DLBENCH: A Comprehensive Benchmark for SQL Translation with Large Language Models

Li Lin
School of Informatics, Xiamen University
Xiamen, China
linli1210@stu.xmu.edu.cn

Hongqiao Chen
School of Informatics, Xiamen University
Xiamen, China
23020241154371@stu.xmu.edu.cn

Qinglin Zhu
School of Informatics, Xiamen University
Xiamen, China
23020241154481@stu.xmu.edu.cn

Liehang Chen
School of Informatics, Xiamen University
Xiamen, China
2320415112@qq.com

Linlong Tang
School of Informatics, Xiamen University
Xiamen, China
Tangll114514@gmail.com

Rongxin Wu
School of Informatics, Xiamen University
Xiamen, China
wurongxin@xmu.edu.cn

***Abstract*—In recent years, the growing complexity of database management systems (DBMSs) and the proliferation of SQL dialects have created significant challenges for database migration, federation, and integration. These challenges arise from the disparities between SQL dialects across different DBMSs, hindering seamless communication and system interoperability. SQL translation, the process of converting SQL queries from a source dialect DBMS to a target dialect DBMS, plays a crucial role in addressing these challenges. To facilitate this process, we introduce DLBENCH, the first comprehensive benchmark designed to evaluate the SQL translation capabilities of Large Language Models (LLMs). The benchmark includes two datasets: BIRDTRANS, which covers real-world database query scenarios across seven DBMSs, and BUTTERTRANS, which spans a broader spectrum of SQL types and encompasses extensive DBMS dialect features. We collect high-quality databases and SQL statements, applying a rigorous multi-step cleaning process that ensures data quality through SQL92-based checks and dialect-specific parser validation. Additionally, both LLM-based and human annotations are used to guarantee the correctness and completeness of the dataset. We demonstrate the utility of DLBENCH through extensive experiments, which show that the benchmark effectively evaluates the SQL translation ability of LLMs. The results highlight the potential of LLMs for SQL translation tasks and provide insights into areas for further improvement.**

***Index Terms*—SQL translation, SQL dialect, Benchmark**

I. INTRODUCTION

Over the past several decades, the realm of database management has grown markedly more intricate. This increased complexity stems not only from the expanding array of specialized Database Management Systems (DBMSs) designed for diverse workloads but also from the proliferation of mutually incompatible SQL dialects. Such disparities hinder migration and federation efforts, effectively binding users to their initial DBMS regardless of potential shortcomings in cost, performance, or specific functionalities. Facilitating SQL translation between these diverse dialects would not only simplify migrations but also enable the development of sophisticated federated systems that intelligently route workload segments to the most efficient or cost-effective DBMS [1]–[5].

SQL translation is a difficult task due to the inherent complexity and subtle differences in syntax and semantics among various SQL dialects. For example, as shown in Figure 1, translating a `CREATE TRIGGER` statement between different DBMSs requires considering the distinct implementations across systems: MariaDB uses `if-else` statements for conditionals, SQLite employs the `WHEN` clause together with the `UPDATE` statement to achieve similar functionality, and PostgreSQL necessitates the declaration of an additional function to create a trigger. Although `CREATE TRIGGER` is a common feature in most relational DBMSs, there remain substantial differences in the grammar across different systems. A conventional solution involves translating queries across DBMSs using rule-based tools such as the SQLGLOT parser. These rule-based tools [3], [6] largely automate the translation process; however, they do not always perform well, leaving developers to manually convert segments of code that the tools fail to handle. For instance, during the translation of the BIRD benchmark [7] from SQLite to BigQuery, approximately 80% of the queries encountered errors using the SQLGLOT parser.

In recent years, Large Language Models (LLMs) have demonstrated potential in code generation and code translation. Existing studies [1], [2], [8] show that models such as GPT-4 [9] and Bard [10] possess impressive code-writing capabilities, positioning them as promising candidates for tackling SQL translation challenges. However, current evaluation practices for LLM-based SQL translation remain overly simplistic. For example, MALLET [2] and SEDAR [8] assess success based merely on whether the translated SQL statement from a known benchmark (e.g., *TPC-DS* [11]—a decision-support performance benchmark) executes without error in the target DBMS. More recently, CRACKSQL [12] takes a step forward by explicitly considering the consistency of query results before and after translation. Nonetheless, execution-driven evaluation still has two key limitations. First, it overlooks both semantic equivalence and dependency integrity. Two queries may produce slightly different outputs across DBMSs (e.g., differences in date–time formatting between MySQL and

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00076

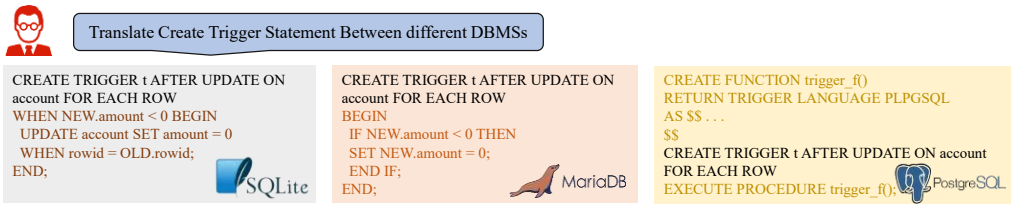

Fig. 1: Translate the CREATE TRIGGER statement in different DBMSs.

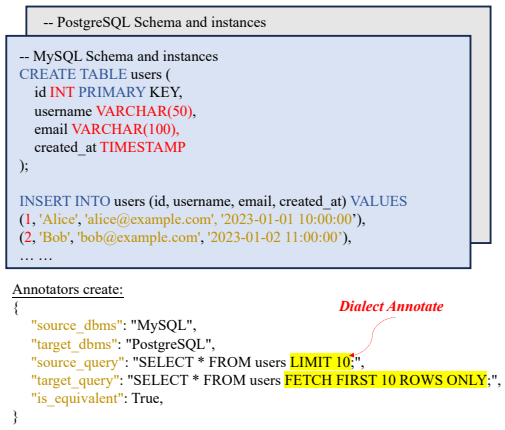

Fig. 2: The annotations for SQL translation task.

PostgreSQL) yet convey the same intent, or a translated query may yield the right answer on a test dataset while silently breaking schema constraints, such as incorrectly translating MySQL's "AUTO_INCREMENT" into a plain integer without PostgreSQL's "SERIAL", or omitting integrity rules like "FOREIGN KEY" references and "UNIQUE" checks. In both cases, execution-based evaluation such as CRACKSQL [12] would fail to capture these semantic issues. Second, it overlooks the breadth and variability of dialect-specific constructs encountered in real-world SQL, leading to an incomplete assessment of model capability.

To better reflect realistic SQL translation scenarios encountered in practical development, we propose a new task formulation for the SQL translation problem in Section II. Specifically, to overcome the first limitation, our formulation introduces two additional constraints: semantic equivalence and dependency integrity. These constraints ensure that the translated query not only produces semantically equivalent results as the source query but also preserves the logical relationships among database objects, such as foreign key dependencies.

To support this task and measure the SQL translation capabilities of LLMs, we present DLBENCH, a comprehensive SQL translation benchmark comprising 6,402 translation tasks, supporting seven different DBMSs, and covering 9,320 SQL dialects, as summarized in Table II. By spanning a broad range of DBMSs and dialect-specific features, DLBENCH directly addresses the second limitation, ensuring that evaluations reflect not only execution feasibility but also the diversity and variability of real-world SQL constructs. DLBENCH is the first innovative benchmark for SQL translation that reflects real-world scenarios by translating statements from one dialect DBMS to another. To make DLBENCH more suitable for evaluation and training, we have meticulously annotated each SQL translation task. As shown in Figure 2, given a database, our corpus creates and annotates complex SQL translation tasks, involving the translation of source queries in one SQL dialect to target queries in another SQL dialect. For each task, we annotate the position of the dialects and include external knowledge, which helps the model understand dialect-specific keywords and ensures accurate translation. Additionally, we provide an equivalence label to guarantee that both the syntax and semantics are preserved across dialects, with more detailed information provided in Section II-B. Our annotations also include relevant schema details and runtime environment specifications, outlining the logical structure of database objects, such as tables, columns, indexes, functions, and constraints. This comprehensive information ensures that the translated SQL queries can be executed correctly in the target DBMS.

