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

Yixi Zhou[1,2*] **Fan Zhang**[3*] **Zhiqiao Guo**[2*] **Yu Chen**[3†]
Haipeng Zhang[2†] **Preslav Nakov**[4] **Zhuohan Xie**[4]
[1]Hong Kong Baptist University [2]ShanghaiTech University
[3]The University of Tokyo [4]MBZUAI
yxzhou@comp.hkbu.edu.hk, zhang-fan@g.ecc.u-tokyo.ac.jp
chen@edu.k.u-tokyo.ac.jp, zhanghp@shanghaitech.edu.cn
{preslav.nakov,zhuohan.xie}@mbzuai.ac.ae

Abstract

In Text-to-SQL tasks, large language models can generate structurally different SQL queries that return correct answers for the same intent. We call this phenomenon *execution-correct structural divergence* (*ECSD*). We introduce **SQLStructEval**, a framework that analyzes this behavior through canonical abstract syntax tree representations. We quantify structural diversity and agreement across generations. Experiments with different LLMs on multiple Text-to-SQL datasets, including Spider, document *ECSD* across models and datasets. Furthermore, our experiments demonstrate that generated queries are sensitive to question paraphrases and schema presentation. To address *ECSD*, we adopt a pipeline that first generates structured intermediate representations and then deterministically compiles them into SQL, improving execution accuracy and structural agreement among correct outputs. Structural analysis thus provides an additional diagnostic perspective that complements execution-based evaluation. Code is available at GitHub.

1 Introduction

Large language models (LLMs) support program generation from natural language, including code synthesis and Text-to-SQL translation (Chen et al., 2021; Austin et al., 2021; Rajkumar et al., 2022). Evaluation commonly focuses on functional correctness: unit tests check generated code, and execution accuracy checks whether SQL produces the reference result on a database (Yu et al., 2018; Zhong et al., 2020). These tests are essential for evaluating whether a program answers the intended question. They leave open a different question: how consistently does a model construct that program across repeated generations and equivalent presentations of the input?

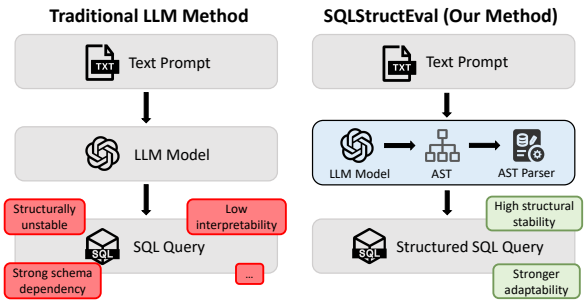


Figure 1: Motivation for **SQLStructEval**. Repeated Text-to-SQL generation can produce different query constructions even when they yield the same execution result. An AST-based view makes these structural differences explicit, supporting comparison of their diversity, stability, and sensitivity to input variations alongside conventional execution accuracy.

Programs that return the same result can use different joins, nested subqueries, aggregation arguments, or redundant operators. Comparing these constructions reveals how a model answers the question. Some alternatives remain equivalent across databases; others produce matching results on the evaluated instance and diverge as the data changes. A model can also change its preferred formulation when the question is reworded or the schema elements are reordered. We study these patterns across repeated generations. We call the occurrence of at least two queries that match the reference result for the same natural language intent but have different canonical ASTs *execution-correct structural divergence* (*ECSD*).

We introduce **SQLStructEval** to summarize canonical SQL structures across repeated generations. It measures concentration, diversity, reference alignment, and agreement under input perturbations. Figure 1 illustrates this structural perspective. We analyze execution-correct outputs separately to characterize *ECSD* and include a compile-style generation case study.

[*]Equal contribution.
[†]Corresponding authors.

Findings of the Association for Computational Linguistics: IJCNLP-AACL 2026, pages 2404–2422
November 6-10, 2026 ©2026 Association for Computational Linguistics

Structural reliability describes how consistently a model constructs queries across repeated generations and equivalent inputs. Examining these constructions alongside execution results reveals recurring choices and alternatives for further semantic inspection.

In experiments with seven LLMs on Spider, we observe *ECSD* across model families. We also find that perturbing the natural language expression of a user's intent can change the generated structure. To examine the scope of *ECSD*, we conduct experiments on three subsets of 100 examples each, which also demonstrate its presence. Finally, a compile-style pipeline improves execution accuracy and agreement among correct outputs in our setting. Our contributions are as follows:

- We document *ECSD* in LLM-generated SQL, showing how correct answers arise from diverse structures.
- We introduce **SQLStructEval**, a framework measuring structural diversity and sensitivity using canonical ASTs.
- We document *ECSD* across model families and additional dataset subsets, and evaluate compile-style generation as a full-pipeline mitigation case study.

2 Related Work

Variation in model generations. Self-consistency aggregates multiple sampled reasoning paths (Wang et al., 2023); related work studies the propagation of early errors (Zhang et al., 2024), and limits of self-correction (Huang et al., 2024). Instance-level randomization addresses variation in evaluation scores arising from random experimental factors (Li et al., 2025).

Program structure has also received direct attention. Song et al. (2025b) analyze code-generation stability through AST-based structural entropy, while Rajput et al. (2025) study algorithmic and runtime variation through opcode distributions. We connect SQL structure with execution correctness and input robustness.

Execution and semantic evaluation. Text-to-SQL evaluation distinguishes generalization to new query patterns from robustness to reformulations (Finegan-Dollak et al., 2018); cross-database tests reveal further transfer challenges (Suhr et al., 2020).

Benchmarks and surveys document execution and reference-based evaluation (Yu et al., 2018; Deng et al., 2022). Stronger tests expose errors missed by original tests (Liu et al., 2023). Distilled SQL test suites probe equivalence across database instances (Zhong et al., 2020), while FLEX studies false execution-based judgments (Kim et al., 2025). These approaches assess correctness; **SQLStructEval** characterizes *ECSD* alongside broader structural variation.

Robustness benchmarks. Spider-Syn tests synonym substitutions (Gan et al., 2021); Dr.Spider perturbs questions, databases, and SQL (Chang et al., 2023). ADVETA examines adversarial table perturbations (Pi et al., 2022). BIRD emphasizes database-grounded queries (Li et al., 2023), while Spider 2.0 targets enterprise workflows involving various dialects and metadata (Lei et al., 2025).

