SQLStructEval: Structural Evaluation of LLM Text-to-SQL Generation

Anonymous ACL submission

Abstract

Despite strong performance on Text-to-SQL benchmarks, it remains unclear whether LLM-generated SQL programs are structurally reliable. In this work, we investigate the structural behavior of LLM-generated SQL queries and introduce SQLStructEval, a framework for analyzing program structures through canonical abstract syntax tree (AST) representations. Our experiments on the Spider benchmark show that modern LLMs often produce structurally diverse queries for the same input, even when execution results are correct, and that such variance is frequently triggered by surface-level input changes such as paraphrases or schema presentation. We further show that generating queries in a structured space via a compile-style pipeline can improve both execution accuracy and structural consistency. These findings suggest that structural reliability is a critical yet overlooked dimension for evaluating LLM-based program generation systems. Our code is available at https://anonymous.4open.science/r/StructEval-2435.

1 Introduction

LLMs have achieved strong performance in program generation tasks such as code synthesis and Text-to-SQL translation (Chen et al., 2021; Austin et al., 2021). These capabilities enable AI systems that translate natural language into executable programs interacting with databases, APIs, and software environments. As such systems are increasingly deployed in real-world settings, evaluating the reliability of generated programs has become critical. Current evaluation protocols primarily focus on functional correctness. Code generation is typically evaluated using unit tests or execution-based metrics (Chen et al., 2021), while Text-to-SQL benchmarks rely on execution accuracy, which treats all execution-correct programs as equivalent (Yu et al., 2018).

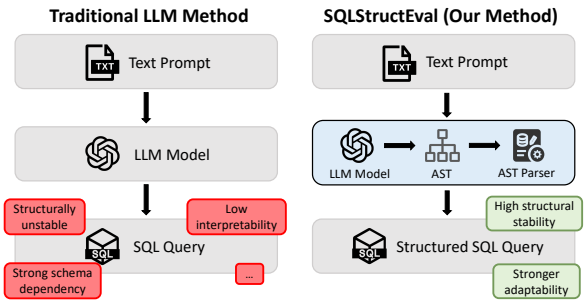


Figure 1: Comparison between traditional LLM-based Text-to-SQL generation and our proposed SQLStructEval framework. Traditional methods directly generate SQL queries from text prompts, which often leads to structurally inconsistent outputs across repeated generations. In contrast, SQLStructEval introduces an AST-based representation that enables explicit structural analysis and improves the stability and comparability of generated programs.

However, programs with identical outputs can differ substantially in structure. Multiple structurally distinct queries may produce the same result, especially in the presence of redundant joins, nested queries, or alternative formulations (Yu et al., 2018; Zhong et al., 2020). Such differences affect interpretability, robustness, and downstream reasoning, but remain invisible to execution-based metrics. In practice, LLM-generated programs often exhibit structural variance, where repeated generations yield different query structures (Wang et al., 2023; Rajkumar et al., 2022).

In this work, we investigate the structural reliability of LLM program generation, which is largely overlooked by existing evaluation protocols. We propose SQLStructEval, a structure-aware evaluation framework that represents SQL queries as canonicalized ASTs and measures structural consistency, diversity, and robustness under repeated sampling and semantically invariant perturbations.

Using this framework, we show that LLMs frequently produce structurally diverse programs for

the same input, even when execution-correct. This variability is partly driven by sensitivity to semantically invariant perturbations, such as paraphrases and schema presentation changes, indicating that execution accuracy may overestimate model reliability. As a case study, we further examine whether structured intermediate representations improve both generation accuracy and structural consistency, and show that compile-style generation enhances both execution performance and structural stability.

Our contributions are as follows:

- We identify structural instability as an underexplored reliability issue in Text-to-SQL, showing that execution-correct programs can exhibit substantial structural variation.
- We propose SQLSTRUCTEVAL, a structure-aware framework that measures structural consistency, diversity, and robustness via canonicalized AST representations.
- We show that execution accuracy often overestimates reliability, and that structural instability is partly associated with semantically invariant perturbations.
- We demonstrate that compile-style generation with structured intermediate representations improves both execution accuracy and structural consistency.

2 Related Work

LLM Generation Instability The instability of LLM outputs has received increasing attention. Prior work shows that LLMs are sensitive to stochastic sampling and may produce inconsistent outputs under identical prompts (Wang et al., 2023), with early reasoning errors propagating to later steps (Zhang et al., 2024). Models may also over-deliberate during decoding, favoring incorrect solutions despite initially correct reasoning paths (Wang et al., 2025).

To improve robustness in program generation, prior approaches have explored schema linking, self-correction, and execution-based verification (Pourreza and Rafiei, 2023; Ni et al., 2023). However, self-correction remains unreliable without external feedback (Huang et al., 2024), and self-consistency methods require multiple sampling runs with high computational cost (Zhu et al., 2024; Zhou et al., 2026). Recent work also highlights structural or algorithmic variance across generations (Song et al., 2025; Rajput et al., 2025), motivating stability-oriented approaches such as instance-level randomization (Li et al., 2025).

However, existing studies primarily focus on improving prediction accuracy or reducing output variance, without explicitly analyzing the structural properties of generated programs. In contrast, our work evaluates the structural reliability of LLM-generated SQL queries through a structure-aware framework based on canonicalized AST representations, and further shows that structured intermediate representations can improve both structural stability and generation accuracy.

Evaluation for Program Generation Evaluating program generation remains challenging. Early approaches relied on surface-level metrics such as exact match or BLEU (Rajpurkar et al., 2016; Papineni et al., 2002), which are inadequate for program synthesis since semantically equivalent programs may differ in form (Kulal et al., 2019).

Modern benchmarks instead adopt execution-based evaluation. In Text-to-SQL, execution accuracy measures whether predicted queries return the same result as the reference (Zhong et al., 2020), while code generation benchmarks use unit tests (Chen et al., 2021; Austin et al., 2021). Although these metrics better capture functional correctness (Roziere et al., 2020), recent work shows that structurally different programs can produce identical outputs (Liu et al., 2023; Kim et al., 2025). This limitation motivates the need for structure-aware evaluation.

Program Equivalence and SQL Canonicalization A closely related line of work studies program equivalence and canonicalization, aiming to determine whether two programs are semantically identical despite syntactic differences. In the context of Text-to-SQL, prior work has explored query normalization, equivalence checking, and semantic parsing techniques to address spurious mismatches between predicted and reference queries (Zhong et al., 2020; Guo et al., 2019). However, these approaches primarily focus on improving evaluation fairness or matching correctness, rather than analyzing the variability of generated program structures across multiple generations. In contrast, our work leverages canonicalized representations not to collapse equivalent programs into a single label, but to explicitly quantify structural diversity and

instability in LLM outputs.

