retrieval, and a deterministic validator with bounded self-correction before execution. We use and compare four interchangeable LLMs: locally run Qwen2.5-Coder-14B (4bit quantization) and cloud run Mistral Large, Minimax M3 and Qwen3.8-Max based on semantic equivalence, execution accuracy, component-level F1, row-count matching, and generation success rate. Qwen3.8-Max performed best on every metrics, while MiniMax M3 outperformed the larger Mistral Large. The much smaller Qwen2.5-Coder-14B matched or exceeded Mistral Large on several metrics. Schema retrieval was reliable enough with Hit@5 = 0.93. An ablation comparing before and after validation and SQL correction shows the largest accuracy gains for weaker models, with limited or slightly negative effect for the strongest model. The locally hosted model was also the fastest (2.671 s/query) and cheapest (\$0.0011/query) among the four models. These findings suggest model scale does not fully determine performance within a fixed pipeline, supporting compact local models as a practical, low-cost, low-latency alternative for enterprise deployment.

Keywords—Text-to-SQL, Retrieval-Augmented Generation, SQL Generation, Large Language Models, enterprise resource planning, Natural Language Processing.

I. INTRODUCTION

Structured business data in operational enterprise resource planning (ERP) systems to analytical warehouses is still

mostly stored in relational databases. In almost all cases, Structured Query Language (SQL) is necessary to retrieve that information. While SQL is one of the most widely used languages used in professional software development, it is a specialized skill many of the people who need reports (such as domain staff, analysts and managers) are not capable of writing the joins, filters and aggregations that their questions imply [1], [2]. This mismatch is resolved by a system referred to as natural-language-to-SQL (text-to-SQL). They are given a query in natural language and a description of the database schema and generate a runnable SQL query [1], [3]. If the translation is accurate, the user receives his desired result without having to learn SQL syntax or the internal names of tables and columns.

The task has been widely used in the field of natural language processing as a semantic parsing task and is also investigated as a natural language interface to databases (NLIDB) [2], [3]. A correct mapping should recover the user's intent, find out the appropriate schema elements and generate syntactically valid SQL for the destination engine and semantically similar to the user's question [1], [4], [5]. This mapping has been measurable on a scale, thanks to public benchmarks. WikiSQL introduced large-scale single-table evaluation [6]. Spider added complex, cross-domain, multi-table queries over unseen schemas and BIRD further stressed models with large, noisy databases and an efficiency aware metric as well as execution accuracy [5]. With the recent success of large language models (LLMs) in being capable of reporting high accuracy on these benchmarks by force filling a database table with generated SQL via prompting or fine-tuning, frequently without a task-specific parser this has gained much attention [7], [8], [9].

Several known challenges continue to constrain the reliability and practical deployment of text-to-SQL systems, includ-

ing ambiguous natural-language queries, complex database schemas, schema linking, and the generation of semantically correct SQL [1], [2], [10]. Natural language questions are often under specified or ambiguous, e.g., “sales this year”, “top customers”, leave the calendar, the measure and the grouping unspecified [1], [2]. Accurate generation also requires schema linking, namely matching the tokens in the question with the appropriate tables, columns and join keys, a task that is difficult and challenging when the names of the entities are abbreviated, ambiguous or not the user’s business jargon [11], [12]. Nested subqueries or window functions are required for some questions or features that do not exist in the training data, require special syntax for the dialect [3], [13]. Often a model that works well on one family of databases will not work so well on another; this is a setting that Spider and BIRD are designed to measure, the cross-domain setting [4], [5]. These pressures are even stronger in enterprise schemas, where local reporting conventions, vendor dialects and multi-tenant filters further limit the possibilities of how a generated query can look.

A. History and Evolution of Text-to-SQL

The last decade has seen significant changes in how text-to-SQL is deployed. Early systems utilized toolkits where the programmer had to write grammar. Later systems trained neural sequence models. Then, others employed schema-aware and pre-trained encoders as the most recent systems use LLMs for in-context learning to generate SQL. Recent surveys and tutorials further describe this evolution as a progression from rule-based and neural approaches toward PLM-, LLM-, and agent-based NL2SQL systems, while emphasizing evaluation, debugging, and practical deployment challenges [10]. Fig. 1 describes this trajectory in approximately five epochs. The dates indicate when the techniques described first became prominent and are given an approximate span of one year. This segmentation is based on recent evolutionary accounts of the field [1], [2], [10], [14].

1) *Rule-Based Era (~2015)*: Early text-to-SQL systems mapped natural language to SQL with manually crafted grammars, templates, and heuristic patterns [15], [1]. A typical design enumerated expected question forms, bound linguistic constituents to schema elements through dictionaries or hand-written rules, and filled a SQL template. Such systems could be reliable on a narrow, well-specified database. For example, a single reporting schema with a closed set of question types—because the designer controlled both the language coverage and the SQL they would emit [3], [15]. Their weakness was scalability. Each new domain, join path, or linguistic variant required additional rules, and ambiguous or nested questions quickly exceeded what a static grammar could cover [1], [3]. As databases grew more heterogeneous, this engineering cost motivated a shift toward models that could learn the mapping from examples rather than from a complete specification of the language.

2) *Neural Era Begins (~2019)*: Around the late 2010s, text-to-SQL was recast as a sequence-to-sequence learning

problem. Seq2SQL trained a neural model with reinforcement learning to generate SQL from questions over Wikipedia tables and released WikiSQL as a large single-table benchmark [6]. SQLNet showed that a sketch-based decoder could avoid some of the sequential prediction errors of full sequence generation [16]. Intermediate representation models such as IRNet then targeted the harder cross-domain setting formalized by Spider, in which the system must generalize to databases not seen during training [4], [17]. These neural models were an important change they learned alignments from data, handled a broader range of phrasings than rule sets and made progress measurable on shared leaderboards [3], [4]. They remained limited however, in how they represented the database itself. Many treated the schema as a flat list of names rather than as a graph of tables, keys and relations, which made multi-table joins and schema generalization brittle [11], [18].

3) *Context-Aware Modeling (~2021)*: The subsequent wave of work used the schema as a structured context instead of just as unstructured text. In fact, RAT-SQL explicitly learned a question schema alignment, via relation-aware self-attention, and found that much of the improvement on Spider stemmed from the ability to handle schema linking and not from the decoder size. To enable the parsing of column names and cell values to aid in text-to-table data transformation, table aware encoders were developed, which were co-trained with text and tabular data [12]. Other neural network-based approaches, such as graph neural models (e.g., LGESQL) used tables, columns, and question tokens as nodes and the relation of the entities as the messages between them [18]. All of these techniques enabled cross-schema generalization to become more systematic, going beyond simply copying names from the question to reasoning over the relational structure of the target schema [2], [3]. The remaining errors focused on uncommon SQL operators, join paths that were long, and questions that were different from the schema context under which a purely supervised encoder had limited vocabulary [4], [13].

4) *Pre-Trained Model Era (~2023)*: The next generation of models were pre-trained language models (PLMs), and they became the standard backbone. The BERT-style encoders, which are already good at understanding general language, were adapted for schema linking and SQL decoding [19]. Encoder-decoder models like T5 were constrained at decoding time so that only grammatical SQLs are generated, as in PICARD [13]. Furthermore, the linking of schemas was separated from the parsing of skeletons, thus decreasing the effort required to predict a complete query on one pass, which was part of RESDSQL [20]. The fine-tuned PLMs performed better in terms of exact-match and execution accuracy on Spider and its robustness variants, while being able to deal with more complex natural language than the previous LSTM-based parsers [1], [3], [20]. They still generally needed significant amounts of task-specific labeled data, however and their scale of parameters meant that they were not highly few-shot flexible, or easily able to execute long, compositional instructions without further fine-tuning [2], [8]. Those restrictions led to

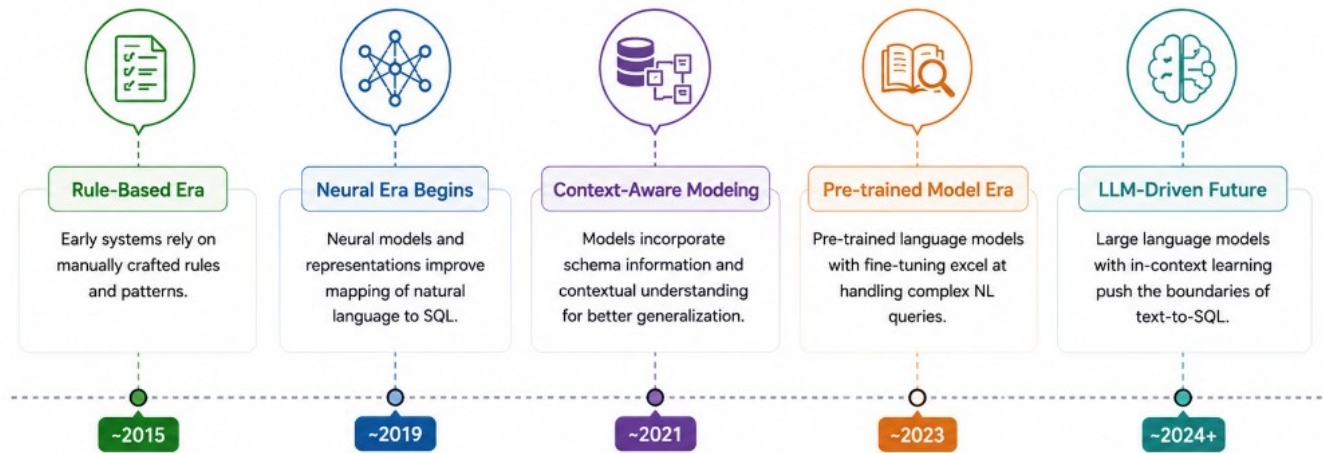


Fig. 1. Evolution of text-to-SQL systems from rule-based parsers to LLM-driven in-context learning. Each era is labeled by an approximate year and a dominant modeling style. Dates are indicative, based on when key methods appeared.

the development of much bigger models which allowed for schema to be steered via prompts rather than a new training run for each one.