Structured program generation. IRNet generates SemQL before constructing SQL (Guo et al., 2019); RAT-SQL models relations for schema encoding and linking (Wang et al., 2020). DIN-SQL decomposes prompting (Pourreza and Rafiei, 2023); DAIL-SQL studies prompt representations and example selection (Gao et al., 2024). LEVER verifies language-to-code candidates using execution (Ni et al., 2023). Synchromesh and PICARD constrain generation using program structure or incremental parsing (Poesia et al., 2022; Scholak et al., 2021). Grammar-constrained decoding further supports structured outputs and logical parsing (Geng et al., 2023; Raspanti et al., 2025); other methods use context-sensitive or logical constraints (Albinhassan et al., 2025; Ma and Hu, 2025).

For example, PTD-SQL uses query-type-specific demonstrations (Luo et al., 2024); SENSE systematically learns from strong-model examples and weak-model errors (Yang et al., 2024). Meanwhile, CHASE-SQL combines diverse reasoning paths with pairwise candidate selection (Pourreza et al., 2025). These prominent methods directly connect candidate generation diversity with final answer quality. Our evaluation framework follows this connection through the SQL constructions themselves: which specific choices recur, which produce correct answers, and which change when the natural language input is rephrased. This provides a unified shared view for examining outputs from direct generation and complex pipelines that first systematically organize query components.

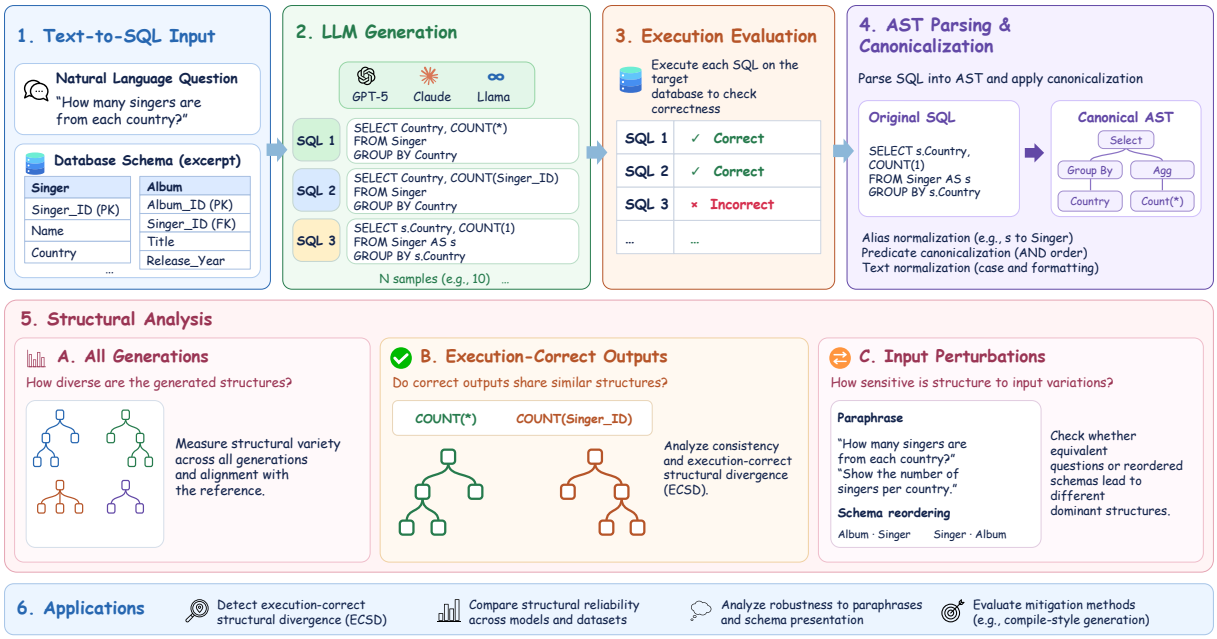


Figure 2: Overview of the **SQLStructEval** evaluation pipeline. Repeated SQL generations for a question and schema undergo execution checks and structural canonicalization. We compare structures across all parsed outputs, within the execution-correct subset, and across input variants. These views connect answer accuracy with structural diversity, agreement, and perturbation sensitivity.

3 SQLStructEval

Figure 2 brings together the two assessment paths and the structural comparisons in this section.

3.1 Problem Setup

Given a natural language question x and database schema S, a Text-to-SQL model generates a query intended to answer x. We sample N candidate queries $\{y_i\}_{i=1}^{N}$ for the same input. Repeated generation lets us follow both the answers a model produces and the query constructions it chooses. For example, when asked to count singers by country, a model may generate a grouped query using `COUNT(*)` in one attempt and `COUNT(Singer_ID)` in another. Both can return the reference result on the evaluated database, while expressing the counting operation differently.

We assess each candidate from two perspectives. The first asks whether the query answers the question correctly on the given database. We execute the candidate and compare its result with the annotated reference result. Repeating this check across the N candidates tells us how frequently the model answers correctly and identifies the correct outputs for further analysis.

The second asks how the model constructs its answers. We convert each query into a common structural representation, described in Section 3.2, and group candidates that share this representation.

We first examine all parsed outputs to see the range of constructions the model produces. We then examine the execution-correct subset to see whether correct answers come from a recurring construction or several alternatives. This second view is where we identify *ECSD*. Section 3.3 develops the measures for these comparisons and extends them to question paraphrases and schema reordering. Together, the two assessments connect answer accuracy with the variety and stability of the queries that produce those answers.

3.2 Structural Representation

Comparing query constructions begins with a common way to represent them. Two SQL strings can differ in spacing or alias names while organizing the same operations in the same way. We standardize these details so that repeated constructions can be grouped together. At the same time, we retain choices such as introducing a subquery or changing an aggregation argument, which help explain how the model builds an answer.

An abstract syntax tree (AST) represents this organization through operators such as `SELECT`, `JOIN`, `WHERE`, and `GROUP BY`, together with their arguments and relationships. For singer counts, the tree records the grouping column as well as the `COUNT` argument, while preserving the two counting expressions.

We parse SQL using `sqlglot` in the SQLite dialect. We then rename table aliases by first appearance, update column references, and flatten and sort conjunctions. For example, reordering the conditions in an `AND` expression is standardized by this step. Finally, we render the AST into SQL and normalize whitespace, case, trailing semicolons, and simple column aliases. Appendix D gives the implementation details.