Structured and Constrained Generation Prior work has explored structured generation and constrained decoding to improve program validity. Methods such as Poesia et al. (2022) and Scholak et al. (2021) enforce structural constraints during decoding, while grammar-based approaches ensure adherence to predefined syntax rules (Geng et al., 2023; Raspanti et al., 2025).

However, these approaches focus on syntactic correctness and do not address structural variability across multiple valid generations (Raspanti et al., 2025; Albinhassan et al., 2025). More recent work incorporates logical constraints during decoding (Ma and Hu, 2025), yet structural variability remains unresolved. *In contrast, our work evaluates structural reliability post hoc and explores a compile-style generation paradigm that produces structured intermediate representations before compiling them into executable programs.*

3 SQLSTRUCTEVAL

In this section, we present SQLSTRUCTEVAL, a structure-aware framework for analyzing the structural reliability of LLM-generated programs. Given multiple generations for the same input, SQLSTRUCTEVAL represents programs as canonicalized ASTs and quantifies their structural consistency, diversity, and robustness. We also examine a compile-style generation paradigm that leverages structured intermediate representations to improve program reliability.

3.1 Problem Setup

We study the problem of program generation with LLMs, focusing on the Text-to-SQL task as a controlled and well-defined environment. Given a natural language question x and a database schema S, the goal is to generate an SQL query y that correctly answers the question.

Unlike conventional evaluation settings that consider only a single generated program, we analyze the structural behavior of LLMs across multiple generations. For each input question x, the model produces a set of candidate programs

$$Y = \{y_1, y_2, ..., y_k\}, \tag{1}$$

where k denotes the number of sampled outputs. Each query is converted into its corresponding AST representation, which captures the hierarchical structure of the program. These representations enable SQLSTRUCTEVAL to systematically analyze how consistently the model constructs programs across generations.

3.2 Structural Representation

We represent each generated SQL query as an abstract syntax tree (AST).[1] An AST encodes the hierarchical structure of a program, where nodes correspond to SQL operators such as `SELECT`, `WHERE`, `JOIN`, and `GROUP BY`, and edges represent syntactic relationships among these components.

Formally, for each generated query y_i, we obtain an AST representation

$$T_i = \text{AST}(y_i). \tag{2}$$

Compared with raw SQL text, AST representations abstract away superficial differences such as formatting or alias naming and focus on the underlying program structure. As a result, queries that are structurally equivalent but textually different can be mapped to comparable structural representations.

These representations enable SQLSTRUCTEVAL to compare programs generated for the same input and quantify their structural similarity, diversity, and robustness across generations.

3.3 Structural Evaluation Measures

Based on canonical AST representations, SQLSTRUCTEVAL introduces a set of evaluation measures to characterize the structural behavior of LLM-generated programs. These measures are computed over multiple generations for the same input and capture complementary aspects of model behavior, including structural consistency, diversity, alignment with reference programs, and robustness under input perturbations.

For an input question x, suppose the model generates N candidate programs $\{p_i\}_{i=1}^{N}$. After SQL parsing and canonicalization, each program is mapped to either a canonical AST structure $a_i \in \mathcal{A}$ or a parsing failure $a_i = \perp$. Let

$$\mathcal{I} = \{i \mid a_i \neq \perp\}, \qquad M = |\mathcal{I}| \tag{3}$$

denote the set of successfully parsed generations. Among these valid structures $\{a_i : i \in \mathcal{I}\}$, suppose there are K distinct canonical structures $\{s_k\}_{k=1}^{K}$. Let

[1] We parse the SQL queries into ASTs using the `sqlglot` library: https://sqlglot.com/.

$$n_k = \big|\{i \in \mathcal{I} : a_i = s_k\}\big|, \qquad p_k = \frac{n_k}{M}, \quad (4)$$

where n_k denotes the count of structure s_k and p_k its empirical frequency.

Structural consistency. To quantify how consistently the model produces the same program structure, we compute the majority-structure ratio

$$\mathrm{Cons}(x) = \max_{1 \le k \le K} p_k. \quad (5)$$

This quantity measures the proportion of generations that share the dominant AST structure. Higher values indicate stronger structural consistency. In the experiments, we also refer to this quantity as the *majority ratio*.

Structural diversity. We measure structural diversity using the number of distinct AST structures produced for the same input

$$\mathrm{Div}(x) = K. \quad (6)$$

This quantity corresponds to the *distinct structure count* reported in the experiments.

To further characterize the distribution of structures, we compute the entropy

$$\mathrm{H}(x) = -\sum_{k=1}^{K} p_k \log_2 p_k, \quad (7)$$

which captures how evenly the model distributes probability mass across alternative program structures.

Gold structure alignment. To measure alignment with the reference program, we compute the fraction of successfully parsed generations whose canonical AST matches that of the gold SQL query

$$\mathrm{Gold}(x) = \frac{1}{M} \sum_{i \in \mathcal{I}} \mathbf{1}[a_i = a^*] \quad (8)$$

where a^* denotes the canonical AST of the gold SQL query. This restriction ensures comparability within the structural space.

Structural robustness. To evaluate robustness under input perturbations, we consider an original input $x^{(0)}$ and T perturbed variants $\{x^{(t)}\}_{t=1}^{T}$, such as paraphrased questions. For each input $x^{(t)}$, we determine the majority AST structure

$$s_t^* = \arg\max_{s \in \mathcal{A}} n_{t,s}, \quad (9)$$

where $n_{t,s}$ denotes the count of structure s among generations for input $x^{(t)}$. We define cross-paraphrase structural agreement as

$$\mathrm{Cons}_{\mathrm{para}}(x) = \frac{2}{(T+1)T} \sum_{0 \le t_1 < t_2 \le T} \mathbf{1}\big[s_{t_1}^* = s_{t_2}^*\big], \quad (10)$$

which measures how consistently the model preserves program structure across semantically equivalent inputs.

We further define perturbation sensitivity as

$$\mathrm{Sens}(x) = \frac{1}{T} \sum_{t=1}^{T} \mathbf{1}\big[s_t^* \neq s_0^*\big], \quad (11)$$

which quantifies how often the majority program structure changes under input perturbations.

At the dataset level, we additionally report the *sensitive fraction*, defined as the proportion of inputs whose majority structure differs from that of the original input. Together, these evaluation measures form the core analysis components of the SQLStructEval framework and are used throughout the experimental analysis in Section 4.