5) *LLM-Driven Era (~2024+)*: The current period is dominated by LLMs that generate SQL through in-context learning (ICL) and in a parallel line of work, through fine-tuning of open models. Early evaluations showed that Codex and GPT style models could produce usable SQL from a schema and a question with little or no gradient updates [7], [8], [9]. Subsequent systems improved this baseline by decomposing the task. DIN-SQL separates schema linking, query classification, generation, and self-correction [21]. DAIL-SQL studies how few-shot examples should be selected, organized and C3 applies a zero-shot ChatGPT pipeline with calibration and consistency voting [22], [23]. Agent-style designs go further MAC-SQL assigns selection, decomposition and refinement to cooperating agents and CHESS uses retrieval, schema pruning, candidate generation and unit testing to keep generation tractable on large catalogs [24], [25]. Interactive refinement methods such as SQL-CRAFT close the loop with database execution feedback [26]. Retrieval-augmented generation, originally proposed for knowledge-intensive NLP, is now commonly used to supply schema fragments and SQL exemplars so that the model need not recall an entire catalog from its parameters [27], [25], [28]. Open models have been specialized for the task in parallel CodeS incrementally pre-trains compact code models (1B–15B) for SQL generation and DTS-SQL decomposes fine-tuning so that 7B-class models can approach proprietary ICL systems on Spider and BIRD [28], [29]. Surveys of this literature emphasize that the strongest published numbers still tend to come from large, often closed, models, while computational cost, robustness and data privacy remain open concerns [1], [2].

But these results raise several practical questions, particularly when deployed as an agent on a public leaderboard

pipeline for context retrieval, SQL generation, SQL validation, and even a possible retriage as opposed to a one-shot parser on a public leaderboard. Even in the best of the leading ICL systems, the initial model used remains based on the GPT class models accessible via external APIs [21], [23], [25]. The design doesn't work very well with ERP data, which is commercially sensitive, affected by residency rules and costly to move off-site at reporting volume [1], [2], [28]. While some models have closed the accuracy gap on Spider and BIRD with task-specific pre-training or decomposed fine-tuning [28], [29]. The comparison is usually also performed with a different model and method and the models are not evaluated on a production schema but on public benchmarks. What happens if a locally installed 10B–15B class generator is deployed without fine-tuning and shares the retrieval, prompting and validation stack as larger cloud models found on the cloud?

One of the drawbacks of this is that validation is seldom assessed separately. Self-correction, execution guided refinement and multi-agent refinement have become ubiquitous but are typically lumped together in an overall accuracy figure [21], [24], [25], [26]. A deterministic checker has the potential to increase execution accuracy by preventing illegal tables, columns or syntax and can reduce the probability of generation success by refusing a syntactically correct query, such as a derived alias. Those two effects work in opposite directions and the failures are rarely blamed on the generator as opposed to the checker. The system of metrics is also somewhat unstable for agents. There are known failure modes for each of the notions of correctness captured by exact set matching, execution accuracy, BIRD's valid efficiency score, and test-suite accuracy [4], [5], [30]. These were created for parsers that output a single SQL string. The results of such an agent are more than just a leaderboard score. They include validator rejections, oracle runtime errors, and equivalent but non-identical result sets.

This paper studies those questions in one controlled setting: a retrieval-augmented, rule-validated text-to-SQL agent for reporting over an on-premise Synergy ERP schema hosted on Oracle. The generator is interchangeable; schema and SQL-example retrieval with BGE-M3 over Qdrant and a deterministic validator with bounded correction are held fixed [27]. We evaluate 4-bit quantized Qwen2.5-Coder-14B, served on site, against substantially larger cloud-hosted models on 100 held-out questions with gold SQL [31]. Generated queries are checked for safety, schema, structure, multi-tenant filters, and Oracle 11g compatibility, with validation error resolved and fed back for at most four retries. Alongside execution accuracy we report semantic equivalence, component matching, row-count matching and generation success rate and we use the gaps among these scores to interpret agent behavior.

The contributions of this paper are as follows:

- A same-pipeline comparison of a small, locally hosted model with much larger cloud-hosted language models on a real ERP reporting workload, evaluating accuracy, per-query latency and estimated API cost, without task-specific fine-tuning.
- An analysis of how a deterministic validation and bounded-correction loop improves execution accuracy, component matching, generation success, and failure attribution.
- An evaluation protocol for text-to-SQL agents that reports semantic equivalence, execution accuracy, component-level F1, row-count matching, and generation success together, rather than a single one-shot leaderboard metric.

II. METHODOLOGY

A. System architecture of the proposed text-to-SQL framework

The proposed system provides a controlled natural language to SQL pipeline where users request are translated to executable SQL query to an Oracle database [1], [2]. The architecture is sequential, as illustrated in Fig. 2, for SQL generation, validation, column resolution and execution, along with retrieval of schema, SQL-example, query-intent, conversational context and caching. This follows retrieval-augmented generation over schema and worked SQL examples [27], [25], together with schema linking so that only relevant tables and columns are shown to the generator [11], [20]. User question is first used to retrieve relevant schema and SQL query which are incorporated with the query intent and session context to construct the LLM prompt [22]. The SQL is then passed to a validation layer, which validates it against the schema retrieved from the server and the query constraints that are applicable [13].

The validation layer exists as a control boundary between the generation of SQL and the Oracle database. If validation finds an invalid column, the Column Resolver will try to resolve a possible column mapping and the information will be provided to the LLM along with the validation error for the purpose of correcting the query [11], [20]. This generate-validate-correct process continues until the query is validated

or rejected (maximum 4 times), following bounded self-correction with explicit error feedback rather than an unguided retry [21], [26], [32], [33]. A validated query is then limited as per the requested row limit before being executed against Oracle, and the number of rows, columns or execution error returned to the application. Interactions are also stored in conversational memory and cached in Redis, which helps with faster responses to repeated interactions and allows for contextual follow-up requests [34], [35].

B. Query Understanding and Intent Extraction (Query Processing; Intent + context understanding)

The query-processing stage converts the user's natural language query into a structure of the intended query. Representing the request as an intermediate structure before SQL generation follows the same idea as intermediate-representation parsers [17] and decomposed prompting pipelines that separate intent from SQL synthesis [21]. The incoming question is first normalized and analyzed to determine the entities that are relevant, temporal constraints, ordering requirements and cues regarding the number of rows [1], [3]. This preprocessing offers input to the differentiation of the user's requested business concepts from incidental wording and to prepare the query for structured intent extraction.

Then, the processed query is fed to an LLM that extracts the appropriate query requirements based on the pre-defined extraction rules [21], [23]. These rules represent more temporal semantics, such as date formats and date ranges, ordering semantics, such as ascending or descending, row-limit expressions, such as top N or first N and the attributes that the user explicitly requested in the output. The intent is then stored as structured JSON with the following properties: date applicability, date range, ordering column, ordering direction, row limit, required output columns. For instance: Total Invoice worth highest 10 customers from 2025 to 2026 is transformed to applicable B.Order the results by TOTAL_INVOICE_VALUE descending, request 10 rows and get the CUSTOMER_NAME field. This structured representation later passes into the downstream SQL generation process, providing an explicit representation of the user's needs, instead of just the original natural-language query.

C. Knowledge Base Construction and Schema Enrichment

This is the first offline stage after the architecture overview. Two stores are built once from the production Oracle Synergy ERP database so that later retrieval never queries the live system. Fig. 4 shows the two paths: schema enrichment on the left, and the SQL example corpus on the right. Both end in Qdrant. The design follows retrieval-augmented text-to-SQL systems that ground generation in retrieved schema fragments and exemplars rather than the full catalog [27], [25].