The resulting canonical SQL string is our comparison representation: two queries share a structure when their normalized strings match. We can therefore group repeated constructions and count how often each appears. For every generation paradigm, this procedure applies to the final SQL, including SQL compiled from JSON.

3.3 Structural Evaluation Measures

We organize the measures around a sequence of questions: which constructions recur, how they relate to the reference query, which ones produce correct answers, and how they change when the input is rephrased or reordered. Throughout, the basic unit of analysis is the collection of queries generated for one question.

For a question x, parsing maps generation i to a canonical structure $a_i \in \mathcal{A}$ or a failure $a_i = \perp$. The valid index set $\mathcal{I}_x = \{i : a_i \neq \perp\}$ contains the outputs available for structural comparison, and $M_x = |\mathcal{I}_x|$ is their number. Among them, let $s_1, \ldots, s_K$ be the distinct structures, with counts n_k and relative frequencies $p_k = n_k/M_x$. These frequencies describe how the sampled outputs are distributed across query constructions.

Concentration and diversity. We begin by asking whether the model repeatedly returns to one construction. The majority-structure ratio (m) is the share of parsed outputs belonging to the most frequent structure:

$$m(x) = \max_{1 \leq k \leq K} p_k. \tag{1}$$

A high m indicates that one construction dominates the sample. To describe the alternatives around it, we also count the distinct structures (D) and calculate structural entropy (H):

$$D(x) = K, \qquad H(x) = -\sum_{k=1}^{K} p_k \log_2 p_k. \tag{2}$$

D counts the constructions that appear, while H describes how evenly outputs are spread across them. Together with m, they distinguish an occasional alternative from several equally frequent constructions.

We further report normalized entropy ($\widehat{H}$) as $\widehat{H}(x) = H(x)/\log_2 K$ when $K > 1$, and zero when $K \leq 1$. Dividing by $\log_2 K$ expresses entropy relative to the maximum for the observed number of structures; $\widehat{H}$ reaches one when those structures occur equally often. For execution-correct subsets, we report unnormalized entropy (H_c).

Annotated-reference alignment. Having described the model's recurring constructions, we next compare them with the annotated query. Reference alignment (r) is the fraction of parsed outputs that match its canonical representation a^*:

$$r(x) = \frac{1}{M_x} \sum_{i \in \mathcal{I}_x} \mathbf{1}[a_i = a^*]. \tag{3}$$

r tells us how often the model follows the reference construction. In the singer-count example, consistently correct queries using a different counting argument would yield high execution accuracy and low r, revealing a successful alternative that the model repeatedly chooses.

Execution-correct structural agreement. We now focus on the outputs that answer the question correctly. This connects the structural distribution directly to *ECSD*: how much variety remains among correct answers? Let E_x contain all execution-correct generation indices and let $C_x = E_x \cap \mathcal{I}_x$ contain those with a parsed structure. For a nonempty selected subset $B_x \subseteq \mathcal{I}_x$, define

$$c(B_x) = \frac{\max_{s \in \mathcal{A}} |\{i \in B_x : a_i = s\}|}{|B_x|}. \tag{4}$$

Correct-output agreement (c) applies this measure to C_x: it is the share of parsed correct outputs using the most frequent structure. The correct-output distinct count (D_c) counts structures in the same subset, and H_c summarizes their frequency distribution. We observe *ECSD* for a question when $D_c > 1$.

Comparing correct outputs with the full sample shows whether structural variety occurs among correct answers, incorrect answers, or both. Empty correct subsets receive zero for c, D_c, and H_c and remain in the dataset average.

Diagnostic mismatch indicators. To locate these patterns across a dataset, we report two question-level diagnostic rates. Let $e(x)$ be the fraction of the N generations whose execution result matches the reference. The low-agreement rate (ℓ) is the fraction of questions with $|E_x| \geq 2$ and $\widetilde{c}(C_x) < 0.7$, where $\widetilde{c}$ equals c for nonempty subsets and zero otherwise. It highlights questions with dispersed correct structures and also includes cases where all correct outputs fail structural parsing. The high-accuracy diversity rate (h) is the fraction with $e(x) \geq 0.8$ and $D_c > 1$. It locates *ECSD* among questions the model usually answers correctly.

We choose 0.7 and 0.8 as diagnostic thresholds. With ten generations, h requires at least eight correct outputs spanning multiple parsed structures. For nonempty correct subsets, ℓ selects cases where the most frequent structure accounts for less than 70%. Together with c and D_c, these rates show how widely correct-answer diversity and low agreement occur.

Robustness under perturbations. So far, the input has stayed fixed. We next ask whether the model retains its preferred construction when the question is paraphrased or the schema reordered. We generate queries for each input presentation and compare their most frequent structures.

For an original input $x^{(0)}$ and T variants $x^{(1)}, \ldots, x^{(T)}$, let s_t^* denote the majority structure for input t, with $t = 0$ denoting the original. Cross-variant agreement (v) is the fraction of input pairs that share a majority structure:

$$v(x) = \frac{2}{T(T+1)} \sum_{0 \leq u < w \leq T} \mathbf{1}[s_u^* = s_w^*]. \quad (5)$$

Perturbation sensitivity (σ) takes the original input as the reference and measures the fraction of variants whose majority changes:

$$\sigma(x) = \frac{1}{T} \sum_{t=1}^{T} \mathbf{1}[s_t^* \neq s_0^*]. \quad (6)$$

The sensitive fraction (f) is the proportion of questions with at least one changed majority. The pooled distinct count (D_p) counts unique structures across all variants. Together, these measures describe how often the preferred construction changes, how many questions are affected, and the total variety encountered.

Execution, coverage, and reporting. We read structural measures alongside execution and parsing results. Execution accuracy (e) is expressed as $e(x) = |E_x|/N$, the share of all generations that answer correctly. Conditional execution accuracy (e_s) divides $|E_x|$ by the number of outputs that execute successfully, using zero when all executions fail. It shows how often a query returns the right answer once it can run. The SQL parse rate (ρ) is M_x/N, recording how much of the generated sample is available for structural comparison. These concrete metrics directly provide a comprehensive overview of performance.

We calculate each statistic per question and then average across the dataset, giving every question equal weight. Empty structural subsets contribute zero to counts, concentration, and entropy; reference alignment is also zero when the generated structural subset is empty. Thus, the reported c and m averages reflect both structural coverage and concentration. Appendix A summarizes these conventions.