3.4 Compile-Style Generation with Structured Intermediate Representations

Beyond analyzing structural reliability, we also investigate a compile-style generation paradigm that incorporates structured intermediate representations into the program generation process. Conventional LLM-based program generation typically follows a direct paradigm, where models translate natural language instructions directly into executable programs (Chen et al., 2021; Austin et al., 2021).

In contrast, compile-style generation introduces an explicit intermediate representation between natural language input and the final program. In this framework, the model first generates a structured program representation in the AST space, which is then deterministically compiled into the final executable SQL query. This design separates structural planning from surface-level code synthesis, enabling the model to explicitly construct program structure before producing executable code.

Compared with constrained decoding approaches that enforce syntactic constraints during token generation (Scholak et al., 2021; Poesia et al., 2022), compile-style generation treats program structure as an explicit intermediate artifact

rather than an implicit decoding constraint. By decoupling structural reasoning from final program synthesis, this paradigm can reduce structural variance across generations and potentially improve the reliability of generated programs.

In our experiments, we compare compile-style generation with conventional direct generation and evaluate their impact on both execution accuracy and structural stability.

4 Experiments and Evaluation

Our experiments investigate the structural reliability of SQL programs generated by LLMs and evaluate the proposed SQLSTRUCTEVAL framework. While most existing Text-to-SQL evaluations focus primarily on functional correctness (e.g., execution accuracy), we study an additional dimension: whether generated programs remain structurally stable across generations and under input perturbations.

To this end, we conduct four experiments. The first two experiments analyze structural variance and its relationship to execution-based evaluation across multiple LLM families. The third experiment studies whether generating SQL through a structured intermediate representation can improve generation accuracy. The fourth experiment evaluates the robustness of generated program structures under semantically equivalent input perturbations such as paraphrases and schema variations.

4.1 Experimental Setup

Dataset We conduct experiments on the **Spider** benchmark (Yu et al., 2018), a widely used cross-domain Text-to-SQL dataset containing natural language questions paired with SQL queries over multiple relational databases. We use the Spider development set, which contains 1,034 questions across 138 databases. Each generated query can be executed against the corresponding SQLite database to verify functional correctness.

Models and Generation We evaluate LLMs from multiple families, including GPT-4.1-mini and GPT-5-mini (OpenAI), Claude-4.5-Sonnet and Claude-4.5-Opus (Anthropic), Gemini-3-Pro and Gemini-2.5-Flash (Google), and DeepSeek-V3.1.

For each input question, the model is provided with the database schema (table names, column names, and foreign-key relations) and asked to generate an SQL query in the SQLite dialect. To analyze structural variance, we sample multiple SQL queries for the same input using stochastic decoding.

Model	Distinct↓	Majority↑	Entropy↓	Gold↑
GPT 5 mini	1.913	0.650	0.413	0.197
GPT 4.1 mini	1.620	0.793	0.301	0.253
Claude 4.5 Opus	0.779	0.687	0.049	0.391
Claude 4.5 Sonnet	1.241	0.775	0.195	0.356
DeepSeek V3.1	1.023	0.735	0.136	0.321
Gemini 3 Pro	0.845	0.595	0.133	0.369
Gemini 2.5 Flash	0.623	0.556	0.044	0.332

Table 1: Structural statistics of SQL queries generated by different LLMs on the Spider development set. For each question, we sample 10 SQL queries and parse them into canonical ASTs. **Distinct** denotes the average number of distinct AST structures per question, **Majority** is the average proportion of sampled generations that share the dominant AST structure for each question, **Entropy** measures the average entropy of the structure distribution, and **Gold** is the fraction of sampled generations whose canonical AST matches that of the gold SQL query.

Structural Analysis We parse generated SQL queries into ASTs using the `sqlglot` parser and canonicalize them to remove superficial differences such as alias names and formatting. These representations allow us to compare program structures independent of surface forms and analyze both functional correctness and structural properties. Evaluation measures are defined in Section 3.3.

4.2 Experiment 1: Structural Variance of Generated SQL

We begin by examining whether LLMs generate structurally consistent SQL queries when producing multiple solutions for the same input. While Text-to-SQL systems are typically evaluated using execution accuracy or exact match (Yu et al., 2018; Zhong et al., 2020; Deng et al., 2022), these metrics verify result correctness, but do not capture the stability of program structures across generations.

To analyze structural variance, we sample ten SQL queries for each Spider development example from each model using stochastic decoding, resulting in over 10,000 queries per model. We parse all queries into ASTs using `sqlglot` and canonicalize the resulting structures to remove superficial differences such as alias naming and formatting.

Table 1 shows substantial structural variance across all models. For example, the main model (`GPT-5-mini`) produces on average 1.91 distinct structures per question, while the majority structure appears only 65% of the time. Although stronger

models such as `Claude-4.5-Opus` exhibit more concentrated distributions, multiple structural variants still frequently occur for the same input. In extreme cases, up to 5–10 distinct structures can appear within ten generations.

These results indicate that structural variance is a common issue in LLM-based Text-to-SQL generation, raising concerns about the reliability of generated programs. Even with identical inputs and schemas, models often produce alternative query compositions, such as different join orders or nested query formulations. This motivates a closer examination of how structural variance affects evaluation metrics and generation strategies.

4.3 Experiment 2: Execution Accuracy vs. Structural Reliability

Experiment 1 showed that repeated sampling often produces multiple structural variants for the same input. This raises a key question: *Does execution accuracy adequately reflect the reliability of generated programs?*

Execution accuracy measures whether the generated query yields the same result as the gold query and is the dominant evaluation metric in Text-to-SQL benchmarks. However, it only captures functional correctness and does not reveal whether program structures are consistent. This experiment reuses the SQL generations from Experiment 1. For each example, we execute all generated queries on the corresponding SQLite database and compare the results with the gold query. Queries that return identical results are labeled as *execution-correct*. Structural statistics are then computed over both all generations and the subset of execution-correct queries using the canonical AST representations described in Section 3.

Table 2 shows that structural variance remains substantial even among execution-correct queries. For instance, `GPT-5-mini` achieves an execution accuracy of 0.74, yet execution-correct queries still contain on average 1.38 distinct AST structures per question and exhibit relatively low structural agreement (AST similarity 0.55). Similar patterns follow across model families. Moreover, a significant fraction of questions fall into the category of *execution-correct but structurally different*, where multiple queries produce the correct answer, but correspond to different program structures. Depending on the model, this occurs in roughly 20%–39% of questions. These findings reveal a systematic mismatch between execution accuracy and structural reliability: even when models frequently produce correct answers, the underlying program structures can remain unstable. Execution-based metrics therefore capture correctness, but fail to reflect structural consistency, motivating the structural evaluation framework proposed in SQLSTRUCTEVAL.