DLBENCH includes two datasets to evaluate the translation capability of LLMs in different scenarios: the BIRDTRANS dataset, which covers popular DBMSs and real-world database query scenarios, and the BUTTERTRANS dataset, which spans a broader spectrum of SQL types and encompasses extensive DBMS dialect features. We construct our benchmark from two primary sources: BIRDTRANS is generated by augmenting the original BIRD text-to-SQL dataset, whereas BUTTERTRANS is collected from popular DBMS test cases to cover a broader range of dialect features. We begin by collecting high-quality databases and SQL statements. Each statement then undergoes a multi-step cleaning and filtering process, which includes SQL92-based checks to verify the presence of dialect-specific syntax, as well as dialect-specific parser validation to ensure data quality. Finally, both LLM-based and human annotations are applied to guarantee the correctness and completeness of the data.

In addition to benchmark construction, we implement an automated *translation–execution–evaluation* pipeline and introduce a novel evaluation framework that includes both content matching-based and execution-based metrics. We introduce a new Dialect Matching (DM) metric for assessing dialect-specific constructs, and adapt Exact Matching (EM) and Execution Accuracy (EX) from Text-to-SQL evaluation [13]. The content matching–based metrics DM and EM evaluate structural and syntactic alignment of translated SQL with the ground truth—focusing on dialect-specific features and

exact matches—while the execution-based metric EX verifies correctness by comparing execution results of translated and ground-truth statements on a database.

We conducted experimental evaluations of SQL translation using DLBENCH with seven advanced LLMs: four general-purpose models (one open-source and three closed-source) and three code-specific models. The results show that the performance of LLMs in SQL translation is far from satisfactory. The best-performing language model, GPT-4o, only achieved a score of 0.70 on the EX metric. Subsequently, we explored two enhanced prompting strategies—few-shot prompting and knowledge-augmented prompting—which led to modest improvements in performance. However, the model's overall effectiveness still falls short of the requirements for reliable automatic SQL translation.

Our contributions can be summarized as follows:

- We introduce DLBENCH, the first comprehensive benchmark for evaluating the SQL translation capabilities of LLMs, featuring over 6,402 translation tasks across seven DBMSs and 9,320 SQL dialects. This benchmark enables robust and realistic assessment of LLM performance in practical SQL translation scenarios.
- Besides introducing DLBENCH, we also develop a systematic evaluation design including translation-execution-evaluation pipeline and three progressive metrics. This evaluation design provides a reference for future SQL translation assessments.
- We conduct extensive experiments that yield several instructive findings. The results show that LLMs' translation performance still falls well short of expectations.
- Our benchmark repository is publicly available at https://dlbenchll.github.io/, providing a valuable resource for future research on SQL translation tasks and enabling further advancements in LLM-based SQL translation.

II. TASK DEFINITION

In this section, we define a SQL translation task that addresses key limitations of prior work and more accurately reflects real-world requirements. Unlike most existing approaches that consider a translation successful if it merely executes without error on the target DBMS, our task formulation evaluates models on both syntactic correctness and semantic fidelity. This stricter definition ensures that models can only succeed when they genuinely understand the semantic meaning of dialect-specific features, rather than relying on pattern matching or memorization. The SQL translation task aims to convert a SQL statement from a source dialect DBMS $\mathcal{D}_s$ (e.g., MySQL) to a target dialect DBMS $\mathcal{D}_t$ (e.g., PostgreSQL), while preserving both syntactic validity and semantic equivalence. This task requires formal specification of its input-output relationships and equivalence constraints, as detailed below.

A. Input-Output Space

Let $\mathcal{S}$ denote the universal set of valid SQL statements, partitioned into dialect-specific subsets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$ corresponding to k distinct SQL dialects in $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k\}$. The translation process operates within a database context $\mathcal{C} = (\Sigma, \mathcal{I})$, where Σ represents the database schema (tables, columns, data types, and constraints) and $\mathcal{I}$ denotes the concrete dataset instance containing actual data rows.

The translation function $\mathcal{T} : \mathcal{S}_s \times \mathcal{D}_t \times \mathcal{C} \to \mathcal{S}_t$ maps a source statement $\mathcal{S}_s \in \mathcal{S}_i$ from dialect DBMS $\mathcal{D}_s$ to a target statement $\mathcal{S}_t \in \mathcal{S}_j$ in dialect DBMS $\mathcal{D}_t$, conditioned on the provided database context. Here, $\mathcal{S}_i$ and $\mathcal{S}_j$ represent the source and target dialect subspaces respectively.

B. Semantic Equivalence Constraints

A valid translation $\mathcal{S}_t$ of a source statement $\mathcal{S}_s$ under the database context $\mathcal{C} = (\Sigma, \mathcal{I})$ must satisfy three core constraints. First, it must be syntactically valid in the target dialect DBMS, as verified by a dialect-specific parser:

$$\text{Parsable}(\mathcal{S}_t, \mathcal{D}_t) = \top \tag{1}$$

Second, the translated query must produce execution results identical to the source query when applied to the database context $\mathcal{C}$:

$$\text{Execute}(\mathcal{S}_s, \mathcal{C}, \mathcal{D}_s) \equiv \text{Execute}(\mathcal{S}_t, \mathcal{C}, \mathcal{D}_t) \tag{2}$$

where $\equiv$ denotes strict equality of result sets, including preservation of row order and duplicates when explicitly specified by the query. Although strict equivalence is required, practical translators may allow datatype-driven relaxations that preserve semantics while differing in representation. For example, MySQL's `DATETIME` is formatted as `%Y-%m-%d %H:%M:%S` (e.g. `2008-04-30 00:00:00`), whereas PostgreSQL's `TIMESTAMP` uses `%Y-%m-%d %H:%M:%S.%f` (e.g. `2008-04-30 00:00:00.000000`); since both denote the same instant, they are treated as approximately equivalent. We annotate these cases with the **SemanticEquivalent** annotation (see Section III).

Third, the translation must preserve all functional dependencies and integrity constraints implied by the schema Σ:

$$\text{ConstraintsPreserved}(\Sigma, \mathcal{S}_t) = \top, \tag{3}$$

For instance, MySQL's `AUTO_INCREMENT` columns should map to PostgreSQL's `SERIAL` type rather than generic integer sequences. Including Σ in the formalism captures SQL's reliance on metadata—column types determine which functions can be applied, and primary/foreign keys or check constraints guide the optimizer's execution plan. Including the instance $\mathcal{I}$ accounts for data-dependent behaviors (for example, locale-specific string comparisons or NULL handling).

III. DLBENCH

In this section, we introduce the construction of DLBENCH, the benchmark meticulously designed to evaluate the translation capabilities of LLMs in SQL translation. Figure 3 illustrates the entire process involved in building our benchmark, starting from the collection of databases and SQL statements to the final SQL annotation and quality control steps. More details will be introduced in the following subsections.

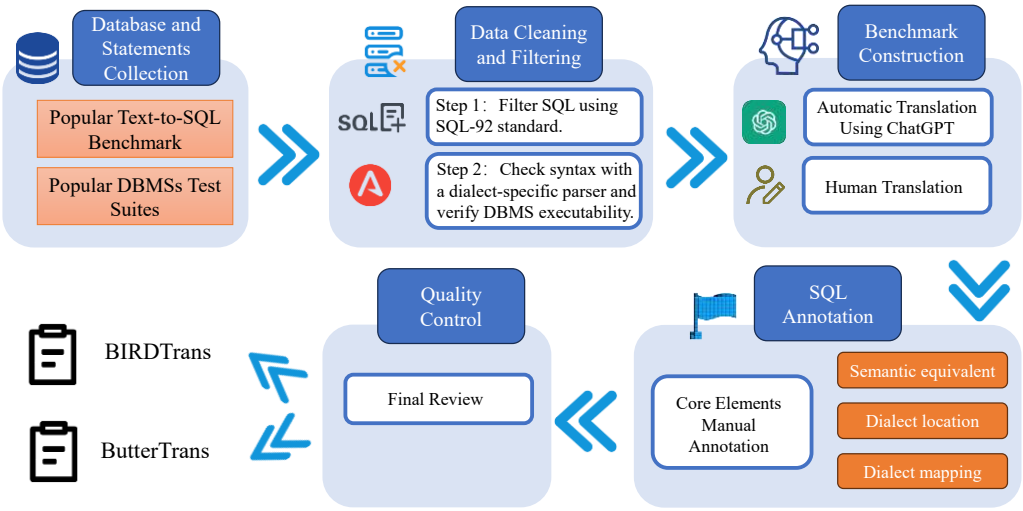

Fig. 3: The annotation process of our DLBENCH benchmark.