3.4 Compile-Style Generation

The measures above describe how query constructions vary. We also examine a procedure that makes these constructions explicit before producing SQL. The model fills a JSON representation with selection, tables, joins, predicates, grouping, ordering, and limits; a deterministic compiler converts these components into executable SQL. Appendix G illustrates the sequence from question to JSON, internal AST, and SQL.

For the singer-count question, the model specifies the country column, a counting expression, and the grouping field. The compiler assembles these choices into a SQL query. This intermediate step makes the selected operations and their arguments available for inspection before execution. Across repeated generations, we can then follow which component choices recur and which lead to alternative constructions. The resulting SQL connects these choices back to the evaluation: execution identifies successful answers, while canonicalization groups the constructions used to produce them. Consequently, this structural tracking offers valuable, deeper insights into model behavior.

We evaluate compiled and directly generated SQL with the same execution checks and canonicalization. Experiment 3 examines how the pipeline affects accuracy and structural variety among correct answers.

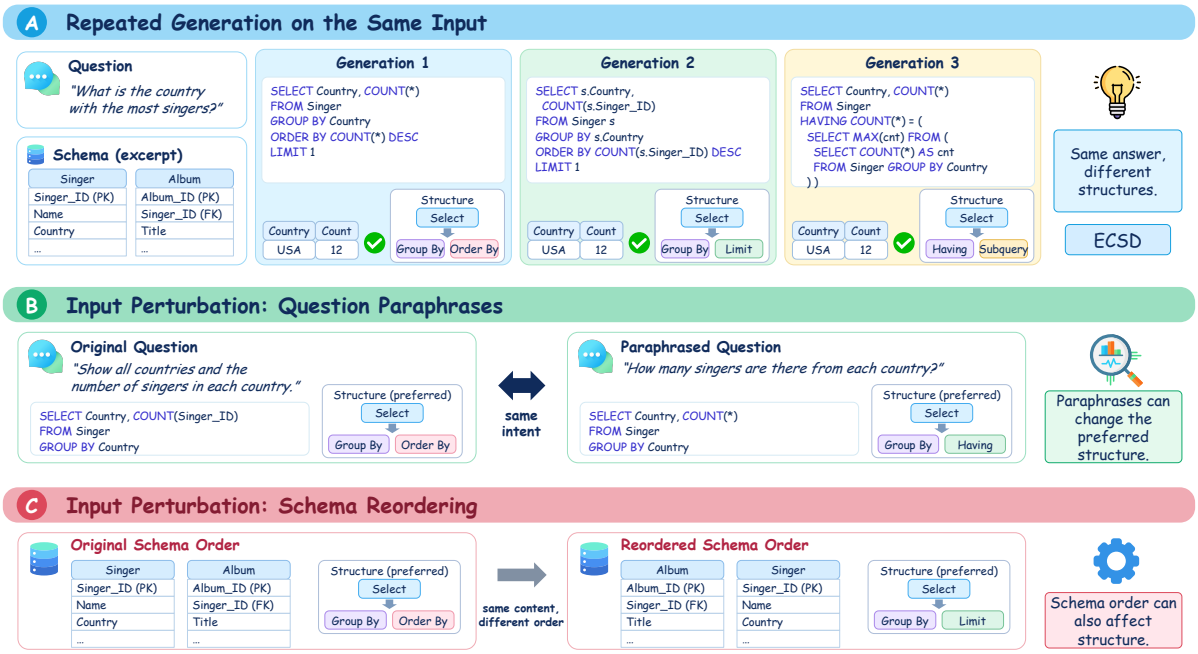


Figure 3: Structural variation examined in our experiments. (A) Repeated generation for a fixed input can produce different query constructions with matching answers on the evaluated database. (B) Question paraphrases and (C) schema reordering can change the preferred construction. The examples illustrate the comparisons used to study repeated-generation diversity and input sensitivity.

4 Experiments and Evaluation

Figure 3 illustrates the structural changes we follow across repeated generations and input variants. We now use the two perspectives from Section 3.1 to follow how models construct answers in practice. The experiments proceed in four steps. We first hold the input fixed and examine the structures that recur across generations. We then check which queries answer correctly and ask how much structural variety remains among them. Next, we evaluate whether compile-style generation changes this combination of accuracy and structural agreement. Finally, we vary the presentation of the input to see how often the model's preferred construction changes. This sequence connects the distributions defined in Section 3.3 to concrete patterns in generated SQL.

4.1 Experimental Setup

Datasets. Our main experiments use the 1,034-question Spider development set (Yu et al., 2018), whose annotated queries and executable databases support joint execution and structural evaluation. Experiment 2 extends the analysis to 100-example subsets of BIRD Mini-Dev, Spider-Syn, and Dr.Spider (Li et al., 2023; Gan et al., 2021; Chang et al., 2023).

Models and sampling. We evaluate GPT-5-mini, GPT-4.1-mini, Claude-4.5-Sonnet, Claude-4.5-Opus, DeepSeek-V3.1, Gemini-2.5-Flash, and Gemini-3-Pro. Each input includes the question and database schema, including tables, columns, and foreign keys. We sample ten outputs per input, using temperature 1.0 and default nucleus sampling unless the model API restricts these settings. Appendix E provides the prompting details.

Repeated sampling gives each question a small collection of query choices. We retain the individual outputs so that their execution results can be matched to their canonical structures. This lets us follow the same collection from overall variation to variation among correct answers, connecting the first two experiments.

Execution and aggregation. We execute each query on its SQLite database and compare the result with the reference; execution errors and mismatches count as incorrect. Canonicalization groups parsed outputs by structure, both across all outputs and within the correct subset. We average per-question statistics, assigning zero to empty structural subsets as in Section 3.3. Each input has one batch of ten generations; a diagnostic rate of 0.01 on a 100-question subset corresponds to one question.

Model	D	$m \uparrow$	$\widehat{H}$	r
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: Repeated SQL generation on Spider yields different structural distributions: GPT-5-mini has the largest D and $\widehat{H}$, and GPT-4.1-mini has the highest m.

4.2 Experiment 1: Repeated-Generation Variance

We first hold the question and schema fixed and generate ten queries per question, yielding 10,340 queries per model. Table 1 summarizes whether their canonical structures concentrate on one construction or spread across several.