4.4 Experiment 3: Compile-style Generation with Structured Intermediate Representation

Experiments 1 and 2 reveal substantial structural variance in LLM-generated SQL queries, even when execution accuracy is relatively high. This raises the question of whether enforcing an explicit structural representation during generation can improve program reliability.

We compare three generation paradigms on the Spider development set (1,034 questions). In the **Direct SQL** setting, the model directly generates SQL queries from the natural language question and database schema. As an additional baseline, we include **DIN-SQL** (Pourreza and Rafiei, 2023), which uses decomposed in-context learning to guide SQL generation through multi-step prompting. In the **compile-style** setting, the model first generates a structured JSON representation corresponding to a SQL AST, which is then deterministically compiled into executable SQL. For all pipelines, we sample 10 outputs per question using the same model (GPT-5-mini) and identical decoding configurations.

Table 3 summarizes the results. Compile-style generation achieves the highest execution accuracy (0.7864), outperforming both direct SQL generation (0.7412) and DIN-SQL (0.7359). It also yields stronger structural consistency among execution-correct queries, with AST similarity increasing from 0.55 to 0.63. Meanwhile, compile-style generation produces more distinct structures across all generations. This suggests that the structured intermediate representation allows broader exploration of valid query structures while concentrating correct solutions around a dominant structural pattern. The structured generation pipeline itself remains highly reliable: nearly all outputs produce valid JSON representations, and approximately 96% of the exmples successfully complete the full pipeline from structured representation to executable SQL. Overall, these results indicate that generating SQL in an explicit structural space can improve both execution accuracy and structural reliability while preserving diversity in generated query structures.

Model	Exec Acc↑	Success Rate↑	Distinct (all)↓	Distinct (corr)↓	AST Sim (corr)↑	High-Acc Low-Struct↓	Exec-Corr Struct-Diff↓
GPT 5 mini	0.741	0.741	1.913	1.378	0.552	0.351	0.297
GPT 4.1 mini	0.754	0.757	1.620	1.129	0.656	0.250	0.163
Claude Sonnet 4.5	0.757	0.804	1.241	0.714	0.624	0.054	0.201
Claude Opus 4.5	0.816	0.816	0.779	0.650	0.600	0.038	0.224
DeepSeek V3.1	0.771	0.771	1.023	0.774	0.612	0.101	0.191
Gemini 2.5 Flash	0.638	0.743	0.623	0.509	0.484	0.016	0.191
Gemini 3 Pro	**0.842**	**0.843**	0.843	0.721	0.537	0.139	**0.337**

Table 2: Execution accuracy and structural reliability statistics of LLM-generated SQL on the Spider development set. For each question, 10 queries are sampled and executed. We report execution-based metrics (Exec Acc, Success Rate) and structure-based metrics (Distinct, AST Sim), along with two inconsistency indicators: High-Acc Low-Struct and Exec-Corr Struct-Diff.

Metric	Direct SQL	DIN-SQL	Compile-style
Execution Accuracy	0.742	0.736	**0.785**
AST Similarity (correct)	0.552	0.579	**0.632**
Distinct Structures (all)	1.908	1.553	2.527
JSON Valid Rate	–	–	0.998
Compilable Rate	–	–	0.971
SQL Parse Rate	–	–	0.902
End-to-End Success	–	–	0.959

Table 3: Performance comparison of three Text-to-SQL generation paradigms on the Spider development set. We report execution accuracy and structural consistency (AST Similarity, Distinct Structures). For the compile-style pipeline, additional intermediate validity metrics are included (JSON Valid, Compilable, SQL Parse, End-to-End Success).

Model	AST Sim (para)↑	Distinct↓	Sensitivity↓	Sensitive Frac↓
GPT 5 mini	0.328	13.220	0.622	0.900
GPT 4.1 mini	0.493	6.640	0.462	0.770
DeepSeek V3.1	0.655	3.330	0.280	0.515
Claude Sonnet 4.5	0.755	2.190	0.179	0.365
Claude Opus 4.5	0.755	2.190	0.179	0.365
Gemini 2.5 Flash	0.787	2.870	0.158	0.395
Gemini 3 Pro	**0.894**	**1.530**	**0.079**	**0.195**

Table 4: Structural robustness of SQL generation under paraphrase perturbations on the Spider development set. We measure cross-paraphrase structural consistency (AST Sim), diversity (Distinct), and sensitivity to input variations (Sensitivity, Sensitive Frac).

Model	AST Sim (schema)↑	Distinct↓	Sensitivity↓	Sensitive Frac↓
GPT 5 mini	0.490	7.330	0.500	0.640
GPT 4.1 mini	0.727	3.600	0.263	0.360
DeepSeek V3.1	0.808	1.870	0.170	0.260
Gemini 2.5 Flash	0.895	1.790	0.098	0.145
Claude Sonnet 4.5	0.955	1.130	0.043	0.065
Claude Opus 4.5	0.955	1.130	0.043	0.065
Gemini 3 Pro	**0.957**	**1.060**	**0.043**	**0.065**

Table 5: Structural robustness of SQL generation under schema presentation perturbations on the Spider development set. We evaluate cross-variant structural consistency (AST Sim), diversity (Distinct), and sensitivity to schema changes (Sensitivity, Sensitive Frac).

4.5 Experiment 4: Structural Robustness under Input Perturbations

We further ask whether the structural variance observed in the previous experiments is driven by genuine semantic ambiguity or by sensitivity to superficial input changes. To study this, we evaluate structural robustness under two types of semantically equivalent perturbations: natural language paraphrases and schema presentation changes.

We randomly select 200 questions from the Spider development set and construct multiple semantically equivalent variants for each question. For every variant and model, we generate SQL queries using the same repeated-sampling protocol as in previous experiments and analyze their canonical AST structures. We report cross-variant AST similarity, the number of distinct structures observed across variants, and perturbation sensitivity.

Paraphrase Perturbations For each selected question, we construct several paraphrases that preserve the original database semantics while varying wording and syntax. Table 4 reports the results.

Paraphrase perturbations frequently change the generated SQL structure. For the main model (`GPT-5-mini`), cross-paraphrase AST similarity is only 0.33, the average number of distinct structures exceeds 13, and nearly 90% of questions are structurally sensitive to paraphrasing. Stronger models are more robust, but even the best model (`Gemini 3 Pro`) still changes structure on about 20% of questions.