A. Benchmark Construction

(1) Database and Statements Collection. The first step in constructing DLBENCH is collecting high-quality databases and SQL statements to prepare for subsequent SQL translation tasks. We leverage two primary sources for this collection: the popular Text-to-SQL benchmark BIRD and widely-used DBMS test suites. BIRD [7] covers a variety of cross-domain query scenarios, including diverse application domains such as e-commerce, finance, and healthcare, and contains a significant number of queries in the SQLite dialects. We also collect popular DBMS test suites, including those for MySQL [14] and PostgreSQL [15], which cover a wide range of SQL dialects and encompass various key SQL features, such as dialect-specific syntax and variations in Data Manipulation Language (DML) statements, Data Control Language (DCL) commands, and Transaction Control Language (TCL) operations. These sources provide a diverse collection of SQL statements, ensuring that DLBENCH covers a broad spectrum of real-world scenarios and dialects.

(2) Data Cleaning and Filtering. Once the SQL statements are collected, they undergo a multi-step data cleaning and filtering process. First, we filter out SQL statements that conform to the SQL-92 standard to ensure that our SQL translation task focuses on dialect-specific translation rather than fully equivalent syntax. Specifically, we use an ANTLR parser based on the ANSI Standard to parse the statements. If the statements fail the parsing process, we retain them for further analysis. Additionally, we check the syntax with a dialect-specific parser and verify DBMS executability. It is worth noting that although SQL-92 provides a common baseline, standard-compliant queries may still behave differently across DBMSs. To assess this risk, we conduct a sampling analysis by selecting 100 SQL-92–compliant queries and testing them across seven representative DBMSs (See Table I). All sampled cases exhibit consistent behavior, and we therefore regard such discrepancies as statistically insignificant. This ensures that we can efficiently filter out queries that are valid for the specific DBMSs and can be executed properly. Finally, after these filtering and verification steps, we obtain a refined set of SQL statements that are ready for translation and annotation.

(3) Benchmark Construction. In this step, we utilize the commercial model GPT-4o-mini to assist in translating SQL queries across DBMSs. We build an automated environment in which GPT-4o-mini receives the source and target dialect identifiers along with the necessary schema definitions, generates the corresponding SQL for the target DBMS, and automatically executes it. Finally, human experts review the execution results to verify that the translated queries are semantically equivalent to the originals. During the translation process, we also record the database context in which each query is executed, ensuring that the context of the translation is preserved. Based on our trials, approximately 35% of the statements were successfully translated using the model. In addition to automatic translation, human experts are involved to manually translate queries, ensuring the high accuracy and reliability of the benchmark. For queries that were not successfully translated, the experts manually translated them. To ensure expert quality, we recruit trained domain specialists with substantial experience in database development and SQL dialects. Before participating in the main construction phase, each expert completes a qualification task consisting of 20 SQL translation items sampled from known dialect transformations. Only experts who achieve at least 90% agreement with the reference answers are included in the final pool. Finally, we select three experts with over three years of database development experience to review all translations.

Our translation process strictly adheres to the constraints outlined in Section II-B. During the translation, we encountered cases where translation was not possible. A major reason for this was the differing support for DBMS-specific features. When the target DBMS does not support features present in the original query, such as unsupported functions, the

SQL translation can result in an executable query failure. For example, MariaDB provides certain geospatial functions that are unavailable in other DBMSs. These issues, though rare, must be accounted for during the translation process. After the combined efforts of ChatGPT and human translators, we successfully collected a total of 6,402 translation tasks, with the entire process requiring 150 man-hours.

(4) SQL Annotation. In this step, three experienced experts manually annotate SQL queries to capture key dialect-specific features. Two primary annotators are selected based on their extensive experience and familiarity with the SQL dialects involved. These annotators are responsible for independently labeling each translation instance, following a detailed guideline. In cases of disagreement, a third expert served as an adjudicator, reviewing both annotations and consulting relevant documentation to determine the final label. This triadic setup ensures both domain expertise and annotation consistency. To evaluate annotation reliability, we measure inter-annotator agreement using Cohen's kappa, which accounts for agreement due to chance. The overall kappa score between the two primary annotators is 0.92, indicating a high level of consistency and confidence in annotation quality. We label each query with three annotations: **semantic equivalent**, **dialect location**, and **dialect knowledge**. The semantic equivalent label includes both approximate and exact equivalence, with approximate equivalence referring to queries that produce similar results, as discussed in Section II-B. For semantic equivalence annotation, we do not rely solely on execution results. Instead, experts analyze the semantics of queries by referencing official SQL documentations and verifying whether two queries express the same intent and yield consistent behavior across DBMSs. Execution results are considered supportive evidence but are not the sole basis for annotation. As illustrated in Section II-B, even when execution outputs differ in representation across DBMSs (e.g., differences in date-time formatting between MySQL and PostgreSQL), such cases are annotated as *approximately equivalent* since they preserve the same underlying semantics. This explicit semantic annotation ensures that equivalence is defined at the logical level rather than being tied to a specific DBMS execution, distinguishing our benchmark from existing ones [2], [12] that judge equivalence primarily by execution behavior. The dialect location annotation identifies specific dialect features, such as unique keywords "`LIMIT 10`" in Figure 2, within the query. In addition, we annotate each dialect feature with external knowledge extracted from official SQL documentation to give the model a comprehensive understanding of dialect-specific behavior and functionality. For any gaps in the documentation, we leverage GPT-4o-mini to generate the missing information. For example, we annotate MySQL's dialect feature "`NULLIF`" as follows:

> **Feature:** NULLIF(expr1, expr2)
> **Explanation:** The NULLIF function returns NULL if expr1 equals expr2. Otherwise, it returns the value of expr1. This function is equivalent to the following CASE expression: CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END. The return value of NULLIF has the same data type as the first argument (expr1).
> **Example:** `mysql> SELECT NULLIF(1, 1);`

These annotations ensure that the model fully understands the context and behavior of each dialect feature, facilitating more accurate translations between SQL dialects. The annotation process took 150 man-hours to complete.

(5) Quality Control. Finally, we rigorously review each translation task to ensure compliance with predefined quality standards. Three domain experts conduct comprehensive validation, verifying syntactic and semantic correctness through dialect-specific parsers, ensuring result consistency between source and translated SQL queries across DBMSs under identical datasets, and confirming accurate handling of dialect-specific features such as function replacements or syntax adaptations. Identified discrepancies—including unsupported functions, logic deviations, or syntax mismatches—are systematically resolved through re-translation or manual adjustments, thereby ensuring the reliability and practical utility of the benchmark.

B. Dataset Statistics

Selected DBMSs. DLBENCH comprises seven distinct relational DBMS, as summarized in Table I. Our source SQL statements are drawn from the SQLite dialects of the BIRD benchmark [7] and the official MySQL [14] and PostgreSQL [15] test suites. For translation targets, we choose six widely used, large-scale open-source relational DBMSs: MySQL, PostgreSQL, MariaDB, MonetDB, DuckDB, and ClickHouse—as our translation targets. These systems were selected because they (1) enjoy broad community adoption, (2) adhere closely to standard SQL syntax while exhibiting realistic dialectal variations, and (3) provide comprehensive, publicly available SQL documentation.