GPT-5-mini produces the most distinct structures, 1.913 per question, and the highest normalized entropy, 0.413. GPT-4.1-mini has a higher majority ratio, 0.793 versus 0.650, with lower diversity and entropy. These patterns distinguish a broader set of alternatives from a stronger tendency to repeat one construction.

For GPT-4.1-mini, m 0.793 and r 0.253 show that a recurring construction can differ from the annotation. This leads to the next question: which constructions answer correctly? Empty parsed subsets contribute zero, explaining the below-one average distinct counts for several models.

4.3 Experiment 2: Validating *ECSD*

We execute the same queries from Experiment 1 and examine the correct subset (Table 2). This follows the full structural distribution into the outputs that successfully answer the question.

Gemini-3-Pro has the highest e, 0.842, while GPT-4.1-mini has the highest dataset-average c, 0.656. Thus, answer frequency and correct-output concentration show different patterns. e_s adds execution context: Claude-4.5-Sonnet scores 0.757 over all outputs and 0.804 among queries that execute successfully.

For GPT-5-mini, the average distinct count remains 1.378 among correct outputs, compared with 1.913 overall. Its h of 0.351 means that 35.1% of questions have at least eight correct outputs spanning multiple parsed structures. Structural variety therefore persists on many questions the model repeatedly answers correctly.

All seven models have positive h, documenting *ECSD* across model families. ℓ ranges from 0.163 to 0.337, locating dispersed correct structures or missing parsed correct outputs. The rates jointly describe correct-answer diversity.

Validating *ECSD* beyond Spider. We repeat the analysis on three 100-example subsets using GPT-5-mini, Claude-4.5-Sonnet, and DeepSeek-V3.1, with ten queries per question (Table 3). This checks the same pattern under additional question and database settings.

h reveals *ECSD* in all three subsets. For GPT-5-mini, it reaches 0.150 on Spider-Syn, 0.110 on BIRD Mini-Dev, and 0.070 on Dr.Spider. On Spider-Syn, 15 questions thus have frequent correct answers with multiple constructions, alongside a dataset-average e of 0.229. Appendix B adds coverage and diversity statistics. We next examine how changing the generation process affects these patterns.

4.4 Experiment 3: A Full-Pipeline Case Study

The compile-style procedure makes query components explicit before deterministic assembly. We compare it with Direct SQL and the decomposed-prompting method DIN-SQL (Pourreza and Rafiei, 2023) on Spider using GPT-5-mini, ten outputs per question, and the same decoding configuration. The comparison asks whether this process improves both accuracy and correct-output agreement.

Alongside the shared execution and structural checks, we track valid JSON (j), successful compilation (κ), and successful execution (η). These rates and ρ divide by all generation attempts before averaging across questions, tracing the path from representation to runnable query.

Compile-style generation improves both outcomes (Table 4): e reaches 0.785 versus 0.742 for Direct SQL and 0.736 for DIN-SQL; c reaches 0.632 versus 0.552 and 0.579. The gains over Direct SQL are 4.3 percentage points in accuracy and 0.080 in average c, which reflects both correct structural coverage and concentration.

The full sample adds another part of the story. Compile-style generation produces 2.527 distinct structures on average, compared with 1.913 for Direct SQL, while also achieving the higher c above. The output collection becomes more varied overall as the correct-output agreement measure improves.

Model	$e \uparrow$	$e_s \uparrow$	D_c	$c \uparrow$	h	ℓ
GPT-5-mini	0.741	0.741	1.378	0.552	0.351	0.297
GPT-4.1-mini	0.754	0.757	1.129	0.656	0.250	0.163
Claude-4.5-Sonnet	0.757	0.804	0.714	0.624	0.054	0.201
Claude-4.5-Opus	0.816	0.816	0.650	0.600	0.038	0.224
DeepSeek-V3.1	0.771	0.771	0.774	0.612	0.101	0.191
Gemini-2.5-Flash	0.638	0.743	0.509	0.484	0.016	0.191
Gemini-3-Pro	**0.842**	0.843	0.721	0.537	0.139	0.337

Table 2: Execution accuracy and structural agreement differ across models on Spider. Gemini-3-Pro has the highest e, whereas GPT-4.1-mini has the highest c; positive h values reveal *ECSD* at high accuracy.

Subset	Model	$e \uparrow$	$c \uparrow$	ℓ	h
BIRD Mini-Dev	GPT-5-mini	0.145	0.091	0.090	0.110
	Claude-4.5-Sonnet	0.214	0.210	0.060	0.040
	DeepSeek-V3.1	0.246	0.212	0.170	0.090
Spider-Syn	GPT-5-mini	0.229	0.186	0.070	0.150
	Claude-4.5-Sonnet	0.332	0.410	0.010	0.000
	DeepSeek-V3.1	0.355	0.300	0.220	0.100
Dr.Spider	GPT-5-mini	0.122	0.121	0.030	0.070
	Claude-4.5-Sonnet	0.244	0.276	0.030	0.010
	DeepSeek-V3.1	0.266	0.287	0.110	0.060

Table 3: *ECSD* is observed on the three additional subsets. h is positive for all evaluated settings except Claude-4.5-Sonnet on Spider-Syn.

Measure	Direct SQL	DIN-SQL	Compile-style
$e \uparrow$	0.742	0.736	**0.785**
$c \uparrow$	0.552	0.579	**0.632**
D	1.913	1.553	2.527
$j \uparrow$	–	–	0.998
$\kappa \uparrow$	–	–	0.971
$\rho \uparrow$	–	–	0.902
$\eta \uparrow$	–	–	0.959

Table 4: On Spider with GPT-5-mini, compile-style generation improves e and c over both baselines while producing more distinct structures overall.

The two views in Section 3.3 connect overall variety with the constructions that produce correct answers.

The intermediate checks help trace this process. Valid JSON is produced in 99.8% of attempts, 97.1% compile, and 95.9% execute successfully. Structural parsing succeeds for 90.2%; parsing and database execution are separate checks, so each describes a different part of the output's usability. The examples in Appendices C and F make the remaining query choices visible, including successful constructions and omitted joins or predicates. These examples connect the aggregate improvements to concrete components that can be inspected in the generated representation.

4.5 Experiment 4: Input Perturbations

The preceding experiments follow repeated generation for a fixed input and examine a change to the generation pipeline. We now return to the input side of the framework: when the intended question stays the same but its presentation changes, does the model continue to favor the same SQL construction? This connects the observed sampling distributions to the perturbation measures in Section 3.3.