The left path starts with a static extract of tables, columns, and keys. There are twenty seven tables related to sales reporting in oracle Synergy database. Each table is passed to a locally served Qwen2.5 model [31], which adds a purpose and a business reading of the columns while leaving names

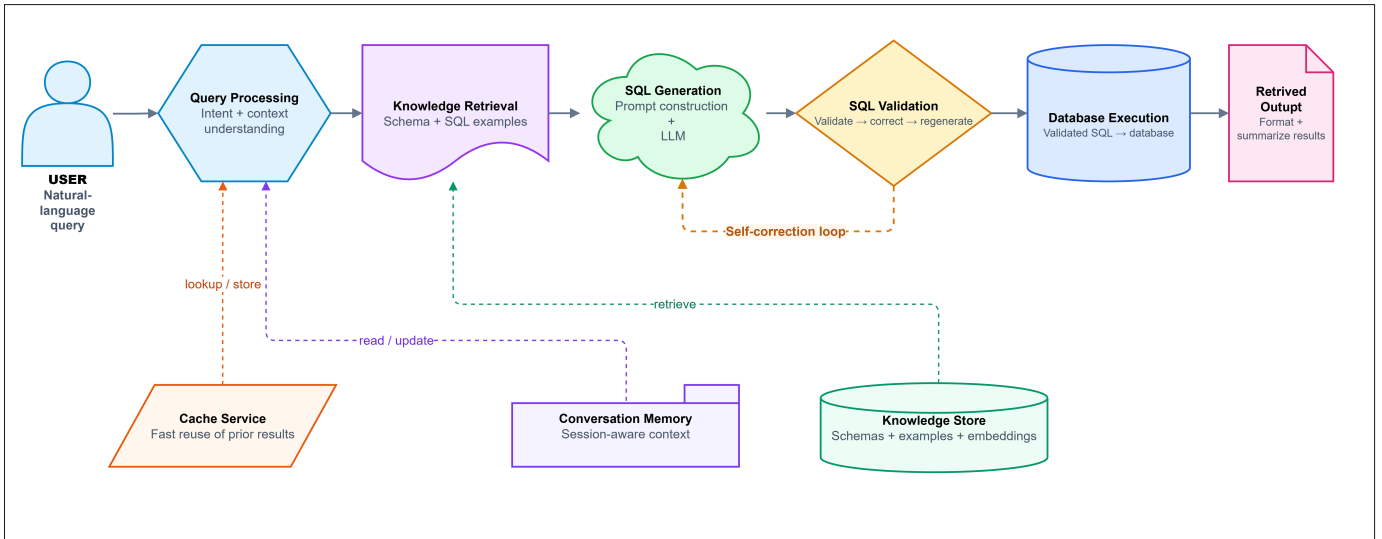


Fig. 2. System overview and architecture design.

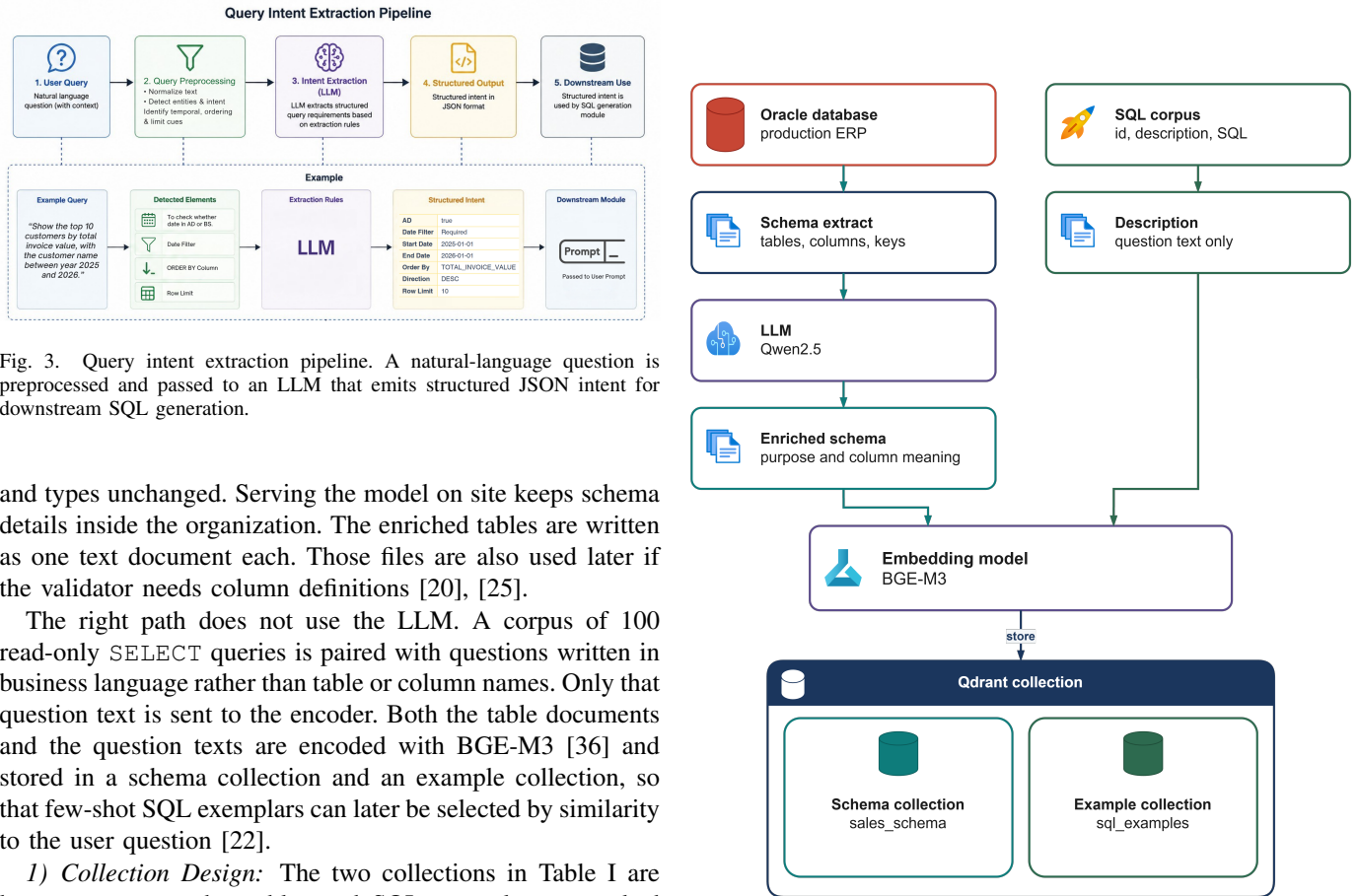


Fig. 3. Query intent extraction pipeline. A natural-language question is preprocessed and passed to an LLM that emits structured JSON intent for downstream SQL generation.

and types unchanged. Serving the model on site keeps schema details inside the organization. The enriched tables are written as one text document each. Those files are also used later if the validator needs column definitions [20], [25].

The right path does not use the LLM. A corpus of 100 read-only `SELECT` queries is paired with questions written in business language rather than table or column names. Only that question text is sent to the encoder. Both the table documents and the question texts are encoded with BGE-M3 [36] and stored in a schema collection and an example collection, so that few-shot SQL exemplars can later be selected by similarity to the user question [22].

1) *Collection Design*: The two collections in Table I are kept separate, so that tables and SQL examples are ranked on their own [27]. Tables are stored as whole documents. Examples are indexed from the question text only [22].

D. Vector Indexing and Knowledge Retrieval Infrastructure

The artifacts from Section II-C are indexed offline in Qdrant so that a user question can retrieve related context by meaning

Fig. 4. Offline construction of the knowledge base. Schema is taken from Oracle, enriched with Qwen2.5-coder-14B, and stored in the schema collection. SQL descriptions are stored in the example collection. Both are encoded with BGE-M3.

TABLE I
QDRANT COLLECTIONS AND THEIR CONTENTS.

Collection	Indexed text	Search mode	Top- k
sales_schema	Full table document	Cosine similarity	5
sql_examples	NL description only	Cosine similarity	5

[27]. Documents and questions are encoded with the same locally served BGE-M3 model, which handles English business phrasing and Nepali date expressions through multilingual dense retrieval [36]. Fig. 5 shows the path where the question is encoded once and cosine similarity is used to search the schema and example collections.

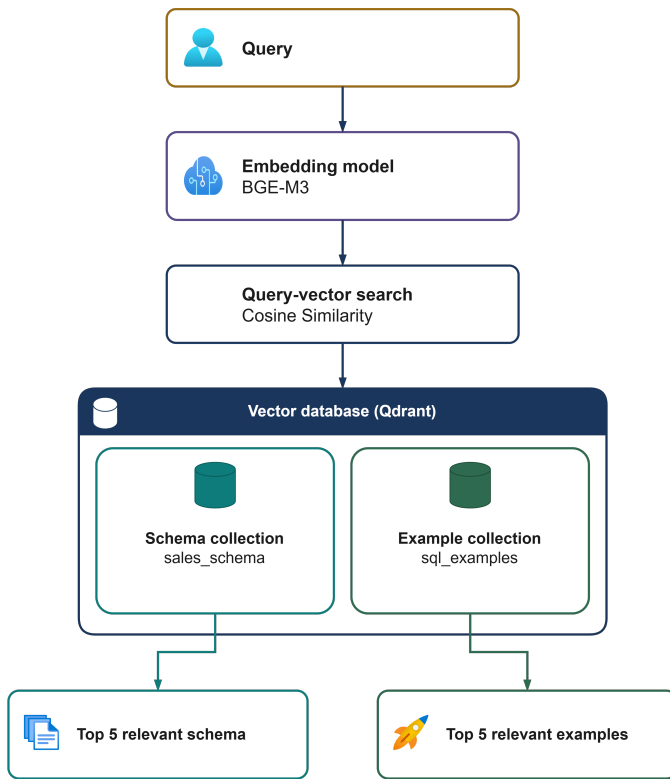


Fig. 5. Query-time retrieval over the Qdrant collections using cosine similarity.