TABLE I: Key characteristics of selected DBMSs

DBMS	GitHub Stars	DB-Engine	First Release	Role
MySQL	8.6K	2	1995	Both
PostgreSQL	16.1K	4	1989	Both
SQLite	6.5K	10	2000	Source
MariaDB	4.6K	15	2009	Target
MonetDB	375	135	2004	Target
DuckDB	23.6K	57	2018	Target
ClickHouse	37.2K	37	2016	Target

Statistics and Dialect Diversity. Table II presents the statistics of two datasets included in DLBENCH: BIRDTRANS and BUTTERTRANS. BIRDTRANS covers 3,206 translation tasks across 4,669 dialect variants, while BUTTERTRANS includes 3,196 tasks spanning over 4,651 dialect features. Each dataset supports translation to six popular DBMSs, with varying average token lengths and statement lengths. We also followed the SQL standardization ISO/IEC 9075 [16] to classify statements in BUTTERTRANS dataset. As shown in Figure 4, DQL dominates, followed by DDL and DML. Other

TABLE II: Dataset Specifications

Dataset	Source DBMS	Target DBMS	Task Numbers	Dialect Numbers	Avg. Tokens	Avg. Length
BIRDTRANS	Text-to-SQL Dataset BIRD (SQLite)	MySQL, PostgreSQL, MariaDB, MonetDB, DuckDB, ClickHouse	3,206	4,669	81.17	236.97
BUTTERTRANS	Popular DBMS Test Suites (MySQL and PostgreSQL)	MySQL, PostgreSQL, MariaDB, MonetDB, DuckDB, ClickHouse	3,196	4,651	28.38	66.22

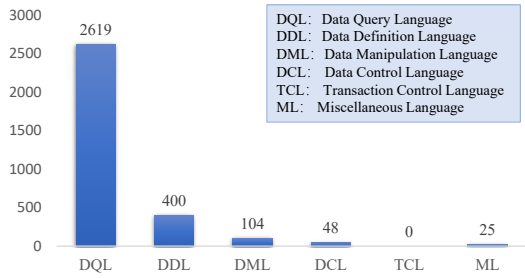

Fig. 4: Distribution of SQL statement types across individual translation tasks in BUTTERTRANS dataset.

categories such as DCL, TCL, and miscellaneous types appear less frequently but remain essential for comprehensive dialect coverage. This distribution reflects the syntactic and semantic diversity of DLBENCH, presenting substantial challenges for LLM-based SQL translation.

IV. EVALUATION METRICS

This section presents metrics for evaluating SQL translation quality, covering both content matching-based and execution-based criteria. We propose a new Dialect Matching (DM) metric to measure translation of dialect-specific constructs, and adapt the Exact Matching (EM) and Execution Accuracy (EX) metrics as originally defined for Text-to-SQL evaluation [13]. We will release the official evaluation script along with our corpus so that the research community can share the same evaluation platform.

A. Content Matching-based Metrics

Dialect Matching (DM) evaluates the performance of the SQL translation system by measuring how many dialect-specific features are successfully translated from the source dialect to the target dialect. Let

$$\mathcal{G} = \bigcup_{i=1}^{k} \Big(\mathcal{S}_i \setminus \bigcup_{j \neq i} \mathcal{S}_j \Big) \tag{4}$$

be the set of all constructs that occur in exactly one dialect subset $\mathcal{S}_i$. For any SQL statement s, define its dialect feature set as

$$\Phi(s) = \{\, g \in \mathcal{G} \mid g \text{ appears in } s \}. \tag{5}$$

For each SQL translation task $n = 1, \ldots, N$, let Y^n be the groundtruth translation in the target dialect and $\hat{Y}^n$ be the model's predicted translation. Then, we define

$$G^n_{\text{gt}} = \Phi\big(Y^n\big) = \{\, g \in \mathcal{G} \mid g \text{ appears in } Y^n \} \tag{6}$$

$$G^n_{\text{pred}} = \Phi\big(\hat{Y}^n\big) = \{\, g \in \mathcal{G} \mid g \text{ appears in } \hat{Y}^n \} \tag{7}$$

Then compute for each task n:

$$\text{TP}_n = \big| G^n_{\text{pred}} \cap G^n_{\text{gt}} \big| \tag{8}$$

$$\text{FP}_n = \big| G^n_{\text{pred}} \setminus G^n_{\text{gt}} \big| \tag{9}$$

$$\text{FN}_n = \big| G^n_{\text{gt}} \setminus G^n_{\text{pred}} \big| \tag{10}$$

Aggregating over all N tasks yields the precision and recall:

$$P_{\text{DM}} = \frac{\sum_{n=1}^{N} \text{TP}_n}{\sum_{n=1}^{N} (\text{TP}_n + \text{FP}_n)}, \tag{11}$$

$$R_{\text{DM}} = \frac{\sum_{n=1}^{N} \text{TP}_n}{\sum_{n=1}^{N} (\text{TP}_n + \text{FN}_n)}, \tag{12}$$

and the final F_1 score:

$$F_{1,\text{DM}} = \frac{2\, P_{\text{DM}}\, R_{\text{DM}}}{P_{\text{DM}} + R_{\text{DM}}}. \tag{13}$$

This metric thus captures how accurately the model translates dialect-specific constructs from the source dialect into their correct counterparts in the target dialect.

Exact Matching (EM) measures the percentage of examples where the predicted SQL statement is identical to the ground truth SQL statement [13]. A predicted SQL is considered correct only if all tokens match exactly with the ground truth statement.

B. Execution-based Metrics

Execution Accuracy (EX) measures the proportion of examples in the evaluation set for which the executed results of both the predicted and ground-truth SQL statements are identical, relative to the total number of statements. Let V_n denote the result set executed by the n-th ground-truth SQL Y_n, and $\hat{V}_n$ the result set executed by the predicted SQL $\hat{Y}_n$. The EX metric is defined as:

$$EX = \frac{1}{N} \sum_{n=1}^{N} \mathbb{1}(V_n, \hat{V}_n) \tag{14}$$

where $\mathbb{1}(\cdot)$ is an indicator function, defined as:

$$\mathbb{1}(V, \hat{V}) = \begin{cases} 1, & V = \hat{V} \\ 0, & V \neq \hat{V} \end{cases} \quad (15)$$

Execution consistency varies depending on the type of SQL statement. For **SELECT queries**, execution consistency is evaluated by comparing the result sets of the translated queries and ground-truth queries, focusing on row order and duplicates, where specified. For **non-query statements** such as INSERT, UPDATE, and DELETE, execution consistency is evaluated by comparing the number of affected rows between the translated statements and the ground-truth.

V. Experimental Setup

We aim to answer the following key research questions (RQs) that explore the utility of DLBench for evaluating diverse LLMs for SQL translation tasks:

1) **RQ1 (Effectiveness of LLMs in SQL Translation)**: How do the recent advanced general and code LLMs perform in SQL translation?
2) **RQ2 (Effectiveness of Enhanced Prompt)**: How do enhanced prompts affect the performance of recent LLMs in SQL translation?
3) **RQ3 (Error Analysis)**: What are the main types of errors that occur in SQL translation?

A. Models

The studied LLMs are listed in Table III. We evaluate DLBench using four state-of-the-art open-source LLMs (SQLCoder [17], codellama [18], deepseek-coder [19], deepseek [20]) and three close-source models (GPT-3.5-turbo [21] ,GPT-4o [22] and Gemini-2.5-flash [23]). Notably, we select these models for their strong performance on SQL-related tasks: each has been fine-tuned or pre-trained for cross-dialect SQL generation, endowing them with advanced SQL reasoning capabilities. Among them, Gemini-2.5-flash is not limited to prompt-based inference but also supports tool-using functionalities such as automatically consulting documentation, database schemas, reference implementations, or dialect-specific resources. Our primary test subject is GPT-3.5-turbo.

TABLE III: Studied Large Language Models

Base Model	Model	Size
StarCoder [24]	SQLCoder-7B [17]	7B
CodeLlama [18]	CodeLlama-7B-Instruct [18]	7B
Deepseek [20]	Deepseek-Coder-6.7B-Instruct [19]	6.7B
Deepseek [20]	DeepSeek-R1-Distill-Llama-8B [20]	8B
–	GPT-3.5-turbo [21]	–
–	GPT-4o [22]	–
–	Gemini-2.5-flash [23]	–

B. Prompting Strategies

We assess LLMs on our SQL translation benchmark using prompt-based evaluations. Figure 5 shows the prompt template used for our evaluation. Following the design of Pan et al. [25], we craft an *Initial Prompt* template composed of four components: (1) System Message (2) Task Description (3) Schema Information and (4) Output Constraints. Under RQ1, we assess model performance using the *Initial Prompt* alone. For RQ2, we further investigate two enhanced prompting techniques: few-shot prompting and knowledge-augmented prompting.

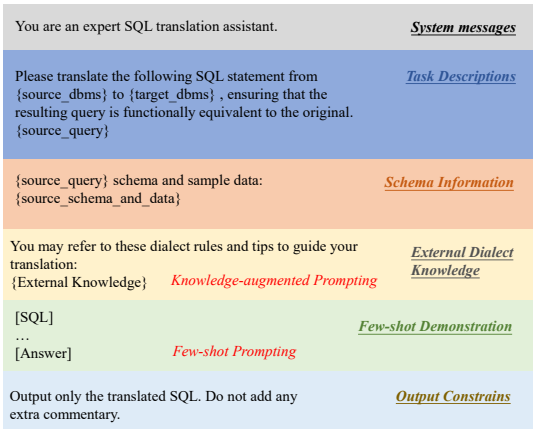

Fig. 5: The prompt template for our evaluation.