We use 200 Spider development questions with two kinds of variants. Question paraphrases change the wording; schema variants reorder tables or columns while preserving the database entities and foreign keys. Candidate paraphrases are generated with an LLM prompt and manually checked to preserve requested attributes, aggregation, filters, comparisons, ordering, limits, and schema meaning. Appendix E.2 gives the prompt and checking criteria.

Repeated generation estimates each input's majority structure. At the same time, v compares these majorities, σ tracks departures from the original, and f records affected questions. The pooled count D_p describes the range of constructions across variants. These comprehensive measures thoroughly capture structural variations.

Model	$v \uparrow$	D_p	$\sigma \downarrow$	$f \downarrow$
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-4.5-Sonnet	0.755	2.190	0.179	0.365
Claude-4.5-Opus	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 5: Question paraphrases change dominant SQL structures across models. GPT-5-mini has the highest f, while Gemini-3-Pro has the highest v.

Model	$v \uparrow$	D_p	$\sigma \downarrow$	$f \downarrow$
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-4.5-Sonnet	0.955	1.130	0.043	0.065
Claude-4.5-Opus	0.955	1.130	0.043	0.065
Gemini-3-Pro	**0.957**	1.060	**0.043**	**0.065**

Table 6: Schema reordering yields higher v than question paraphrasing for every evaluated model. Each model exhibits majority-structure changes on some questions.

Question paraphrases. Rewording a question often changes the sampled majority structure (Table 5). For GPT-5-mini, f is 0.900: 180 of the 200 questions have a changed majority under at least one paraphrase. Its v of 0.328 shows that agreement between presentations is also relatively low. Gemini-3-Pro retains its preferred construction more often, with v 0.894 and f 0.195, corresponding to 39 questions with a change. The same evaluation thus exposes both widespread changes and more persistent structural preferences across models.

The pooled samples show the variety accompanying these shifts. Across paraphrases, an average question elicits 13.220 distinct structures from GPT-5-mini and 1.530 from Gemini-3-Pro. Their σ values, 0.622 and 0.079, show how often the majority changes from the original input. Together, D_p and σ describe the range of constructions and the persistence of the model's preferred choice.

Schema presentation. We then keep the question wording fixed and change the order in which the schema is shown. This tests a second source of presentation changes within the same task. Table 6 shows higher v than the paraphrase experiment for every model. For GPT-5-mini, v rises from 0.328 to 0.490 and f falls from 0.900 to 0.640. For Gemini-3-Pro, v rises from 0.894 to 0.957 and f falls from 0.195 to 0.065. Schema reordering thus preserves the preferred construction more often than question paraphrasing.

Schema reordering changes the majority for 128 questions with GPT-5-mini and 13 with Gemini-3-Pro. Both perturbations thus expose input-sensitive query choices. Together with the earlier experiments, these alternatives connect repeated-generation diversity and correct answers with the semantic inspection discussed next.

5 Discussion

The effect of *ECSD* depends on the data: tied minima can separate `LIMIT 1` from a `MIN` subquery, while counting rows and non-null identifiers gives the same result. These differences guide semantic tests involving ties, nulls, and duplicates (Guagliardo and Libkin, 2017; Chu et al., 2018; Zhong et al., 2020).

Reading accuracy, correct-output diversity, and perturbation sensitivity together helps identify query alternatives for inspection (Ribeiro et al., 2020). The compile-style results illustrate this joint view: accuracy and correct-output agreement improve alongside greater overall diversity.

Coverage and sampling shape these comparisons. Empty correct subsets contribute zero to c, so ρ and D_c provide useful context. Matched sampling budgets and independent repeats help clarify recurring structural patterns.

6 Conclusion and Future Work

We introduced **SQLStructEval** to connect execution accuracy with the structure of generated SQL. Experiments revealed *ECSD* across models and datasets, structural changes under input perturbations, and improved accuracy and correct-output agreement with compile-style generation. These findings show how structural analysis makes the routes to a correct answer visible. Following these routes across samples and input variants helps identify recurring query choices and concrete alternatives for closer inspection.

In future work, we plan to test database changes and separate generation and compilation effects. To understand if structurally different but currently correct queries remain equivalent as data evolves, we will test them against edge cases like tied values, nulls, and duplicates.

Limitations

Our main evaluation uses Spider, with additional 100-question subsets from three benchmarks and ten generations per input. Larger datasets, repeated sampling, and threshold comparisons would help assess how consistently the observed patterns extend across settings. Structural statistics depend on parsing coverage, while execution correctness is measured on the supplied database instances. Further semantic tests and component ablations would clarify the practical effects of query alternatives and the contributions of the compile-style pipeline.

Ethical Considerations

We evaluate generated queries offline using publicly available Text-to-SQL benchmarks. In practical applications, incorrect queries can lead to misleading decisions. Structural diagnostics can guide inspection, alongside execution tests and semantic checks suited to the application's data and requirements.

Data and artifact use. We use Spider, BIRD Mini-Dev, Spider-Syn, and Dr.Spider for research evaluation under their respective access and licensing terms. Derived question variants and generated queries remain associated with those source resources. The code link provides the evaluation implementation; use and redistribution of datasets and model outputs must respect the original resource and provider terms. Hosted model updates can affect reproducibility even when the prompt remains fixed.

AI assistance. For manuscript preparation, we used AI solely for language polishing.

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)*, ACL-SRW '25, pages 834–842, Vienna, Austria. Association for Computational Linguistics.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *arXiv preprint*, arXiv:2108.07732.

Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Wang, Zhiguo Wang, Vittorio Castelli, Bing Xiang, and Patrick Ng. 2023. Dr.Spider: A diagnostic evaluation benchmark towards text-to-SQL robustness. In *Proceedings of the Eleventh International Conference on Learning Representations*, ICLR '23, Kigali, Rwanda. OpenReview.net.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde 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, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint*, arXiv:2107.03374.

Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proceedings of the VLDB Endowment*, 11(11):1482–1495.

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*, COLING '22, pages 2166–2187, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-SQL evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, ACL '18, pages 351–360, Melbourne, Australia. Association for Computational Linguistics.

Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. 2021. Towards robustness of text-to-SQL models against synonym substitution. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, ACL-IJCNLP '21, pages 2505–2515, Online. Association for Computational Linguistics.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment*, 17(5):1132–1145.

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*, EMNLP '23, pages 10932–10952, Singapore. Association for Computational Linguistics.