The schema search returns the five nearest tables. Each hit gives the table name, purpose, keys and columns, which tells the generator which structures it may use. Restricting the prompt to retrieved tables is the retrieval-era analogue of schema linking [11], [20], [25].

The example search returns the five nearest SQL pairs. Each hit gives a business description and a reference `SELECT` SQL query, which shows the reporting pattern to follow [22].

Both outputs are passed on as structured fields. The prompt is built from those fields, not from the fully stored documents.

E. SQL Generation: Prompt construction and LLM Based Generation

SQL generation translates the contextualized query and information that the user has retrieved to an initial SQL statement. The query context, schema metadata retrieved, query intent and retrieved SQL examples are passed to the generation stage. The prompt builder takes these as inputs and applies them to a system prompt that includes instructions for generating SQL and constraints for the database, creating the messages that are fed to the chosen LLM [7], [8], [23]. Few-shot exemplars are organized in the prompt following evidence that example selection and ordering affect text-to-SQL accuracy [22]. The LLM can be either installed on site or offered as a cloud-based service, which matches recent work on compact open code models and decomposed small-model text-to-SQL [28], [29], [31]. This produces a cleaned up SQL that is sent to the rule-based validation layer.

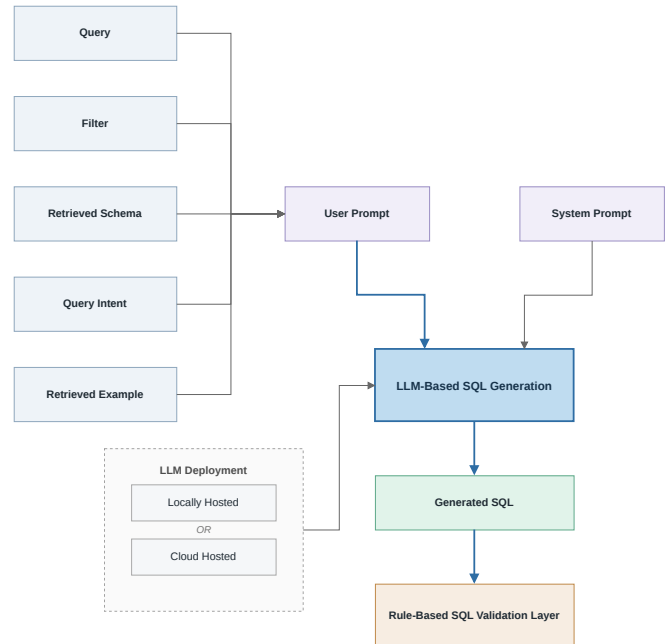


Fig. 6. SQL generation pipeline. Query context, retrieved schema, query intent, and SQL examples are combined with the system prompt and supplied to an LLM deployed locally or through a cloud provider. The generated SQL is then forwarded to the rule-based validation layer.

The system prompt is paired with query context, retrieved schema, query intent and SQL examples and passed to an LLM that is locally deployed or provided by a cloud services provider [31], [9]. The SQL generated is then passed to the rule-based validation layer.

F. Deterministic Rule-Based SQL Validation

Generated SQL does not run directly, but is instead sent through a deterministic rule-based validation layer which validates safety, schema, structural, contextual and Oracle 11g compatibility constraints. Checking generated SQL before execution is related to constrained decoding, which rejects

illegal SQL during generation [13], and to rule-based query processing that enforces a permitted statement class [15]. The validator preprocesses the SQL and checks that the SQL contains only statements for which there is a valid read-only operation, as well as valid Oracle syntax. It then validates referenced tables, columns, aliases and scope of a nest query using schema information available [11], [20], then validates required contextual constraints like date ranges and COMPANY_CODE/BRANCH_CODE filters.

If the validation is successful, the query will then go on to be executed. Wherever an invalid query is used, the validator creates a descriptive feedback for each specified error. This validation error, feedback, schema evidence, and query context are then passed to the LLM for correction, followed by re-validation in a bounded retry process, to which the corrected SQL is passed [21], [26]. Unlike LLM-only self-critique, in which the same model both generates and reviews the SQL [32], the checks here are deterministic and do not invoke a second LLM. Execution-guided methods similarly use a concrete failure signal rather than an unguided retry [33].

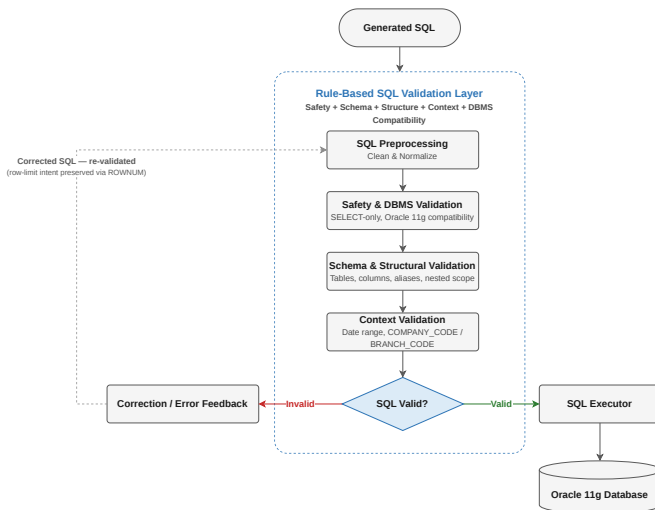


Fig. 7. Rule-based SQL validation and correction pipeline.

Fig. 8 is an example of a validation and correction cycle for the query “Show the total invoice value and count of invoices for each item category.” The first SQL is wrongly using the identifier `i.CATEGORY_CODE` that is not included in the `SA_SALES_INVOICE` table. The column resolver finds evidence for the schema, such as `IP_ITEM_MASTER_SETUP.CATEGORY_CODE`, which is passed to the LLM in addition to the validation error and query context. This is a directed form of schema linking: the missing column is located on another table rather than guessed [11], [20]. The LLM then produces a corrected query using `im.CATEGORY_CODE`, which is subsequently re-validated [21].

G. Execution and Result Handling

The execution is performed on the `oracledb` driver in thin mode. One call will open a connection, runs the statement once as a read-only output retrieval, fetches the column names from the cursor metadata and the rows as typed fetch, and closes the connection. At this scale, it was consciously accepted to open a new connection for each query, as pooling would imply that the stateless API-layer may not be safely shared.

These results are normalized to a fixed contract prior to returning it upstream shown in Table II. The failures are not raised into the pipeline, they are collected as tagged outcomes and the API will report to which part of the pipeline the request failed. Execution against the live database is the operational counterpart of execution-accuracy evaluation used on public text-to-SQL benchmarks [4], [5].

TABLE II
EXECUTION RESULT CONTRACT.

Field	Content on success	Content on failure
success	true	false
columns	Names from cursor description	empty
rows	Fetches tuples	empty
row_count	Number of rows	0
error	none	Exception message

In addition to the own failure channel, this results in two-stage error attribution: failures detected before execution are the validators’ failures, and failures detected during the execution of the database are the oracles’ failures. The difference allows evaluation to differentiate between generation faults and infrastructure faults, which execution-guided decoding also separates by using the database engine as a correctness signal only after a candidate is formed [33], [13].