This suggests that the model's program construction is not invariant to semantically equivalent inputs, and may depend on superficial linguistic cues. In particular, different phrasings of the same question may trigger different decomposition strategies or join patterns, indicating instability in the underlying reasoning process.

Schema Presentation Perturbations We next keep the natural language question fixed and perturb only the schema presentation, for example by reordering tables or columns while preserving the same database semantics. Results are shown in Table 5.

Compared to paraphrase perturbations, schema perturbations lead to higher structural stability, suggesting that models are more sensitive to linguistic variation than to schema ordering. Models are generally more robust to schema perturbations than to paraphrases, but the effect is still substantial. For `GPT-5-mini`, cross-schema AST similarity is 0.49 and about 64% of questions remain structurally sensitive. Even strong models such as Claude and Gemini still exhibit non-negligible sensitivity despite much higher robustness.

Overall, Experiment 4 shows that structural instability is strongly influenced by surface-form sensitivity. Even when semantics remain unchanged, perturbing either the question wording or the schema presentation can trigger different SQL structures. These results suggest that structural instability is not merely a consequence of stochastic decoding, but is systematically influenced by surface-form variations. This reinforces the motivation for structural evaluation and for generation strategies that operate more explicitly in the structural space.

4.6 Discussion

Across our experiments, we observe that LLM-generated SQL queries exhibit substantial structural variability, even when execution accuracy is high. Multiple structurally distinct programs are often produced for the same input, including cases where all outputs are execution-correct.

This reveals a key limitation of execution-based evaluation: while it verifies output correctness, it does not capture how consistently models construct programs. As a result, execution accuracy can mask significant variation in underlying reasoning processes.

Our robustness analysis further shows that this structural variability is sensitive to superficial input changes, such as paraphrases and schema presentation. This suggests that current models may rely on surface-level patterns rather than stable structural mappings.

Finally, our results indicate that incorporating structured intermediate representations, as in compile-style generation, can partially reduce this variability and improve structural consistency.

Overall, these findings highlight the importance of evaluating structural reliability alongside functional correctness for a more complete understanding of LLM-based program generation. From a broader perspective, these results suggest that current LLMs lack a stable structural representation of programs, and instead construct programs in a context-sensitive manner that is vulnerable to input variation.

5 Conclusions and Future Work

This paper investigated the structural reliability of SQL programs generated by LLMs. While existing Text-to-SQL evaluations primarily focus on functional correctness, such as execution accuracy (Yu et al., 2018; Deng et al., 2022), our study showed that these metrics do not fully capture the stability of generated program structures.

We introduced STRUCTEVAL, a framework for analyzing the structural properties of generated SQL queries through canonical AST representations. Across a series of experiments on the Spider benchmark (Yu et al., 2018), we showed that LLM-generated SQL often exhibits substantial structural variance, even among execution-correct outputs, and that this instability is frequently triggered by superficial input variations such as paraphrases or schema presentation changes. These results suggest that structural instability is not merely a consequence of stochastic decoding, but is systematically influenced by surface-form variations.

Our results also suggest that explicitly generating programs in the structural space, as in compile-style generation with structured intermediate representations, can improve both execution performance and structural consistency, indicating a promising direction for more reliable program generation. This is particularly important for real-world applications, where consistent program behavior is critical for interpretability and system reliability.

More broadly, our findings highlight a gap between functional correctness and structural reliability, suggesting that current LLMs rely on context-sensitive generation strategies that are vulnerable to input variation.

Future work includes extending structural evaluation to other program generation tasks (e.g., general code synthesis and API calling), exploring stronger structural constraints during generation, and incorporating structural objectives into training. These findings highlight the importance of incorporating structural constraints or representations into both evaluation and generation pipelines.

Limitations

This study focuses on the Spider benchmark (Yu et al., 2018), which represents a widely used but relatively controlled Text-to-SQL setting. Structural behaviors observed in this work may differ in environments involving larger real-world databases or more complex query distributions. In addition, our analysis relies on canonical AST representations, which capture structural differences between SQL queries but do not fully account for semantic equivalence between different query formulations. Exploring richer semantic metrics and evaluating structural reliability across broader program generation tasks remain important directions for future work.

Ethical Considerations

This work studies the structural reliability of LLM-generated SQL queries using publicly available benchmarks. We conduct all experiments on the Spider dataset, which contains no personally identifiable or sensitive user information. Therefore, this work does not involve human subjects or private data.

A potential risk of LLM-based program generation is the production of incorrect or misleading queries, which could lead to erroneous downstream decisions if deployed in real-world systems. Our work aims to mitigate such risks by proposing structure-aware evaluation measures that better reveal instability and hidden failure modes beyond execution accuracy.

All experiments are conducted in an offline evaluation setting, and we do not deploy generated queries to real-world databases. We also acknowledge that different LLM providers may exhibit variability in outputs due to model updates or API changes, which may affect reproducibility. We encourage future work to consider standardized evaluation protocols and open-source implementations to improve transparency and reproducibility.

Data License Our experiments are conducted on the Spider dataset, which is publicly available for research purposes. We use the dataset in accordance with its original licensing terms.

The paraphrased inputs used in our robustness experiments are automatically generated transformations of the original Spider questions and do not introduce new proprietary content. All derived data, including canonicalized AST representations and generated SQL outputs, are produced by the models during evaluation.

We will release our code and evaluation scripts to facilitate reproducibility. Any released artifacts will comply with the licensing terms of the original datasets and model usage policies.

References

Mohammad Albinhassan, Pranava Madhyastha, Mark Law, and Alessandra Russo. 2025. Learning and enforcing context-sensitive control for LLMs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pages 834–842, Vienna, Austria. Association for Computational Linguistics.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Naihao Deng, Yulong Chen, and Yue Zhang. 2022. Recent advances in Text-to-SQL: A survey of what we have and what we expect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10932–10952, Singapore. Association for Computational Linguistics.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-SQL in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy. Association for Computational Linguistics.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large Language Models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Heegyu Kim, Jeon Taeyang, SeungHwan Choi, Seungtaek Choi, and Hyunsouk Cho. 2025. FLEX: Expert-level false-less EXecution metric for Text-to-SQL benchmark. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4448–4475, Albuquerque, New Mexico. Association for Computational Linguistics.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. SPoC: Search-based pseudocode to code. 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*, pages 11883–11894.