- **Few-shot Prompting (FS):** The technique of few-shot prompting is widely used in both practical applications and well-designed research, which has been proven efficient for eliciting better performance of LLMs [26]. In the few-shot setting, we provide the model with 3 translating examples within the prompt, aiming to improve model performance through limited task-specific examples. To ensure task consistency, these examples are randomly selected from the same translation direction (e.g., MySQL to PostgreSQL).
- **Knowledge-augmented Prompting (KA):** Knowledge-augmented prompting is widely used in code tasks to infuse models with domain-specific information [27]. In this setup, we augment prompts with our labeled SQL dialect knowledge to test whether extra task-specific information improves SQL translation accuracy.

C. Experimental Environment

We perform our experiments on a workstation an 8-core "11th Gen Intel(R) Core(TM) i7-11700@2.50GHz" processor, 64GB of memory, and NVIDIA RTX A6000 with 48GB of VRAM running Ubuntu 22.04.1 LTS. For open-source LLMs, we deploy a local API server based on vLLM [28] which is a unified library for LLM serving and inference. All models are not quantized and we use their original precisions. Model temperature can control the randomness in the generated results of models [29]. Specifically, we follow Gu et al. [30] and set the temperature to 0.8. To mitigate evaluation randomness, we repeat each experiment five times under the same configuration. Following the self-consistency strategy [31], we select the answer with the highest consistency across these runs, which helps reduce the impact of output randomness. For the rest of the parameters, we use the default settings in vLLM, to ensure a fair comparison.

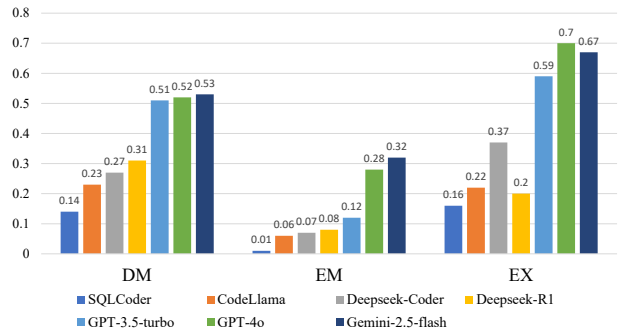

Fig. 6: Overall translation performance of LLMs on DL-BENCH, covering BIRDTRANS and BUTTERTRANS in terms of DM, EM, and EX

VI. RESULTS AND ANALYSIS

A. Effectiveness of LLMs in SQL Translation

The overall performance of the studied LLMs on DLBENCH is shown in Figure 6. From the model perspective, GPT-4o achieves the highest overall performance, with DM $= 0.52$, EM $= 0.28$, and EX $= 0.70$. Gemini-2.5-flash follows closely, reaching DM $= 0.53$, EM $= 0.32$, and EX $= 0.67$. This superiority can be attributed to its larger parameter scale and the diversity of its pre-training corpus. Among the code-specialized models, Deepseek-Coder leads with EX $= 0.37$, whereas SQLCoder performs worst across DM, EM, and EX. From the metrics perspective, EM remains uniformly low across all models, indicating that EM is overly sensitive to syntactic details and often fails to recognize semantically equivalent queries expressed differently. The gap between DM and EX varies across models: Deepseek-Coder actually surpasses its DM in EX (0.37 vs. 0.27), demonstrating its ability to generate executable variants that diverge from strict dialect patterns and reflect a diversity of valid dialect mappings. In contrast, Deepseek-R1 shows a larger drop ($\Delta = 0.11$) and CodeLlama a smaller one ($\Delta = 0.01$), revealing that even when dialect recognition succeeds, semantic usage can still fail. Figure 9(b) illustrates such an example, where the model applies the "`ROW`" function correctly but overlooks the required type conversion.

Finding 1: Among all evaluated LLMs, GPT-4o achieves the highest translation performance. Furthermore, DM and EX serve as more informative indicators of SQL translation quality than EM.

Table IV and Table V detail the evaluation results of each model on different datasets (BIRDTRANS and BUTTERTRANS) and various target DBMSs. From the dataset perspective, most models achieve a moderate improvement in EX on BUTTERTRANS; for instance, GPT-3.5-turbo's EX on MySQL increases from 62.71% on BIRDTRANS to 76.47% on BUTTERTRANS. This gap highlights the models' greater challenges in processing deeply nested, lengthy SQL queries in BIRDTRANS, as opposed to the more diverse yet concise statements encountered in BUTTERTRANS. From the target DBMS perspective, MySQL, PostgreSQL, MariaDB and DuckDB consistently yield the highest EX scores, while MonetDB and ClickHouse remain the most challenging. For example, on BUTTERTRANS, GPT-3.5-turbo achieves an EX score of 76.47% when translating to MySQL, but only 48.28% when translating to MonetDB. Similarly, GPT-4o drops from 53.45% EX on PostgreSQL to 42.84% on ClickHouse, and Gemini-2.5-flash decreases from 97.22% on MariaDB to 48.65% on MonetDB, confirming the consistent difficulty of these DBMSs across different models. These variations underscore differences in dialect support across DBMS platforms.

Finding 2: Models generalize better to shorter, varied-dialect queries than to long, complex real-world SQL, and DBMS-specific features drive significant performance differences, with MySQL/PostgreSQL/MariaDB/DuckDB easiest and MonetDB/ClickHouse hardest.

B. Effectiveness of Enhanced Prompt

In RQ2, we investigate two prompting strategies, **FS** and **KA**, to evaluate their impact on SQL translation performance. Due to space constraints, we only show the impact of GPT-3.5-turbo on EX metric. Similar observations hold for other LLMs and can be found in our released artifact.

As shown in Figure 7, both FS and KA prompting strategies improve the SQL translation performance. Specifically, on the BIRDTRANS dataset, the improvement is particularly pronounced for DuckDB and ClickHouse. In these two systems, KA demonstrates a clear advantage, achieving significantly higher EX scores compared to both IP and FS. Compared to IP, KA improves EX by 15% on DuckDB and 22% on ClickHouse; compared to FS, the improvements are 5% and 5%, respectively. The only exception is PostgreSQL in the BIRDTRANS dataset, where KA causes a slight performance drop. A possible reason is that the model already possesses a strong understanding of the basic PostgreSQL dialect, and the addition of external knowledge or longer context may introduce interference rather than benefits. However, this phenomenon is not observed in the more complex dialects of the BUTTERTRANS dataset, where KA consistently demonstrates clear advantages. Under the BUTTERTRANS dataset, KA again leads to consistent EX gains across all target DBMSs. Compared to IP, KA improves EX by 8% on MySQL, 7% on PostgreSQL, 14% on MariaDB, 10% on MonetDB, 9% on DuckDB, and 17% on ClickHouse; relative to FS, the gains are 14%, 7%, 10%, 7%, 8%, and 15%, respectively. These results demonstrate that knowledge augmentation effectively enhances translation accuracy.

Finding 3: Both enhanced prompting strategies improve translation performance to varying degrees, with the knowledge-augmented prompting (KA) showing greater potential overall.

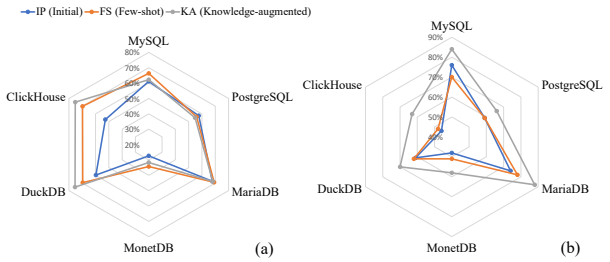

Fig. 7: Execution Accuracy (EX) of GPT-3.5-turbo under different prompting strategies: (a) Performance on BIRDTRANS dataset, (b) Performance on BUTTERTRANS dataset.

C. Error Analysis

To gain a comprehensive understanding of SQL translation errors, we conduct a fine-grained analysis on 300 randomly sampled examples, and systematically categorize the observed errors following the principles of open coding [32] to uncover recurring patterns and underlying causes. Although not determined through formal statistical power analysis, the sample size aligns with conventions in prior studies on code tasks (e.g. SPIDER2.0 [33], UNITRANS [34]). This sample size allowed us to identify stable and recurring error patterns across models. Figure 8 illustrates the classification results. The errors are grouped into four major categories: E1 (Syntax Errors), E2 (Semantic Errors), E3 (Logic Errors), and E4 (Others). Limited by the space, we provide only three representative examples in Figure 9.