H. Evaluation Metrics

Eight evaluation metrics were used to evaluate generated SQL from different angles like outcome-level, structural and cardinality-level correctness [3], [4], [5], as well as table-retrieval quality:

a) *Semantic Equivalence*: It verifies if the generated SQL is semantically equivalent to the gold SQL. A query is considered equivalent if that answer is the same as the gold answer, otherwise an LLM-based semantic judge using Deepseek v4 pro model [31] determines equivalence, permitting minor differences (e.g., aliasing, predicate ordering, logically equivalent joins/expressions, etc.), and disallowing changes that alter the answer. Using an LLM as a judge for equivalence follows the LLM-as-a-judge protocol [37]; test-suite evaluation likewise treats logical equivalence beyond exact string match [30].

b) *Execution Accuracy*: Execution accuracy is the standard outcome-level metric on Spider and BIRD [4], [5]. Each query will run both G_i and P_i on the database. If the generated

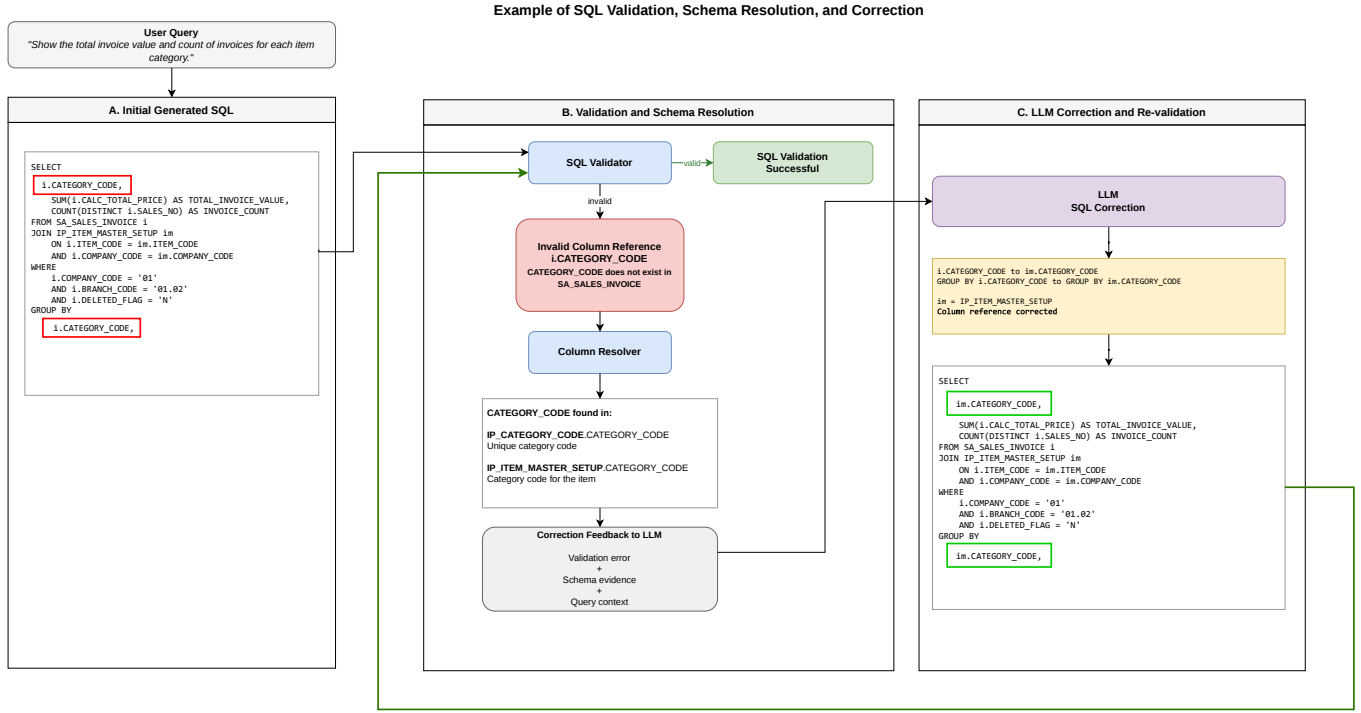


Fig. 8. Example of SQL validation, schema resolution, and LLM-based correction.

query returns the same compared result as the gold query, then let $E_i = 1$; else let $E_i = 0$.

$$E_i = \begin{cases} 1, & \text{if Result}(G_i) = \text{Result}(P_i) \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

The execution accuracy is

$$EA = \frac{\sum_{i=1}^N E_i}{N} \times 100. \quad (2)$$

c) Component Matching: SQL is evaluated at the component level, following exact-set matching of SQL clauses used on Spider [4], [3]. Let G_c be the set of gold components and P_c be the set of predicted components for SQL component c . The matched components are

$$TP_c = |G_c \cap P_c|. \quad (3)$$

The component-level precision, recall, and F1-score are

$$\text{Precision}_c = \frac{TP_c}{|P_c|}, \quad \text{Recall}_c = \frac{TP_c}{|G_c|}, \quad (4)$$

$$F1_c = \frac{2 \times \text{Precision}_c \times \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c}, \quad (5)$$

where c represents a SQL component such as: table, columns, WHERE condition, JOIN, GROUP BY, ORDER BY, aggregation.

d) Row-Count Matching: Row-count matching is a cardinality check on executed results, complementary to strict execution accuracy [4], [5]. Let R_i^G represent the number of rows returned by the gold SQL and R_i^P represent the number returned by the generated SQL. Define

$$C_i = \begin{cases} 1, & \text{if } R_i^G = R_i^P \\ 0, & \text{otherwise} \end{cases} \quad (6)$$

Then the row-count match rate is

$$\text{RCM} = \frac{\sum_{i=1}^N C_i}{N} \times 100. \quad (7)$$

e) Generation Success Rate: Generation Success Rate is the ratio of queries that the model is able to generate an SQL query without a generation error, capturing validity independently of result match [7], [13]. Let

$$S_i = \begin{cases} 1, & \text{if SQL generation for query } i \text{ is successful} \\ 0, & \text{otherwise} \end{cases} \quad (8)$$

Then

$$\text{GSR} = \frac{\sum_{i=1}^N S_i}{N} \times 100. \quad (9)$$

f) Hit@5: Hit@5 measures whether all gold (ground-truth) tables for a query are present anywhere within the top-5 retrieved tables; it is a strict, all-or-nothing metric per query. Let G_q be the gold table set and $R_q^{(5)}$ the set of top-5 retrieved tables for query q . Then

$$\text{Hit@5} = \frac{1}{|Q|} \sum_{q \in Q} \mathbb{1}[G_q \subseteq R_q^{(5)}], \quad (10)$$

where the indicator is vacuously 1 if $G_q = \emptyset$.

g) *Recall@5*: Recall@5 measures the fraction of a query's gold tables that were successfully retrieved within the top-5 results, averaged across all queries:

$$\text{Recall@5} = \frac{1}{|Q|} \sum_{q \in Q} \frac{|G_q \cap R_q^{(5)}|}{|G_q|}. \quad (11)$$

h) *Mean Reciprocal Rank (MRR)*: MRR measures how early the first relevant gold table appears in the ranked retrieval list, averaged across all queries; higher values indicate that relevant tables surface earlier. Let rank_q be the position of the first gold table in the ranked retrieval list for query q . Define

$$\text{RR}_q = \begin{cases} \frac{1}{\text{rank}_q}, & \text{if a gold table is retrieved} \\ 0, & \text{otherwise} \end{cases} \quad (12)$$

Then

$$\text{MRR} = \frac{1}{|Q|} \sum_{q \in Q} \text{RR}_q. \quad (13)$$

III. RESULTS AND DISCUSSION

A. Experimental Setup and Evaluation Protocol

The Text-to-SQL agent was tested against a set of 100 natural-language queries from a held-out set of sales-report evaluation benchmark queries created for this study. The natural language question and a fixed Oracle database gold standard query that is executable in SQL are included in each query record, as well as company and branch code filters that are used to limit the scope of the query. The queries are created depending on their difficulty level (Easy, Medium, Hard), which helps to evaluate the agent in a balanced manner.

The four language models were assessed as the SQL-generation engine of the agent, as summarized in Table III.

TABLE III
EVALUATED GENERATION MODELS.

Model	Hardware/Infra	Parameters
Qwen2.5-Coder-14B	NVIDIA GeForce RTX 4080	14.7B
Mistral Large	cloud provider: xKiro	675B
MiniMax M3	cloud provider: xKiro	428B
Qwen3.8-Max	cloud provider: xKiro	2.4T

The four models were evaluated using the same set of 100 queries, the same retrieval, prompting, generation, validation pipeline and the same downstream evaluation harness, so that differences in observed performance are due to the generation model itself, and not to the evaluation protocol. Table IV summarizes the overall experimental configuration.

B. Overall Performance Comparison

The overall results of all four models for all five evaluation metrics are shown in Fig. 9.

Qwen3.8-Max outperforms all other models on every metric evaluated, including a 100% generation success rate indicating

TABLE IV
EXPERIMENTAL CONFIGURATION.

Parameter	Value
Evaluation set size	100 queries
Query difficulty levels	Easy, Medium, Hard
Target database	Oracle
Models compared	4 (see Table III)
Semantic judge model	DeepSeek V4 Pro
Embedding model (retrieval)	BGE-M3
Decoding temperature	0.1
Retrieval depth (top- k)	5
Prompting strategy	Few-shot
Metrics	Semantic Equivalence, Execution Accuracy, Component F1, Row-Count Match

that it always produced a executable SQL statement across the 100-query sample set, though this reflects generation reliability rather than correctness, which is separately captured by the other four metrics. MiniMax M3 is the second-strongest model overall, and despite Mistral Large's much larger parameter footprint, it performs notably better on execution accuracy (0.770 vs. 0.660), semantic equivalence (0.800 vs. 0.680), and component F1 (0.729 vs. 0.709). Mistral Large is the least successful of the four models at generating usable SQL statements (generation success rate of 0.820),

Perhaps the most striking result is the performance of the locally hosted, 4-bit quantized Qwen2.5-Coder-14B model. Despite being orders of magnitude smaller than the cloud-hosted alternatives, it achieves an execution accuracy (0.700), row-count match rate (0.800), and semantic equivalence rate (0.740) that all exceed Mistral Large's corresponding scores (0.660, 0.760, and 0.680, respectively), while matching MiniMax M3's generation success rate (0.950). This indicates that a compact, task-specialized coding model can approach and on several metrics exceed the performance of much larger general-purpose models, which is a meaningful finding for the cost, latency, and privacy trade-offs involved in deploying such systems for this sales-report SQL domain.