Paolo Guagliardo and Leonid Libkin. 2017. A formal semantics of SQL queries, its validation, and applications. *Proceedings of the VLDB Endowment*, 11(1):27–39.

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*, ACL '19, 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 *Proceedings of the Twelfth International Conference on Learning Representations*, ICLR '24, Vienna, Austria. OpenReview.net.

Heegyu Kim, Taeyang Jeon, Seunghwan Choi, Seungtaek Choi, and Hyunsouk Cho. 2025. FLEX: Expert-level false-less EXecution metric for reliable 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)*, NAACL '25, pages 4448–4475, Albuquerque, New Mexico, USA. Association for Computational Linguistics.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida I. Wang, and Tao Yu. 2025. Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows. In *Proceedings of the Thirteenth International Conference on Learning Representations*, ICLR '25, Singapore. OpenReview.net.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM already serve as a database interface? A BIg bench for laRge-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems*, volume 36 of *NeurIPS '23*, pages 42330–42357, New Orleans, Louisiana, USA. Curran Associates, Inc.

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*, Findings-EMNLP '25, 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 *Advances in Neural Information Processing Systems*, volume 36 of *NeurIPS '23*, pages 21558–21572, New Orleans, Louisiana, USA. Curran Associates, Inc.

Ruilin Luo, Liyuan Wang, Binghuai Lin, Zicheng Lin, and Yujiu Yang. 2024. PTD-SQL: Partitioning and targeted drilling with LLMs in text-to-SQL. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, EMNLP '24, pages 3767–3799, Miami, Florida, USA. Association for Computational Linguistics.

Franklin Ma and Alan J. Hu. 2025. Logically constrained decoding. In *Proceedings of the 3rd Workshop on Mathematical Natural Language Processing*, MathNLP '25, pages 150–167, Suzhou, China. Association for Computational Linguistics.

Peixian Ma, Xialie Zhuang, Chengjin Xu, Xuhui Jiang, Ran Chen, and Jian Guo. 2025. SQL-R1: Training natural language to SQL reasoning model by reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 38 of *NeurIPS '25*, pages 174505–174537, San Diego, California, USA. Curran Associates, Inc.

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*, volume 202 of *ICML'2023*, pages 26106–26128, Honolulu, Hawaii, USA. PMLR.

Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-aligned decoding. In *Advances in Neural Information Processing Systems*, volume 37 of *NeurIPS '24*, pages 24547–24568, Vancouver, British Columbia, Canada. Curran Associates, Inc.

Xinyu Pi, Bing Wang, Yan Gao, Jiaqi Guo, Zhoujun Li, and Jian-Guang Lou. 2022. Towards robustness of text-to-SQL models against natural and realistic adversarial table perturbation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, ACL '22, pages 2007–2022, Dublin, Ireland. Association for Computational Linguistics.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *Proceedings of the Tenth International Conference on Learning Representations*, ICLR '22, Online. OpenReview.net.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Özcan, and Sercan Ö. Arık. 2025. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. In *Proceedings of the Thirteenth International Conference on Learning Representations*, ICLR '25, Singapore. OpenReview.net.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In *Advances in Neural Information Processing Systems*, volume 36 of *NeurIPS '23*, pages 36339–36348, New Orleans, Louisiana, USA. Curran Associates, Inc.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-SQL capabilities of large language models. *arXiv preprint*, arXiv:2204.00498.

Prateek Rajput, Abdoul Aziz Bonkoungou, Yewei Song, Abdoul Kader Kabore, Iyiola E. Olatunji, Jacques Klein, and Tegewende Bissyande. 2025. Dynamic stability of LLM-generated code. *arXiv preprint*, arXiv: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)*, ACL-Industry '25, pages 485–499, Vienna, Austria. Association for Computational Linguistics.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, ACL '20, pages 4902–4912, Online. Association for Computational Linguistics.

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*, EMNLP '21, pages 9895–9901, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Jinwang Song, Hongying Zan, Kunli Zhang, Lingling Mu, Yingjie Han, Haobo Hua, and Min Peng. 2025a. JOLT-SQL: Joint loss tuning of text-to-SQL with confusion-aware noisy schema sampling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, EMNLP '25, pages 6040–6053, Suzhou, China. Association for Computational Linguistics.

Yewei Song, Tiezhu Sun, Xunzhu Tang, Prateek Kumar Rajput, Tegawendé F. Bissyandé, and Jacques Klein. 2025b. Measuring LLM code generation stability via structural entropy. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, ASE '25, pages 3922–3926, Seoul, Republic of Korea. IEEE.

Alane Suhr, Ming-Wei Chang, Peter Shaw, and Kenton Lee. 2020. Exploring unexplored generalization challenges for cross-database semantic parsing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, ACL '20, pages 8372–8388, Online.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, ACL '20, pages 7567–7578, Online.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *Proceedings of the Eleventh International Conference on Learning Representations*, ICLR '23, Kigali, Rwanda. OpenReview.net.

Jiaxi Yang, Binyuan Hui, Min Yang, Jian Yang, Junyang Lin, and Chang Zhou. 2024. Synthesizing text-to-SQL data from weak and strong LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, ACL '24, pages 7864–7875, Bangkok, Thailand. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. 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*, EMNLP '18, pages 3911–3921, Brussels, Belgium.

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*, volume 235 of *ICML '24*, pages 59670–59684, Vienna, Austria. PMLR.

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 '20, pages 396–411, Online.

A Structural Measure Details

Parse-failure conventions. Our evaluation implementation assigns zero to counts, majority ratios, and entropies when the selected structural subset is empty. Dataset summaries average these values over all questions. The denominator therefore includes questions with no usable structure. An average below one can consequently reflect structural coverage as well as concentration among available structures.

Subset interpretation. The metric definitions and symbols in Section 3.3 apply throughout. c and D_c use parsed execution-correct outputs, while D uses all parsed outputs. A low c can reflect either *ECSD* or an empty correct structural subset. ℓ inherits this coverage dependence; h explicitly requires multiple parsed correct structures and therefore identifies *ECSD* at high accuracy.

B Additional Scope-Validation Statistics

Table 7 complements the main-text subset results with parsing coverage and structural diversity. Each row summarizes 100 questions with ten generations per question; the zero convention above applies to all structural averages.

C 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.