Yiyang Li, Yonghuang Wu, Ying Luo, Liangtai Sun, Zishu Qin, Lin Qiu, Xuezhi Cao, and Xunliang Cai. 2025. Instance-level randomization: Toward more stable LLM evaluations. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 3411–3425, Suzhou, China. Association for Computational Linguistics.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Franklin Ma and Alan J. Hu. 2025. Logically constrained decoding. In *Proceedings of The 3rd Workshop on Mathematical Natural Language Processing (MathNLP 2025)*, pages 150–167, Suzhou, China. Association for Computational Linguistics.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. Lever: learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of Text-to-SQL with self-correction. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the Text-to-SQL capabilities of large language models. *CoRR*, abs/2204.00498.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas. Association for Computational Linguistics.

Prateek Rajput, Abdoul Aziz Bonkoungou, Yewei Song, Abdoul Kader Kaboré, Iyiola E. Olatunji, Jacques Klein, and Tegawendé F. Bissyandé. 2025. Dynamic stability of LLM-Generated code. *CoRR*, abs/2511.07463.

Federico Raspanti, Tanir Ozcelebi, and Mike Holenderski. 2025. Grammar-constrained decoding makes large language models better logical parsers. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pages 485–499, Vienna, Austria. Association for Computational Linguistics.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yewei Song, Tiezhu Sun, Xunzhu Tang, Prateek Kumar Rajput, Tegawendé F. Bissyandé, and Jacques Klein. 2025. Measuring LLM code generation stability via structural entropy. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 3922–3926.

Kaishen Wang, Hengrui Gu, Meijun Gao, and Kaixiong Zhou. 2025. DAMO: decoding by accumulating activations momentum for mitigating hallucinations in vision-language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. 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*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Muru Zhang, Ofir Press, William Merrill, Alisa Liu, and Noah A. Smith. 2024. How language model hallucinations can snowball. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic evaluation for Text-to-SQL with distilled test suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 396–411, Online. Association for Computational Linguistics.

Yixi Zhou, Fan Zhang, Yu Chen, Haipeng Zhang, Preslav Nakov, and Zhuohan Xie. 2026. FinCARDS: Card-based analyst reranking for financial document question answering. *CoRR*, abs/2601.06992.

Jiace Zhu, Yingtao Shen, Jie Zhao, and An Zou. 2024. Path-Consistency: Prefix enhancement for efficient inference in LLM. *CoRR*, abs/2409.01281.

A Additional Error Analysis

To better understand the behavioral differences between traditional direct SQL generation and the proposed compile-style generation paradigm, we conduct an additional qualitative error analysis on representative examples from the Spider development set.

A.1 Error Taxonomy

Based on manual inspection of generated SQL queries, we categorize errors into six major types:

- **A. Schema Linking Errors.** Incorrect or missing references to tables, columns, or foreign-key relationships between tables.
- **B. Logical Form Errors.** Errors in high-level query logic such as aggregation, grouping, nested queries, or set operations.
- **C. Join Path Errors.** Incorrect or incomplete join paths between tables, including missing joins or incorrect join conditions.
- **D. Predicate / Constraint Errors.** Incorrect or missing filtering conditions in `WHERE` or `HAVING` clauses, such as incorrect comparison operators or incorrect attribute references.
- **E. Structural Variance but Executable.** Queries that produce correct execution results but differ substantially in structure from the gold query or other correct solutions.
- **F. Intermediate Representation / Compilation Errors** Errors introduced by the structured intermediate representation or the compilation process in the compile-style generation pipeline.

A.2 Error Distribution

Table 6 shows the distribution of error types observed in baseline and compile-style generation. Each ratio represents the proportion of errors belonging to the corresponding category.

As shown in Table 6, compile-style generation reduces several common semantic errors, including schema linking, logical form mistakes, incorrect join paths, and predicate errors. However, two categories increase: structurally diverse but executable queries (Type E) and intermediate representation or compilation-related failures (Type F). This pattern is consistent with the quantitative observations reported in Experiment 3.

Error Type	Baseline	Compile-style	Δ
A. Schema linking	0.20	0.13	-0.07
B. Logical form	0.18	0.15	-0.03
C. Join path	0.14	0.09	-0.05
D. Predicate / constraint	0.16	0.10	-0.06
E. Structural variance (exec OK)	0.28	0.40	+0.12
F. Intermediate representation / compile failure	0.04	0.13	+0.09

Table 6: Error type statistics for baseline and compile-style generation on the Spider development set. Each value represents the fraction of observed errors belonging to the corresponding category. **Baseline** refers to direct SQL generation, while **Compile-style** refers to generation using structured intermediate representations followed by deterministic compilation into SQL. Δ denotes the difference between the two settings (compile-style minus baseline). Positive values indicate that the error type becomes more frequent under compile-style generation.

A.3 Representative Case Studies

We further illustrate typical failure modes using representative examples from the Spider development set.

Case Group A: Baseline Execution Failures. In many cases, baseline generation produces incorrect SQL queries due to schema linking or join path errors. Compile-style generation often recovers the correct structure by explicitly modeling table relations in the intermediate representation.

For example, for the question:

> "Which model of the car has the minimum horsepower?"

the gold query joins `CAR_NAMES` and `CARS_DATA` to retrieve the corresponding model. Baseline generation frequently produces queries that omit this join and directly access the `CARS_DATA` table, resulting in incorrect results. In contrast, compile-style generation constructs the join structure explicitly in the intermediate representation, producing SQL queries that are structurally equivalent to the gold query.

Case Group B: Execution-Correct but Structurally Divergent Queries. We also observe cases where multiple generated queries produce identical execution results but differ substantially in their syntactic structure. For example, for the question:

> "Show all countries and the number of singers in each country."

baseline generation often produces multiple structurally distinct SQL queries, including variants with redundant subqueries or unnecessary

DISTINCT operators. Although these queries produce identical results, they correspond to different AST structures and therefore increase structural entropy. Compile-style generation tends to produce a more consistent structural form, typically converging to a simple GROUP BY aggregation.

Case Group C: Paraphrase Sensitivity. Semantically equivalent questions can trigger different query structures in baseline generation. For example, the two questions

> "Show all countries and the number of singers in each country"
> "How many singers are from each country?"

share the same semantic intent and gold SQL query. However, baseline generation often produces different structural patterns for the two inputs, including variations involving subqueries or derived tables. Compile-style generation demonstrates improved structural consistency under paraphrasing, although some degree of variance remains.