C. Contribution of Validation and Self-Correction

To assess the contribution of the validation and self-correction loop, the five evaluation metrics were computed for both initially generated SQL and final SQL after validation and correction attempts. The results show that the validation and correction loop provides greater benefits to smaller or weaker models than to larger and stronger models. Mistral Large showed the largest improvement, with execution accuracy increasing from 0.49 to 0.66 and semantic equivalence from 0.52 to 0.68, while Qwen2.5-Coder-14B also showed consistent gains. MiniMax M3 exhibited smaller improvements, whereas Qwen3.8-Max showed slight decreases in execution accuracy and component F1 despite reaching 100% generation success. Overall, the findings indicate that the proposed correction loop acts primarily as a performance-enhancing mechanism for weaker models, helping them approach the reliability of stronger models which can be seen in Table V.

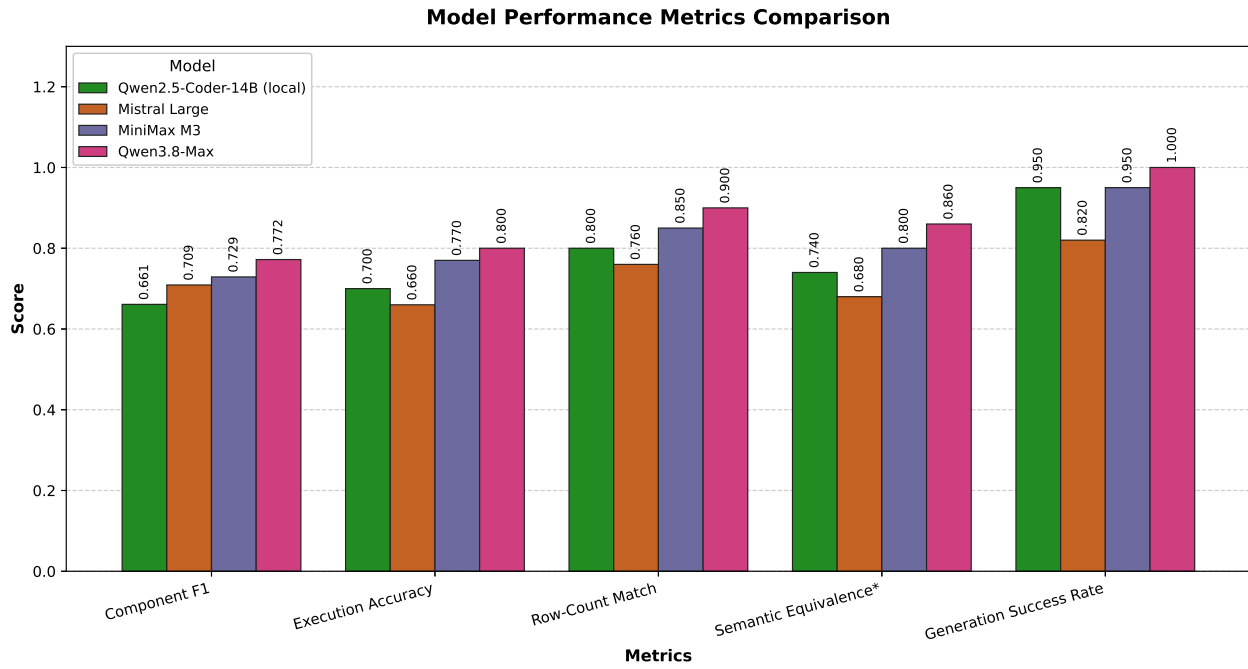


Fig. 9. Overall model performance comparison across evaluation metrics.

TABLE V
INITIAL (BEFORE VALIDATION) VS. FINAL (AFTER VALIDATION & CORRECTION) SQL METRICS BY MODEL

Model	Validation Status	Component F1	Execution Accuracy	Row-Count Match	Semantic Equivalence	Generation Success Rate
Qwen2.5-Coder-14B	Before	65.8%	63.0%	71.0%	67.0%	89.0%
	After	66.1%	70.0%	80.0%	74.0%	95.0%
Mistral Large	Before	66.0%	49.0%	57.0%	52.0%	70.0%
	After	70.9%	66.0%	76.0%	68.0%	82.0%
MiniMax M3	Before	72.2%	76.0%	83.0%	79.0%	91.0%
	After	72.9%	77.0%	85.0%	80.0%	95.0%
Qwen3.8-Max	Before	79.2%	84.0%	89.0%	86.0%	98.0%
	After	77.2%	80.0%	90.0%	86.0%	100.0%

D. Schema Retrieval Performance

The quality of the retrieval of the correct table schema(s) from the schema store was evaluated separately from the downstream generation models because SQL generation relies on the success of this step. The evaluation module was used to test the retrieval module with 100 evaluation queries and determine whether the appropriate schema exists among the retrieved candidates.

TABLE VI
SCHEMA RETRIEVAL PERFORMANCE.

Metric	Value
Hit@5	93.0%
Recall@5	96.67%
Mean Reciprocal Rank (MRR)	98.08%

A Hit rate of 0.93 means that the right schema was found among the top-5 retrieved schemas for 93% of the queries, and

a Recall rate of 0.9667 means that on average, almost all of the required schema elements were in the top-5 retrieved schema. The high MRR (0.9808) also shows that, if the right schema was retrieved, it was typically the top or very top candidate in the list of retrieved schemas, and not the lowest.

E. SQL Component-Level Analysis

Component matching is combined into a single F1 score, but the agent's performance in generating correct SQL varies substantially depending on which part of the query is being generated. Fig. 10 breaks down component F1 by SQL clause type such as Table, Column, WHERE, JOIN, GROUP BY, and Aggregation. ORDER BY is excluded from both the component-level analysis and the overall component score. This is because different models may use different columns, aliases, or expressions to represent the same ordering intent. In addition, some gold SQL queries contain ORDER BY clauses mainly for reporting purposes, even when the natural-language question does not explicitly require ordering. Including OR-

DER BY in the score could therefore penalize models for differences in SQL style rather than actual errors. However, ORDER BY is considered when the natural-language query explicitly requires a specific ordering or ranking.

Two consistent patterns emerge across all four models:

- Table identification is by far the easiest sub-task. All models score at or above 0.85 on table F1 (ranging from 0.85 for Mistral Large to 0.90 for Qwen2.5-Coder-14B and Qwen3.8-Max), indicating that the schema-retrieval/context step reliably surfaces the correct table(s) regardless of which generation model is used downstream.
- GROUP BY generation is the weakest sub-task for every model, ranging from 0.51 (Qwen2.5-Coder-14B) to 0.69 (Qwen3.8-Max). Unlike table identification, the second-weakest component is not consistent across models: it is Column F1 for Qwen2.5-Coder-14B and MiniMax M3, WHERE/JOIN F1 (tied) for Mistral Large, and Aggregation F1 for Qwen3.8-Max. This indicates that the primary bottleneck in this pipeline is not locating the right table, but correctly identifying which columns to select and how to group and aggregate results, i.e., schema-linking and query-planning are more challenging than table-level retrieval, and the specific weak point beyond GROUP BY depends on the generation model itself.

It is notable that Qwen2.5-Coder-14B's column F1 (0.59) is the lowest column-level score among the four models, tied with its own aggregation F1 (0.59) as its second-weakest component. This suggests that the local model reliably narrows down the correct table but struggles more than the cloud-hosted models to select the precise column set.

F. Deployment Considerations: Latency and Cost

Since the motivation for comparing a locally hosted model against cloud-hosted alternatives rests on cost, latency, privacy, and infrastructure control, Table VII reports the average per-query response latency and estimated per-query API cost for each model.

TABLE VII
PER-QUERY LATENCY AND COST BY EACH MODEL.

Model	Deployment	Avg. Latency per Query (s)	Avg. Cost per Query (USD)
Qwen2.5-Coder-14B	Local (Ollama)	2.671	\$0.00110
MiniMax M3	Cloud API	4.709	\$0.00144
Qwen3.8-Max	Cloud API	6.153	\$0.00947
Mistral Large	Cloud API	8.497	\$0.00333

The locally hosted Qwen2.5-Coder-14B achieved the lowest latency with 2.671 sec and cost with \$0.00110/query. Among cloud models, MiniMax M3 showed moderate latency with 4.709 sec and low cost with \$0.00144/query, while Mistral Large had the highest latency with 8.497 sec. Qwen3.8-Max achieved the best accuracy but incurred the highest cost with \$0.00947/query. Overall, the results indicate that the local

model provides a faster and substantially cheaper alternative, with a moderate accuracy trade-off compared with the best-performing cloud model.

Latency denotes the wall-clock response time for SQL generation only (model response time), not including retrieval, validation or database execution. For the three cloud-hosted models, this latency also incorporates network round-trip time, API request queuing and provider-side processing overhead, since generation was performed via API calls rather than on local hardware. Cost figures for models are estimated from provider token pricing at the time of evaluation.