E1. Syntax Errors (46%) are typically caught during parsing. We identify five common subtypes: *Keyword Error*, where reserved keywords are used incorrectly; *Function Syntax Error*, involving incorrect function formats (Figure 9(a) illustrates this case, where the misuse of a non-existent function resulted in a syntax error); *Statement Structure Error*, such as improper clause ordering; *Identifier Quoting Difference*, where mismatched quotation conventions lead to invalid identifiers; and *Data Type Error*, where unsupported data types are used. These errors indicate that LLMs may lack precise dialect-specific syntax awareness, especially when dealing with reserved keywords and varying quoting conventions.

E2. Semantic Errors (15%) occur when the generated query is syntactically valid but produces incorrect semantics in the context of the target DBMS. We observe two key subtypes: *Schema Error*, which arises from incorrect assumptions about table or column names, and *Type Incompatibility*, where incompatible data types are used in expressions or joins (Figure 9(b) illustrates this case, where PostgreSQL does not support comparisons between numeric values and non-numeric strings, resulting in a runtime error). Such errors suggest that LLMs struggle with aligning the input SQL's schema context to the target dialect's data definitions, often due to hallucinated or misinterpreted metadata.

E3. Logic Errors (28%) represent mistakes in query intent or execution logic, where the structure is acceptable, but the behavior diverges from the original query's semantics. These

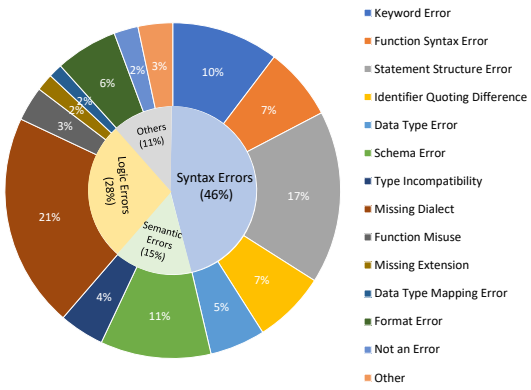

Fig. 8: Taxonomy and statistics of errors.

include *Missing Dialect*, where dialect-specific features are omitted (Figure 9(c) illustrates this case, in which the required ClickHouse engine specification was dropped); *Function Misuse*, where an incorrect function is invoked; *Missing Extension*, when required extensions are assumed available; and *Data Type Mapping Error*, where incompatible or incorrect type conversions lead to invalid behavior. These errors highlight LLMs' limitations in modeling nuanced dialect-specific behaviors and implicit assumptions, revealing a gap in understanding execution-time dependencies across dialects.

E4. Others (11%) include a variety of miscellaneous errors that do not fit neatly into the previous categories. These errors often involve malformed query fragments caused by incomplete generation, the use of unsupported or hallucinated SQL constructs, and other unpredictable failures. Although less frequent, such issues highlight robustness limitations in prompt formulation, decoding stability, or model generalization.

> **Finding 4:** Syntax Errors and Logic Errors are Top-2 dominating error types, accounting for 74% of SQL translation failures.

VII. DISCUSSION

LLM- VS. Non-LLM-Based SQL Translation. In this section, we compare the effectiveness of LLM-based and non-LLM-based SQL translation techniques to underscore the potential of LLMs in this domain. Non-LLM-based techniques follow a rule-based pipeline: a SQL parser first produces an abstract syntax tree (AST), then handcrafted transformation rules map dialect-specific constructs to their equivalents (e.g., replacing MySQL's keyword "`DATETIME`" with PostgreSQL's "`TIMESTAMP`"). While offering high control and consistency, it struggles with complex queries, database-specific behaviors, and requires high maintenance costs. Specifically, we compare two state-of-the-art traditional rule-based SQL translation methods, namely JOOQ [3] and SQLGLOT [6], to highlight our contribution to SQL translation. We evaluate on

TABLE IV: Evaluation results of each model for SQL translation on BIRDTRANS dataset

Models	SQLite → MySQL			SQLite → PostgreSQL			SQLite → MariaDB			SQLite → MonetDB			SQLite → DuckDB			SQLite → ClickHouse		
	DM	EM	EX	DM	EM	EX	DM	EM	EX	DM	EM	EX	DM	EM	EX	DM	EM	EX
Code Large Language Models																		
SC	14.71%	0.64%	19.96%	13.00%	0.12%	11.34%	13.71%	1.65%	18.70%	13.23%	0.20%	7.46%	16.28%	0.00%	20.64%	10.41%	0.43%	16.08%
CL	30.57%	15.78%	33.98%	26.39%	0.21%	18.90%	29.16%	12.19%	29.59%	14.01%	0.00%	14.26%	27.29%	1.53%	15.98%	19.09%	4.57%	34.56%
DSC	30.14%	15.78%	44.60%	27.13%	0.63%	41.17%	29.02%	11.06%	40.97%	20.68%	0.25%	20.56%	19.14%	0.69%	49.65%	20.25%	5.20%	41.95%
General Large Language Models																		
DSR1	28.33%	4.64%	35.10%	28.24%	0.42%	6.30%	28.89%	2.43%	18.04%	17.09%	0.66%	4.14%	28.32%	0.00%	9.97%	16.41%	1.09%	11.96%
GPT35T	48.63%	29.79%	62.71%	44.17%	1.20%	57.56%	43.25%	18.37%	67.80%	41.49%	0.66%	27.12%	43.81%	8.76%	60.53%	39.24%	17.83%	52.61%
GPT-4o	26.18%	24.15%	81.32%	45.69%	4.83%	84.03%	22.59%	9.92%	85.20%	38.80%	1.99%	78.77%	43.74%	1.16%	87.94%	34.56%	16.30%	85.22%
Gemini	36.56%	18.18%	81.82%	37.89%	3.03%	60.61%	56.25%	27.27%	72.73%	37.21%	3.03%	54.55%	35.42%	0.00%	76.47%	37.25%	8.82%	85.29%

Abbreviations: SC = SQLCoder; CL = CodeLlama; DSC = Deepseek-Coder; DSR1 = Deepseek-R1; GPT35T = GPT-3.5-turbo; Gemini = Gemini-2.5-flash.

TABLE V: Evaluation results of each model for SQL translation on BUTTERTRANS dataset

Models	Many → MySQL			Many → PostgreSQL			Many → MariaDB			Many → MonetDB			Many → DuckDB			Many → ClickHouse		
	DM	EM	EX	DM	EM	EX	DM	EM	EX	DM	EM	EX	DM	EM	EX	DM	EM	EX
Code Large Language Models																		
SC	11.17%	0.12%	18.40%	15.70%	2.70%	13.94%	14.12%	5.12%	21.57%	11.20%	1.21%	10.46%	15.80%	2.80%	24.09%	11.20%	2.20%	18.46%
CL	15.58%	35.29%	47.05%	22.59%	5.05%	15.39%	25.81%	7.49%	22.76%	22.40%	0.57%	0.95%	22.49%	2.57%	8.95%	18.80%	3.57%	16.61%
DSC	29.63%	23.52%	35.29%	34.56%	6.79%	33.49%	35.26%	21.90%	46.10%	28.02%	4.58%	8.78%	30.01%	8.19%	23.21%	19.12%	6.21%	20.34%
General Large Language Models																		
DSR1	14.92%	11.76%	35.29%	24.82%	6.16%	16.58%	59.01%	32.42%	48.84%	50.17%	15.83%	20.99%	44.21%	15.93%	22.91%	19.83%	5.12%	13.35%
GPT35T	32.43%	17.64%	76.47%	60.42%	13.39%	59.71%	65.99%	15.57%	74.38%	70.95%	26.71%	48.28%	68.30%	14.52%	60.54%	44.58%	6.52%	46.46%
GPT-4o	26.67%	46.06%	47.05%	66.37%	34.80%	53.45%	82.71%	62.52%	86.41%	68.52%	49.91%	48.58%	75.86%	55.27%	50.15%	56.87%	22.80%	42.84%
Gemini	27.27%	57.80%	55.82%	61.39%	35.14%	64.86%	94.89%	88.89%	97.22%	72.00%	51.35%	48.65%	65.08%	58.33%	47.22%	53.33%	35.14%	48.65%

Abbreviations: SC = SQLCoder; CL = CodeLlama; DSC = Deepseek-Coder; DSR1 = Deepseek-R1; GPT35T = GPT-3.5-turbo; Gemini = Gemini-2.5-flash.