Case Group D: Compile-style Failure Cases. Despite the overall improvements, compile-style generation introduces new failure modes related to the intermediate representation and compilation process. For instance, when answering the question

> "How many car models are produced in the USA?"

the correct query requires joining MODEL_LIST, CAR_MAKERS, and COUNTRIES. In some compile-style generations, the intermediate representation omits the final join to COUNTRIES and instead applies the filtering condition directly to the CAR_MAKERS table. This leads to execution errors despite the structured generation pipeline.

Overall, these case studies highlight complementary strengths and limitations of the two paradigms. Compile-style generation reduces many semantic reasoning errors but may introduce new failure modes related to the intermediate representation, suggesting opportunities for future improvements in intermediate representation design and compilation strategies.

B Implementation Details

This section describes the implementation details of StructEval, including SQL parsing and canonicalization, canonical AST representation, execution evaluation, and generation settings.

B.1 SQL Parsing and Canonicalization

We parse all generated SQL queries using the sqlglot parser, which provides robust SQL parsing and AST construction across multiple SQL dialects. In our experiments we use the SQLite dialect to ensure compatibility with the Spider benchmark databases.

To remove surface-level differences that are irrelevant to program structure, we apply both text-level normalization and AST-level canonicalization.

Text-level normalization. After rendering the parsed AST back into SQL text, we perform lightweight normalization including: (1) removing trailing semicolons and leading/trailing whitespace, (2) replacing line breaks with spaces and collapsing redundant whitespace, (3) removing simple column aliases (e.g., SELECT col AS x), and (4) converting the SQL string to lowercase.

AST-level canonicalization. To ensure that structurally equivalent queries share the same representation, we further canonicalize the AST structure:

- **Alias normalization.** Table aliases are renamed according to their first appearance order (e.g., t1, t2, ...). All column references are updated accordingly so that structurally identical joins remain equivalent under different alias choices.
- **Logical operator normalization.** Nested AND predicates are flattened into condition lists, sorted by their SQL string representation, and reconstructed into canonical AST nodes. This ensures that logically equivalent expressions such as A AND B and B AND A share the same canonical form.

These normalization steps are shared across all experiments (Exp1–Exp4) and applied to both baseline and compile-style generated queries.

B.2 Canonical AST Representation

For each SQL query we obtain a canonical representation through the following pipeline:

1. Parse the SQL query into an AST using sqlglot;

2. Apply AST-level canonicalization (alias normalization and logical operator normalization);

3. Render the canonical AST back into SQL;

4. Apply text-level normalization.

The resulting canonical SQL string uniquely corresponds to a canonical AST structure and is used as the basis for all structural metrics in our analysis.

For each question with k generated SQL queries, we compute the frequency distribution over canonical structures. Structural statistics are then derived from this distribution:

- **Distinct structures:** the number of unique canonical SQL representations.
- **Majority structure ratio:** the relative frequency of the most common structure among the k generations.
- **Structural entropy:** the Shannon entropy of the structure distribution, measuring structural diversity.

To compare generated structures with ground-truth queries, the gold SQL is canonicalized using the same pipeline, and structural matches are determined by canonical representation equality.

B.3 Execution Environment

Execution-based evaluation follows the standard Spider evaluation protocol. All experiments use the official Spider SQLite database files.

For each generated SQL query, we execute the query against the corresponding SQLite database and obtain the result set. A generated query is considered execution-correct if its execution result exactly matches the result produced by the gold SQL query. Queries that produce execution errors or mismatched results are considered incorrect.

B.4 Generation Settings

For each question we generate multiple candidate SQL queries from each model to analyze structural variation.

Sampling configuration. We sample $k = 10$ SQL queries for every $(question, schema)$ pair. For APIs that cannot return multiple samples in a single request, we perform multiple independent calls and aggregate the generated outputs.

Decoding parameters. Unless restricted by the model API, we use stochastic decoding with temperature $= 1.0$ and default nucleus sampling parameters. Beam search or greedy decoding is not used, as our goal is to analyze structural variation under realistic sampling conditions.

Compile-style generation. In Experiment 3, the compile-style setting requires models to generate a structured intermediate representation instead of directly producing SQL text. The structured representation is expressed as JSON and contains fields such as `select`, `from`, `joins`, `where`, `group_by`, `having`, `order_by`, and `limit`. A deterministic compiler converts the JSON representation into executable SQL. The compiled SQL is then evaluated using the same canonicalization and execution evaluation pipeline as the baseline generation.

C Prompt and Generation Settings

This section provides the prompt templates used in our experiments.

C.1 Prompt for Direct SQL Generation

In the baseline setting, models are prompted to directly generate a SQL query given a natural language question and the corresponding database schema.

```
You are an expert SQL generator.
Use SQLite dialect.
Only output ONE SQL query, no explanation.

Database ID: <db_id>

Database schema (JSON):
<schema_json>

Question:
<natural_language_question>

SQL:
```

C.2 Prompt for Compile-style JSON Generation

In the compile-style setting, the model is instructed to generate a structured JSON representation of the SQL query rather than directly producing SQL text. The prompt used for this setting is shown below.

```
You are an expert Text-to-SQL system for the
    Spider benchmark.
Your task is to write a structured JSON
    representation of a SQL query
for the given question and database schema.

Requirements:
```

```
- Use ONLY tables and columns that exist in the
     provided schema.
- Assume the database uses the SQLite dialect.
- You MUST output a single JSON object, and
    nothing else (no explanations).
- The JSON must describe the logical structure
    of the SQL query with the following fields:
  - type: "query"
  - query: {
    select: [ ... ],
    from: { ... },
    joins: [ ... ],
    where: [ ... ],
    group_by: [ ... ],
    having: [ ... ],
    order_by: [ ... ],
    limit: ...,
    distinct: ...
  }
- Do NOT include any natural language text in
    the JSON.

Database ID: {db_id}

Database schema (JSON):
{schema_json}

Question:
{question}

Now output ONLY the JSON object for the query
    structure:
```

D Additional Structural Examples

To complement the quantitative structural metrics reported in the main experiments, we present several representative examples illustrating common forms of structural variation in LLM-generated SQL queries.

Example 1: Execution-correct but structurally different queries **Question.** Which model of the car has the minimum horsepower? (`db_id = car_1`)

Gold SQL.

```
SELECT T1.Model
FROM CAR_NAMES AS T1
JOIN CARS_DATA AS T2 ON T1.MakeId = T2.Id
ORDER BY T2.Horsepower ASC
LIMIT 1;
```

Generated SQL (baseline).

```
SELECT DISTINCT cn.Model
FROM cars_data cd
JOIN car_names cn ON cd.Id = cn.MakeId
WHERE cd.Horsepower = (
  SELECT MIN(Horsepower) FROM cars_data
);
```

Although both queries return identical execution results, their canonical AST structures differ substantially, as illustrated in Figure 2. The gold query computes the minimum value using an `ORDER BY` + `LIMIT` structure, while the generated query uses an aggregate subquery with `MIN(Horsepower)` and an equality predicate. This introduces a nested subquery node in the AST, resulting in a significantly different structural representation.