G. Error Analysis and Failure Cases

The generation logs of each query in Qwen2.5-Coder-14B were manually analyzed in representative samples for all 100 queries to uncover the sources of error in the aggregate metrics in Sections III-B–III-F. While evaluating and testing the agent, several common types of errors were found, such as row-limit mismatches, column mismatches, date-filter mismatches, table mismatches, etc. and these errors were added to the validation stage of the agent pipeline, so that the LLM can regenerate and self-correct its output when it detects these errors in the validation stage, but before it is executed. The error analysis methodology is targeted specifically to Qwen2.5-Coder-14B, which is the smallest and least parameterized of the four models evaluated, and is thus expected to make the highest number of mistakes, making it the most useful model to showcase failure modes. This inspection highlighted four non-mutually exclusive areas of error which are summarised in Table VIII.

TABLE VIII
ERROR CATEGORY FREQUENCY FOR QWEN2.5-CODER-14B.

Error Category	Count	Metric(s) Affected
JOIN structure mismatch (extra/missing/substituted JOINS)	24	JOIN F1, Component F1
Missing ORDER BY clause	72	Excluded from scoring; usability only
Generation abstention ("I don't know")	3	Generation Success Rate
Oracle execution error (invalid SQL)	2	Generation Success Rate, Execution Accuracy

JOIN mismatch. For "Show the total invoice revenue for each item category," the gold query requires a two-hop join — SA_SALES_INVOICE → IP_ITEM_MASTER_SETUP → IP_CATEGORY_CODE — since category information is not directly attached to invoice rows. The model rather joined the IP_CATEGORY_CODE onto the SA_SALES_INVOICE, and importantly used an invalid join predicate, `c.CATEGORY_CODE = i.ITEM_CODE`: assigning a category code to an item code. This avoids the need to create the intermediate table and generates a semantically useless join, returning null or garbage values even if it's a syntactically correct SQL query. This demonstrates how JOIN

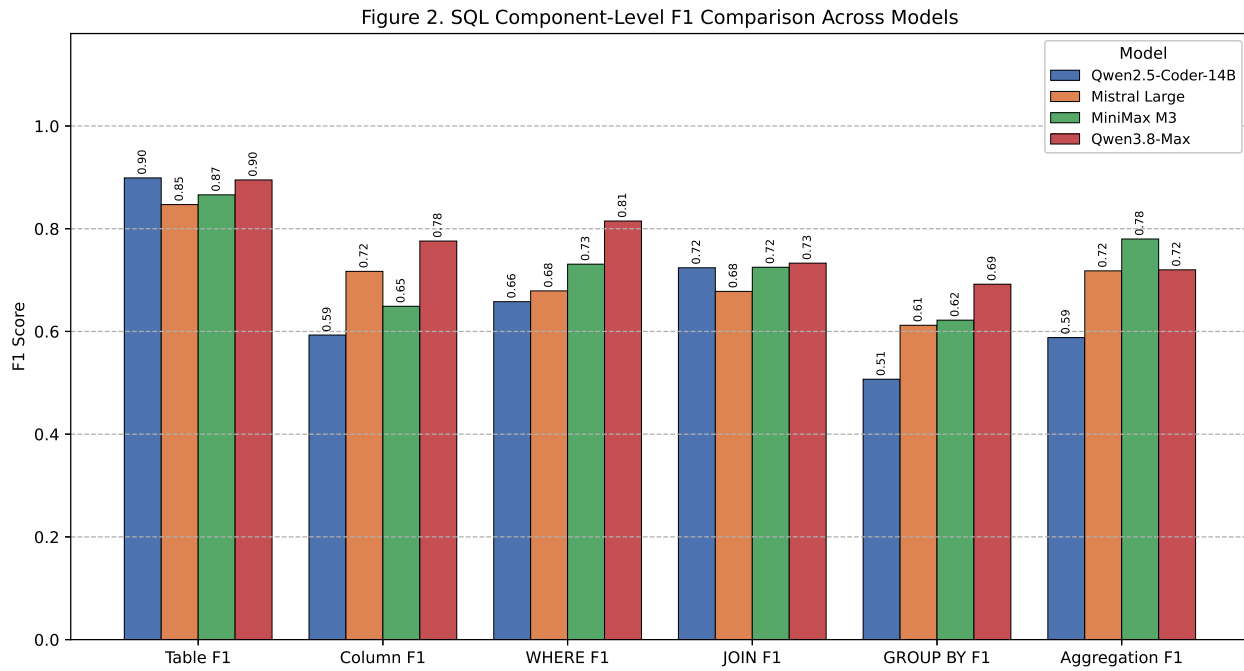


Fig. 10. SQL component-level F1 comparison across models.

errors in this pipeline may not be just extra/missing joins, but erroneous join keys caused by multiple hop schema-linking failures.

Missing ORDER BY. The model correctly copied the join, date filter, and GROUP BY in the “Show the total sales order value for each item in 2025, with the item description” gold query, but not the gold query’s ORDER BY TOTAL_SALES DESC. This is not a change in any of the reported metrics but does indicate the consistent trend to not order output based on the filter/join logic.

Generation abstention. The pipeline log for “Rank items by their total invoice revenue and show only the top 3 for each item category” shows the model initially produced a structurally reasonable solution using RANK() OVER (PARTITION BY CATEGORY_EDDESC ORDER BY SUM(TOTAL_PRICE) DESC) wrapped in a subquery, filtering on the rank alias in the outer WHERE — the correct overall pattern for a top-*N*-per-group query (again using TOTAL_PRICE rather than CALC_TOTAL_PRICE). But, the computed column alias CATEGORY_RANK was marked as “not found in the schema”, which is a false positive as it is a derived column and not a true column; it is not a SQL error. On the third attempt, after two automated self-correction attempts failed to address this validator complaint, the model failed to provide “I don’t know” as the answer. A schema validation step in the pipeline was too strict, rather than a failure of the model to create the correct SQL logic, is responsible for this case of abstention.

Oracle execution error. The model’s query “Show the average number of days between consecutive sales orders for each customer,” had LAG(ORDER_DATE) OVER (PARTITION

BY CUSTOMER_CODE ORDER BY ORDER_DATE) inside the AVG(...) at the same query level. This passed the pipeline’s internal SQL validator (schema/column-level check) but failed at actual Oracle execution with ORA-30483: window functions are not allowed here, since Oracle requires the window function to be resolved in a sub-query before aggregation. This example illustrates a difference between the pipeline’s validator, which checks column and table validity, and genuine Oracle syntax legality, which is only verified at final execution.

H. Summary of Findings

Across the four evaluated models, the following findings stand out:

- On all five metrics used to measure performance: component F1, execution accuracy, row-count match, semantic equivalence and generation success rate. Qwen3.8-Max outperforms the other models tested, but it is also the largest model (2.4T parameters) and is a cloud-only dependency.
- Schema retrieval is very accurate and is not a major constraint for the agent. The retrieval module has zero failed retrievals, and a Hit@5 of 0.93, Recall@5 of 0.9667 and MRR of 0.9808, as seen in the table F1 scores which are uniformly high, across all four generation models regardless of scale. As a result, the performance of the agent is mostly determined by the generation stage and not the retrieval stage.
- The ablation comparing initial and final (post-validation) SQL further shows that the deterministic validation and correction loop consistently improves evaluation metrics

across all models and yields substantial accuracy gains for weaker generators such as Qwen2.5-Coder-14B and Mistral Large, but produces only marginal gains for the already-strong like Minimax M3 and Qwen3.8-Max, indicating that the correction loop functions primarily as a reliability safety net rather than a uniform accuracy booster.

- Model scale is not completely predictive of performance in this domain. The locally hosted Qwen2.5-Coder-14B is competitive with, and at times superior to Mistral Large on execution accuracy and generation success rate, and MiniMax M3 is consistently better than Mistral Large despite having only about a third of its parameters. Considering accuracy, latency (2.67 s per query, the fastest of the four models) and cost (\$0.0011 per query) together, Qwen2.5-Coder-14B represents a well-rounded and practical choice for enterprise deployment, indicating that for a scoped, domain-specific SQL-generation task with deployment cost and speed.

IV. CONCLUSION

The aim of this study was to explore the feasibility of replacing much larger, cloud-hosted language models with a locally hosted one for enterprise Text-to-SQL generation, in the context of a shared retrieval-augmented and rule-validated pipeline. The suggested system includes schema retrieval, schema and SQL example retrieval, LLM-generated SQL, deterministic validation, bounded self-correction, and Oracle execution. The study was conducted to address only the impact of the various small and large language model for SQL-generation.