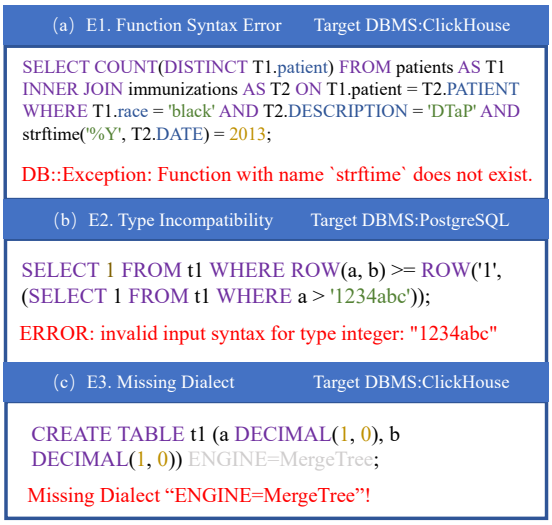

Fig. 9: Representative examples of LLM translation failures.

DLBENCH using the above two non-LLM baselines and GPT-3.5-turbo. DBMSs unsupported by the rule-based methods are excluded, and EX is used as the evaluation metric.

TABLE VI: Execution Accuracy of LLM- vs. Non-LLM-Based Techniques on DLBENCH

Technique	BIRDTRANS	BUTTERTRANS
JOOQ	15.4%	18.2%
SQLGlot	8.2%	5.2%
GPT-3.5-turbo	54.9%	62.6%

As shown in Table VI, non-LLM-based approaches perform poorly on both datasets. JOOQ achieves only 15.4% EX on BIRDTRANS and 18.2% on BUTTERTRANS, while SQLGLOT fails to generalize effectively, scoring as low as 8.2% and 5.2%, respectively. These results indicate that handcrafted rule-based systems struggle to handle the diverse and complex SQL dialects featured in DLBENCH, especially when encountering dialect-specific syntax and semantic nuances not covered by predefined rules. In contrast, GPT-3.5-turbo demonstrates significantly higher performance, highlighting the advantage of LLMs in SQL translation tasks.

Implications. Although LLM-based SQL translation methods surpass traditional approaches, there remains room for improvement. We recommend that researchers leverage DLBENCH for targeted evaluation and fine-tuning, and construct richer dialect-specific datasets to improve robustness. For practitioners, our findings reveal that larger models and the integration of external knowledge (e.g., dialect documentation) consistently enhance translation quality, and retrieval-augmented generation (RAG) is a practical strategy for deployment. Finally, our error analysis highlights diverse risks even in state-of-the-art LLMs, providing clear guidance for advancing both research and practice in SQL translation.

Long-term Significance of DLBENCH. A natural concern is whether the benchmark will remain challenging as LLMs continue to advance. Although our latest results show notable progress—for example, GPT-4o achieves 70% EX—SQL translation remains far from solved. The initial *35% success rate* of GPT-4o-mini in automatic translations and the continued reliance on human annotation underscore the inher-

ent difficulty of the task. To ensure long-term applicability, DLBENCH is designed with extensibility in mind: it can readily incorporate new dialect phenomena, more complex queries, and additional DBMSs as they emerge. Moreover, we constructed a dialect feature knowledge base by crawling features from seven major DBMSs, enabling experts to focus on validating dialect-specific conversions rather than performing full manual annotation, thereby reducing effort and supporting future expansion.

Threats to Validity. Threats to external validity concern whether our results can be generalized to other experimental settings. Key factors include the multi-dialect DBMSs, LLMs, and benchmark sources. In terms of database selection, we evaluate three source DBMSs and six popular target DBMSs, and results may differ for other DBMSs. However, since the selected DBMSs are among the most widely used, our benchmark and findings remain crucial for evaluating LLM performance in SQL translation tasks. Furthermore, we plan to apply the method proposed in this paper to additional DBMSs in the future to extend both our benchmark and findings. Due to resource constraints, we limit our evaluation to four open-source LLMs (6.7–8 B parameters) and three close-source LLMs, thereby covering a broad spectrum of model complexity and capability. In terms of datasets, we gather our datasets from various sources, including Text-to-SQL datasets and DBMS test suites, which exhibit different characteristics and reflect real-world project scenarios.

Threats to internal validity primarily include issues with benchmark construction and the insufficient coverage of evaluation metrics. In terms of benchmark construction, the quality and level of detail in the natural language descriptions of the automatic translation process may affect the translation results generated by the LLM. Human translations may also be influenced by subjective factors. To mitigate this threat, we conducted a final review to ensure that the constructed benchmark adheres to predefined quality standards. The limitations of the evaluation metrics are especially evident for certain query types, where the metrics might not fully capture the functional correctness or performance of translations. In the future, we plan to incorporate additional metrics such as execution time comparisons, logical consistency checks, and performance benchmarking to provide a more comprehensive assessment.

VIII. Related Work and Existing Dataset

Text-to-SQL Systems and Benchmarks. Large language models have recently achieved strong results on Text-to-SQL tasks, using in-context learning techniques—such as chain-of-thought [35], [36], question decomposition [35]–[37], and self-reflection [35]–[38]—and supervised fine-tuning methods exemplified by CodeS [39] and SQL-PaLM [40]. Evaluation typically relies on manually crafted benchmarks like SPIDER [13] and metrics such as Exact Matching Accuracy and Execution Accuracy. However, these benchmarks mostly focus on a single dialect (e.g., SQLite), leaving cross-dialect translation unexamined. To fill this gap, we propose a benchmark for automatic evaluation of SQL translation across multiple dialects, reflecting real-world usage and exposing dialect-specific failure modes [1].

Code Translation and Benchmarks. Code translation is essential for cross-language migration, and recent LLMs—such as Deepseek [20], StarCoder [24], and GPT-4 [9]—have achieved strong results through prompt engineering and fine-tuning. Benchmarks like CoST [41], XL-Cost [42], CodeXGLUE [43], TransCoder-test [44], and G-TransEval [45] assess translation quality using metrics such as Success@k and Build@k. In contrast, SQL translation across dialects is more challenging due to diverse syntax and semantics. Existing benchmarks focus on programming languages, limiting comprehensive evaluation of LLMs on SQL tasks. Moreover, pretraining corpora often lack dialectal variety, underscoring the need for dedicated SQL translation benchmarks and datasets.

Code-related Benchmarks Construction. Recent LLMs exhibit remarkable capabilities in software development [29] and are evaluated using diverse code-related benchmarks spanning code generation [46]–[52], defect detection [53]–[57], program repair [58]–[62], and code summarization [63]–[67]. Our work differs from these benchmarks by targeting the SQL translation task with a purpose-built benchmark. We adopt the same five-phase construction process—Design, Construction, Evaluation, Analysis, and Release—and follow the HOW2BENCH principles of reliability, validity, open access, and reproducibility [68] to ensure high standards and foster a reliable, transparent benchmarking environment. Consequently, we introduce a SQL translation benchmark, DLBENCH, that adheres to this methodology while addressing the unique syntactic and semantic challenges of cross-dialect translation.

IX. Conclusion

In this paper, we presented DLBENCH, the first comprehensive benchmark for evaluating the SQL translation capabilities of LLMs. DLBENCH comprises 6,402 translation tasks across seven popular DBMSs and 9,320 SQL dialects, and includes both content-matching and execution-based evaluation metrics to measure structural, syntactic, and semantic fidelity. We conducted experiments on seven advanced LLMs of diverse architectures and sizes and the results show that DLBENCH effectively distinguishes their performance profiles, highlighting strengths on common constructs and exposing limitations when handling complex, dialect-specific features.

Acknowledgement

We sincerely thank the anonymous reviewers for their valuable and insightful feedback. This research was supported by the Natural Science Foundation of China (Grant No. 62272400) and Fujian Provincial Natural Science Foundation of China (Grant No. 2025J010002). Rongxin Wu is the corresponding author and works as a member of Xiamen Key Laboratory of Intelligent Storage and Computing in Xiamen University.

REFERENCES

[1] T. Kraska, T. Li, S. Madden, M. Markakis, A. Ngom, Z. Wu, and G. X. Yu, "Check out the big brain on brad: simplifying cloud data processing with learned automated data meshes," *Proceedings of the VLDB Endowment*, no. 11, pp. 3293–3301, 2023.

[2] A. L. Ngom and T. Kraska, "Mallet: Sql dialect translation with llm rule generation," in *Proceedings of the Seventh International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2024, pp. 1–5.