Example 2: Structural sensitivity to paraphrase Consider the following pair of paraphrased questions from the Spider dataset (`db_id = concert_singer`):

Question A. Show all countries and the number of singers in each country.

Question B. How many singers are from each country?

Both questions correspond to the same gold SQL query:

```
SELECT country, COUNT(*)
FROM singer
GROUP BY country;
```

However, the baseline model produces slightly different structural variants:

```
-- Variant A
SELECT Country, COUNT(Singer_ID)
FROM singer
GROUP BY Country;

-- Variant B
SELECT Country, COUNT(*)
FROM singer
GROUP BY Country;
```

Although both queries are execution-equivalent, their AST structures differ because the aggregation argument differs (`COUNT(*)` vs. `COUNT(Singer_ID)`). Across multiple generations, additional variations such as derived tables or `DISTINCT` operators may also appear. This example illustrates how small linguistic changes in the question can lead to different structural query plans.

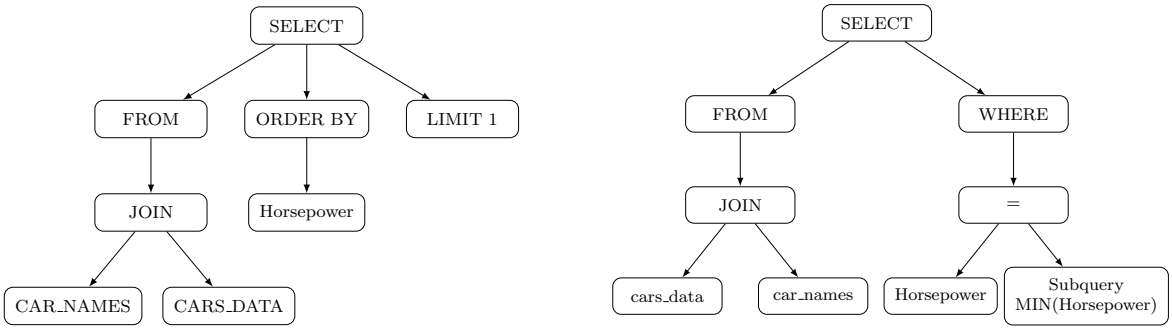
Example 3: Compile-style correcting structural errors **Question.** What are the names and ids of all countries with at least one car maker? (`db_id = car_1`)

Gold SQL.

```
SELECT T1.CountryName, T1.CountryId
FROM COUNTRIES AS T1
JOIN CAR_MAKERS AS T2 ON T1.CountryId = T2.
    Country
GROUP BY T1.CountryId
HAVING COUNT(*) >= 1;
```

Baseline SQL (typical failure).

```
SELECT CountryName, CountryId
FROM COUNTRIES
JOIN CAR_MAKERS
```



(a) AST of gold SQL query

(b) AST of generated SQL query

Figure 2: AST structures of two execution-equivalent SQL queries in Example 1 of Appendix D.

```
ON COUNTRIES.CountryId = CAR_MAKERS.Country;
```

The baseline model frequently omits the `GROUP BY` and `HAVING` clauses, producing incorrect query results.

Compile-style SQL.

```
SELECT COUNTRIES.CountryName, COUNTRIES.
    CountryId
FROM COUNTRIES
JOIN CAR_MAKERS
ON COUNTRIES.CountryId = CAR_MAKERS.Country
GROUP BY COUNTRIES.CountryId
HAVING COUNT(*) >= 1;
```

In the compile-style generation setting, the structured intermediate representation explicitly models aggregation components such as `group_by` and `having`. As a result, the compiled SQL consistently contains the correct aggregation structure and closely matches the gold AST.

E Example of the Compile-style Generation Pipeline

To illustrate the compile-style generation process used in Experiment 3, we present a concrete example showing the full pipeline from natural language to structured intermediate representation, internal AST, and final SQL query.

E.1 Input Natural Language Query

Question.

> Show the name and capacity of the stadium with the largest average attendance.

Database schema (simplified).

```
{
  "tables": [
    {"table_name": "stadium",
     "columns": ["Stadium_ID", "Name", "
    Location", "Capacity", "Average"]},
    {"table_name": "concert",
     "columns": ["concert_ID", "Stadium_ID", "
    Year"]}
  ],
  "foreign_keys": [
    {"source_table": "concert", "source_column":
     "Stadium_ID",
     "target_table": "stadium", "target_column":
     "Stadium_ID"}
  ]
}
```

E.2 LLM Output: Structured JSON Intermediate Representation

In compile-style generation, the model does not directly produce SQL text. Instead, it generates a structured JSON representation that encodes the program structure.

```
{
  "type": "query",
  "query": {
    "select": [
      {"expr": {"col": ["stadium", "Name"]}, "
    alias": null},
      {"expr": {"col": ["stadium", "Capacity"]},
     "alias": null}
    ],
    "from": {"table": "stadium", "alias": null},

    "joins": [],
    "where": [],
    "group_by": [],
    "having": [],
    "order_by": [
      {"expr": {"col": ["stadium", "Average"]},
     "direction": "desc"}
    ],
    "limit": 1,
    "distinct": false
  }
}
```

This JSON structure explicitly represents the logical components of the query, including `select`, `from`, `order_by`, and `limit`.

E.3 Intermediate AST Representation

The compiler then converts the JSON structure into an internal AST:

```
Select(
  columns = [
    Column(table="stadium", name="Name"),
    Column(table="stadium", name="Capacity")
  ],
  from = Table(name="stadium"),
  joins = [],
  where = [],
  group_by = [],
  having = [],
  order_by = [
    OrderBy(
      expr = Column(table="stadium", name="
     Average"),
      direction = DESC
    )
  ],
  limit = 1
)
```

This AST provides a canonical structural representation of the query, which can be analyzed and compared across different generations.

E.4 Compiled SQL Query

Finally, the AST is deterministically compiled into an executable SQL query:

```
SELECT stadium.Name, stadium.Capacity
FROM stadium
ORDER BY stadium.Average DESC
LIMIT 1;
```

This compiled SQL query is then used for execution evaluation and structural analysis in our experiments.