The results reveal that Qwen3.8-Max outperformed the other models in all five metrics: semantic equivalence, execution accuracy, component F1, row-count matching, and generation success rate. But the locally hosted 4-bit quantized Qwen2.5-Coder-14B was still competitive with significantly fewer parameters, performing comparable to other large models. Schema retrieval was also demonstrated to be not a main bottleneck with Hit@5 of 0.93, Recall@5 of 0.9667, and MRR of 0.9808. In both cases, the most difficult SQL components were GROUP BY generation, and selection by column, with most of the limitations being focused in the construction of detailed queries, not on table retrieval. An ablation comparing SQL before validation and final SQL after validation further showed that the deterministic validation and self-correction loop is not uniformly beneficial: it produced substantial accuracy gains for weaker models than large and strong model, indicating that its main value lies in reliability rather than raw accuracy improvement.

These results are useful for enterprise deployments where data privacy, infrastructure control, and dependency on external APIs are key considerations. They show that model size alone does not determine whether a text-to-SQL model is suitable for a bounded enterprise workload. A smaller, locally deployed model, combined with effective retrieval, structured prompts, and deterministic validation with bounded correction,

can offer a practical balance between SQL generation capability and deployment considerations such as cost, latency, and infrastructural control. At the same time, Qwen3.8-Max's consistent performance advantage highlights that a larger cloud-hosted model retains real benefits when generation quality is the overriding priority and cost or latency constraints are less critical.

There are a number of limitations of this study that should be taken into account when interpreting these results. First, this evaluation was performed using 100 held-out queries which may not be enough to generalize the models performance. Second, only four generation models were compared, and the study did not perform a thorough ablation study of each individual factor of retrieval and prompting. Third, the reported latency and cost figures for cloud-hosted models include network and provider-side overhead in addition to model inference time and were not accompanied by hardware specifications, memory consumption, or tokens-per-second throughput for the locally hosted model, limiting the precision of the deployment-cost comparison.

Future work should be directed towards the validation layer, which could also be improved to be able to better identify valid derived expressions from actual schema errors and do more in-depth Oracle syntax checks. A second promising path is to include Oracle execution errors into the self-correcting loop, enabling further SQL correction and re-execution within a bounded retry process, based on Oracle error messages. Lastly, a cross evaluation of the architecture on a wider set of enterprise schemas and direction (HR, Employee, Customer, etc), SQL dialects, and local and cloud models would give increased insights into its broad applicability.

REFERENCES

- [1] A. Mohammadjafari, A. S. Maida, and R. Gottumukkala, "From natural language to sql: Review of llm-based text-to-sql systems," 2025.
- [2] Z. Hong, Z. Yuan, Q. Zhang, H. Chen, J. Dong, F. Huang, and X. Huang, "Next-generation database interfaces: A survey of llm-based text-to-sql," 2025.
- [3] G. Katsogiannis-Meimarakis and G. Koutrika, "A survey on deep learning approaches for text-to-sql," *The VLDB Journal*, vol. 32, no. 4, p. 905–936, Jan. 2023.
- [4] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "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.-Nov. 2018, pp. 3911–3921.
- [5] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K. C. C. Chang, F. Huang, R. Cheng, and Y. Li, "Can LLM already serve as a database interface? a big bench for large-scale database grounded text-to-SQLs," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.
- [6] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating structured queries from natural language using reinforcement learning," *arXiv preprint arXiv:1709.00103*, 2017.
- [7] N. Rajkumar, R. Li, and D. Bahdanau, "Evaluating the text-to-SQL capabilities of large language models," *arXiv preprint arXiv:2204.00498*, 2022.
- [8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler,

- J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.
- [9] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.
- [10] Y. Luo, G. Li, J. Fan, C. Chai, and N. Tang, "Natural language to sql: State of the art and open problems," *Proc. VLDB Endow.*, vol. 18, no. 12, p. 5466–5471, Aug. 2025.
- [11] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, "RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers," in *Proc. 58th Annu. Meeting of the Association for Computational Linguistics (ACL)*, 2020, pp. 7567–7578.
- [12] P. Yin, G. Neubig, W. Yih, and S. Riedel, "TaBERT: Pretraining for joint understanding of textual and tabular data," in *Proc. 58th Annu. Meeting of the Association for Computational Linguistics (ACL)*, 2020, pp. 8413–8426.
- [13] T. Scholak, N. Schucher, and D. Bahdanau, "PICARD: Parsing incrementally for constrained auto-regressive decoding from language models," in *Proc. 2021 Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2021, pp. 9895–9901.
- [14] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.
- [15] T. Mahmud, K. A. Hasan, M. Ahmed, and T. H. C. Chak, "A rule based approach for NLP based query processing," in *Proc. 2nd Int. Conf. Electrical Information and Communication Technologies (EICT)*, 2015, pp. 78–82.
- [16] X. Xu, C. Liu, and D. Song, "SQLNet: Generating structured queries from natural language without reinforcement learning," *arXiv preprint arXiv:1711.04436*, 2017.
- [17] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J.-G. Lou, T. Liu, and D. Zhang, "Towards complex text-to-SQL in cross-domain database with intermediate representation," in *Proc. 57th Annu. Meeting of the Association for Computational Linguistics (ACL)*, 2019, pp. 4524–4535.
- [18] R. Cao, L. Chen, Z. Chen, S. Zhu, and K. Yu, "LGESQL: Line graph enhanced text-to-SQL model with mixed local and non-local relations," in *Proc. 59th Annu. Meeting of the Association for Computational Linguistics (ACL)*, 2021, pp. 2541–2555.
- [19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. 2019 Conf. North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, 2019, pp. 4171–4186.
- [20] H. Li, J. Zhang, C. Li, and H. Chen, "RESDSL: Decoupling schema linking and skeleton parsing for text-to-SQL," in *Proc. AAAI Conf. Artificial Intelligence*, vol. 37, no. 11, 2023, pp. 13 067–13 075.
- [21] M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.
- [22] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou, "Text-to-SQL empowered by large language models: A benchmark evaluation," *Proc. VLDB Endowment*, vol. 17, no. 5, pp. 1132–1145, 2024.
- [23] X. Dong, C. Zhang, Y. Ge, Y. Mao, Y. Gao, L. Chen, J. Lin, and D. Lou, "C3: Zero-shot text-to-SQL with ChatGPT," *arXiv preprint arXiv:2307.07306*, 2023.
- [24] B. Wang, C. Ren, J. Yang, X. Liang, J. Bai, L. Chai, Z. Yan, Q.-W. Zhang, D. Yin, X. Sun, and Z. Sui, "MAC-SQL: A multi-agent collaborative framework for text-to-SQL," in *Proc. 31st Int. Conf. Computational Linguistics (COLING)*, 2025, pp. 540–557.
- [25] S. Taleai, M. Pourreza, Y.-C. Chang, A. Mirhoseini, and A. Saberi, "CHESS: Contextual harnessing for efficient SQL synthesis," *arXiv preprint arXiv:2405.16755*, 2024.
- [26] H. Xia, R. Jiang, J. Deng, Y. Wang, C. Wang, Y. Chen, H. Chen, W. Zhang, and J. Wang, "SQL-CRAFT: Text-to-SQL through interactive refinement and enhanced reasoning," *arXiv preprint arXiv:2402.14851*, 2024.
- [27] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459–9474.
- [28] H. Li, J. Zhang, H. Liu, J. Fan, X. Zhang, J. Zhu, R. Wei, H. Pan, C. Li, and H. Chen, "CodeS: Towards building open-source language models for text-to-SQL," *Proc. ACM on Management of Data*, vol. 2, no. 3, pp. 1–28, 2024.
- [29] M. Pourreza and D. Rafiei, "DTS-SQL: Decomposed text-to-SQL with small large language models," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024, pp. 8212–8220.
- [30] R. Zhong, T. Yu, and D. Klein, "Semantic evaluation for text-to-SQL with distilled test suites," in *Proc. 2020 Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 396–411.
- [31] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, K. Dang, Y. Fan, Y. Zhang, A. Yang, R. Men, F. Huang, B. Zheng, Y. Miao, S. Quan, Y. Feng, X. Ren, X. Ren, J. Zhou, and J. Lin, "Qwen2.5-Coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.
- [32] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhume, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark, "Self-refine: Iterative refinement with self-feedback," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.
- [33] C. Wang, K. Tatwawadi, M. Brockschmidt, P.-S. Huang, Y. Mao, O. Polozov, and R. Singh, "Robust text-to-SQL generation with execution-guided decoding," *arXiv preprint arXiv:1807.03100*, 2018.
- [34] T. Yu, R. Zhang, M. Yasunaga, Y. C. Tan, X. V. Lin, S. Li, H. Er, I. Li, B. Pang, T. Chen, E. Ji, S. Dixit, D. Proctor, S. Shim, J. Kraft, V. Zhang, C. Xiong, R. Socher, and D. Radev, "SPaRc: Cross-domain semantic parsing in context," in *Proc. 57th Annu. Meeting of the Association for Computational Linguistics (ACL)*, 2019, pp. 4511–4523.
- [35] F. Bang, "GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings," in *Proc. 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, 2023, pp. 212–218.
- [36] J. Chen, S. Xiao, P. Zhang, K. Luo, D. Lian, and Z. Liu, "M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation," in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 2318–2335.
- [37] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging LLM-as-a-judge with MT-Bench and chatbot arena," in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 46 595–46 623.