[3] L. Eder, "Jooq: Fluent api for typesafe sql query construction and translation," https://www.jooq.org/, 2024, accessed: 2024-12-29.

[4] A. Iqbal and R. Colomo-Palacios, "Key opportunities and challenges of data migration in cloud: results from a multivocal literature review," *Procedia computer science*, vol. 164, pp. 48–55, 2019.

[5] Amazon Web Services, "Amazon web services schema conversion tool," https://aws.amazon.com/dms/schema-conversion-tool/, accessed: 2025-05-23.

[6] T. Mao, "Sqlglot: Python sql parser and translator," https://github.com/tobymao/sqlglot, 2024, accessed: 2024-12-29.

[7] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo *et al.*, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," *Advances in Neural Information Processing Systems*, vol. 36, pp. 42 330–42 357, 2023.

[8] J. Fu, J. Liang, Z. Wu, and Y. Jiang, "Sedar: Obtaining high-quality seeds for dbms fuzzing via cross-dbms sql transfer," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[9] OpenAI, "Gpt-4 technical report," *CoRR*, vol. abs/2303.08774, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.08774

[10] J. Manyika, "An overview of bard: an early experiment with generative ai," Google, 2023, https://ai.google/static/documents/google-about-bard.pdf.

[11] Transaction Processing Performance Council, "TPC-DS: Decision Support Benchmark," https://www.tpc.org/tpcds/, 2011, accessed: 2025-05-15.

[12] W. Zhou, Y. Gao, X. Zhou, and G. Li, "Cracksql: A hybrid sql dialect translation system powered by large language models," *arXiv preprint arXiv:2504.00882*, 2025.

[13] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman *et al.*, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," *arXiv preprint arXiv:1809.08887*, 2018.

[14] MySQL, "Mysql server," https://github.com/mysql/mysql-server/tree/trunk, accessed: 2025-03-22.

[15] PostgreSQL, "Postgresql," https://github.com/postgres/postgres, accessed: 2025-03-22.

[16] International Organization for Standardization, "ISO/IEC 9075-1:2023 – Information technology — Database languages — SQL — Part 1: Framework," https://www.iso.org/standard/76583.html, 2023, accessed: 2025-05-19.

[17] D. AI, "Sqlcoder-7b," 2023, accessed: 2023-03-23. [Online]. Available: https://huggingface.co/defog/sqlcoder-7b-2

[18] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[19] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[20] DeepSeek, "Deepseek-r1-distill-llama-8b," 2025. [Online]. Available: https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Llama-8B

[21] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[22] OpenAI, "Hello gpt-4o," https://openai.com/index/hello-gpt-4o/, May 13 2024, accessed: YYYY-MM-DD.

[23] "Google gemini," https://gemini.google.com/, accessed: 2025-09-15.

[24] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[25] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[26] L. Nan, Y. Zhao, W. Zou, N. Ri, J. Tae, E. Zhang, A. Cohan, and D. Radev, "Enhancing text-to-sql capabilities of large language models: A study on prompt design strategies," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 14 935–14 956.

[27] G. Ou, M. Liu, Y. Chen, X. Du, S. Wang, Z. Zhang, X. Peng, and Z. Zheng, "Enhancing llm-based code translation in repository context via triple knowledge-augmented," *arXiv preprint arXiv:2503.18305*, 2025.

[28] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[29] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[30] A. Gu, B. Rozière, H. Leather, A. Solar-Lezama, G. Synnaeve, and S. I. Wang, "Cruxeval: A benchmark for code reasoning, understanding and execution," *arXiv preprint arXiv:2401.03065*, 2024.

[31] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," 2023. [Online]. Available: https://arxiv.org/abs/2203.11171

[32] S. H. Khandkar, "Open coding," *University of Calgary*, vol. 23, no. 2009, 2009.

[33] F. Lei, J. Chen, Y. Ye, R. Cao, D. Shin, H. Su, Z. Suo, H. Gao, W. Hu, P. Yin *et al.*, "Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows," *arXiv preprint arXiv:2411.07763*, 2024.

[34] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, "Exploring and unleashing the power of large language models in automated code translation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1585–1608, 2024.

[35] M. Pourreza and D. Rafiei, "Din-sql: Decomposed in-context learning of text-to-sql with self-correction," *Advances in Neural Information Processing Systems*, vol. 36, pp. 36 339–36 348, 2023.

[36] Y. Xie, X. Jin, T. Xie, M. Lin, L. Chen, C. Yu, L. Cheng, C. Zhuo, B. Hu, and Z. Li, "Decomposition for enhancing attention: Improving llm-based text-to-sql through workflow paradigm," *arXiv preprint arXiv:2402.10671*, 2024.

[37] B. Wang, C. Ren, J. Yang, X. Liang, J. Bai, L. Chai, Z. Yan, Q.-W. Zhang, D. Yin, X. Sun *et al.*, "Mac-sql: A multi-agent collaborative framework for text-to-sql," *arXiv preprint arXiv:2312.11242*, 2023.

[38] S. Talaei, M. Pourreza, Y.-C. Chang, A. Mirhoseini, and A. Saberi, "Chess: Contextual harnessing for efficient sql synthesis," *arXiv preprint arXiv:2405.16755*, 2024.

[39] 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," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–28, 2024.

[40] R. Sun, S. Ö. Arik, A. Muzio, L. Miculicich, S. Gundabathula, P. Yin, H. Dai, H. Nakhost, R. Sinha, Z. Wang *et al.*, "Sql-palm: Improved large language model adaptation for text-to-sql (extended)," *arXiv preprint arXiv:2306.00739*, 2023.

[41] M. Zhu, K. Suresh, and C. K. Reddy, "Multilingual code snippets training for program translation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 36, no. 10, 2022, pp. 11 783–11 790.

[42] M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C. K. Reddy, "Xlcost: A benchmark dataset for cross-lingual code intelligence," *arXiv preprint arXiv:2206.08474*, 2022.

[43] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[44] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.

[45] M. Jiao, T. Yu, X. Li, G. Qiu, X. Gu, and B. Shen, "On the evaluation of neural code translation: Taxonomy and benchmark," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1529–1541.

[46] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[47] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[48] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abq1158

[49] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html

[50] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with apps," *NeurIPS*, 2021.

[51] C. S. Xia, Y. Deng, and L. Zhang, "Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm," *arXiv preprint arXiv:2403.19114*, 2024.

[52] J. Chen, Q. Zhong, Y. Wang, K. Ning, Y. Liu, Z. Xu, Z. Zhao, T. Chen, and Z. Zheng, "Rmcbench: Benchmarking large language models' resistance to malicious code," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 995–1006.

[53] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-2_Li_paper.pdf

[54] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 10 197–10 207. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html

[55] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Trans. Software Eng.*, vol. 48, no. 9, pp. 3280–3296, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3087402

[56] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *CoRR*, vol. abs/2001.02334, 2020. [Online]. Available: http://arxiv.org/abs/2001.02334

[57] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=B1lnbRNtwr

[58] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, C. S. Pasareanu and D. Marinov, Eds. ACM, 2014, pp. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[59] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019. [Online]. Available: https://doi.org/10.1145/3340544

[60] C. Le Goues, N. J. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of C programs," *IEEE Trans. Software Eng.*, vol. 41, no. 12, pp. 1236–1256, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2015.2454513

[61] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1430–1442. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00125

[62] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs?: An evaluation on quixbugs," in *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*. IEEE, 2022, pp. 69–75. [Online]. Available: https://doi.org/10.1145/3524459.3527351

[63] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. [Online]. Available: https://doi.org/10.18653/v1/p16-1195

[64] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[65] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 397–407. [Online]. Available: https://doi.org/10.1145/3238147.3238206

[66] N. Muennighoff, Q. Liu, A. R. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, and S. Longpre, "Octopack: Instruction tuning code large language models," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=mw1PWNSWZP

[67] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 2023, pp. 31 693–31 715. [Online]. Available: https://proceedings.mlr.press/v202/shrivastava23a.html

[68] J. Cao, Y.-K. Chan, Z. Ling, W. Wang, S. Li, M. Liu, R. Qiao, Y. Han, C. Wang, B. Yu, P. He, S. Wang, Z. Zheng, M. R. Lyu, and S.-C. Cheung, "How should we build a benchmark? revisiting 274 code-related benchmarks for llms," 2025. [Online]. Available: https://arxiv.org/abs/2501.10711