C.1 Diagnostic Categories

We assign inspected queries to six diagnostic categories. Five describe errors, and the sixth records structural variation among execution-correct alternatives:

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

Schema linking is also an explicit target of recent methods: JOLT-SQL jointly optimizes schema linking and SQL generation and uses confusion-aware noisy schema sampling to improve robustness (Song et al., 2025a). Our diagnostic categories distinguish these schema-reference errors from errors in query logic and from variation among execution-correct queries.

C.2 Category Distribution

Table 8 compares category proportions for Direct SQL and compile-style generation. Each ratio represents the fraction of inspected examples assigned to the corresponding primary category. We inspected 100 examples for each setting, sampled from cases with execution errors or structural disagreement. One author annotated the examples and a second author checked the labels. We assigned mutually exclusive primary categories, choosing the earliest issue that determined failure or structural divergence when multiple issues occurred. Here, Δ is the compile-style proportion minus the Direct SQL proportion. The proportions summarize these selected cases of execution errors and structural disagreement.

As shown in Table 8, the selected compile-style sample has smaller proportions of several semantic-error categories, including schema linking, logical form mistakes, incorrect join paths, and predicate errors. However, two categories increase: structurally diverse but executable queries (structural variation) and errors in the intermediate representation or its compilation (compilation). This pattern is consistent with the quantitative observations reported in Experiment 3.

Subset	Model	$\rho \uparrow$	D	D_c	$c \uparrow$	H_c
BIRD Mini-Dev	GPT-5-mini	0.657	3.380	0.560	0.091	0.200
	Claude-4.5-Sonnet	0.496	1.060	0.290	0.210	0.041
	DeepSeek-V3.1	0.399	1.600	0.500	0.212	0.152
Spider-Syn	GPT-5-mini	0.713	2.630	0.540	0.186	0.177
	Claude-4.5-Sonnet	0.686	1.200	0.410	0.410	0.000
	DeepSeek-V3.1	0.453	1.260	0.530	0.300	0.126
Dr.Spider	GPT-5-mini	0.668	2.610	0.300	0.121	0.092
	Claude-4.5-Sonnet	0.720	1.460	0.290	0.276	0.010
	DeepSeek-V3.1	0.470	1.490	0.420	0.287	0.072

Table 7: Parsing coverage varies substantially across the additional subsets: For example, GPT-5-mini has the greatest overall structural diversity in each subset, while Claude-4.5-Sonnet has no observed correct-output diversity on Spider-Syn.

Category	Direct SQL	Compile-style	Δ
Schema linking	0.20	0.13	-0.07
Logical form	0.18	0.15	-0.03
Join path	0.14	0.09	-0.05
Predicate	0.16	0.10	-0.06
Structural variation	0.28	0.40	+0.12
Compilation	0.04	0.13	+0.09

Table 8: The inspected compile-style sample has smaller shares of schema, logic, join, and predicate errors, but larger shares of structural variation and compilation errors.

C.3 Representative Case Studies

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

Direct SQL Execution Failures. The inspected examples include schema-linking and join-path errors in direct generation. The intermediate representation makes table relations explicit, allowing us to inspect the model's table choices and joins before compilation.

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. An inspected baseline output omits the join; such queries 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.

Examples of *ECSD*. We also observe *ECSD* in the inspected baseline outputs: multiple generated queries return the correct result but have different canonical structures. For example, for the following question:

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

the inspected baseline outputs include 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. The compile-style example uses a simple `GROUP BY` aggregation.

Paraphrase Sensitivity. Semantically equivalent questions can trigger different query structures in baseline generation. For example, the following 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 annotated reference SQL query. However, the inspected baseline generations use different structural patterns for the two inputs, including variations involving subqueries or derived tables. This example illustrates why agreement across paraphrases is evaluated separately from agreement across repeated generations of one input.

Compile-style Failures. 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.

These cases illustrate successful query construction as well as failures in the intermediate representation. They highlight specific joins and predicates to examine when refining the representation and compiler.

D Implementation Details

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

D.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.

D.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 serves as the canonical comparison representation and is used as the basis for all structural measures 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 annotated reference queries, the annotated reference SQL is canonicalized using the same pipeline, and structural matches are determined by canonical representation equality.

D.3 Execution Environment

The main experiments use the Spider SQLite database files; the additional scope checks use the corresponding benchmark databases. We compare query execution results against the annotated reference result on the same database instance.

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 annotated reference SQL query. Queries that produce execution errors or mismatched results are considered incorrect.

Execution feedback also serves as a learning signal in recent Text-to-SQL systems: SQL-R1 uses rewards for successful execution and result correctness (Ma et al., 2025). Structural evaluation can complement such feedback by distinguishing query constructions within the execution-correct subset.

D.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 return one sample per request, we make independent calls and aggregate the outputs.

Decoding parameters. Unless restricted by the model API, we use stochastic decoding with temperature $= 1.0$ and default nucleus sampling parameters. Stochastic decoding lets us analyze structural variation across repeated generations.

Decoding constraints can also affect output distributions. Park et al. (2024) show that conventional grammar-constrained decoding can distort the model distribution. Structural diversity should therefore be interpreted together with the decoding policy used to produce the samples.

Compile-style generation. In Experiment 3, the compile-style setting asks models to generate a structured intermediate representation for compilation into SQL. 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.

E Prompt and Generation Settings

This section provides the prompt templates used in our experiments.

E.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:
```

E.2 Paraphrase Generation and Checking

Candidate paraphrases were generated using an LLM prompt and manually checked before use. The prompt is:

```
Rewrite the following Text-to-SQL question into
      semantically equivalent paraphrases.
    Preserve the requested attributes,
    aggregation, filters, comparison operators,
     ordering, limits, and all schema-specific
    meanings.
Do not add or remove constraints. Do not make
     the question more ambiguous.

Original question:
<question>

Output <n> paraphrases, one per line.
```

We excluded paraphrases that changed selected columns, aggregation targets, predicates, comparison relations, ordering, limits, or intended schema entities. We also excluded variants that added conditions, omitted necessary constraints, or introduced ambiguity. These criteria guide the check that each paraphrase preserves the original intent.

E.3 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 for compilation into SQL. 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:
```

F Additional Structural Examples

In order to complement the quantitative structural measures reported in the main empirical experiments detailed above, we present several highly detailed representative examples that illustrate various 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 4. 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 different structural representation.

Example 2: *ECSD* across paraphrases 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 annotated reference SQL query:

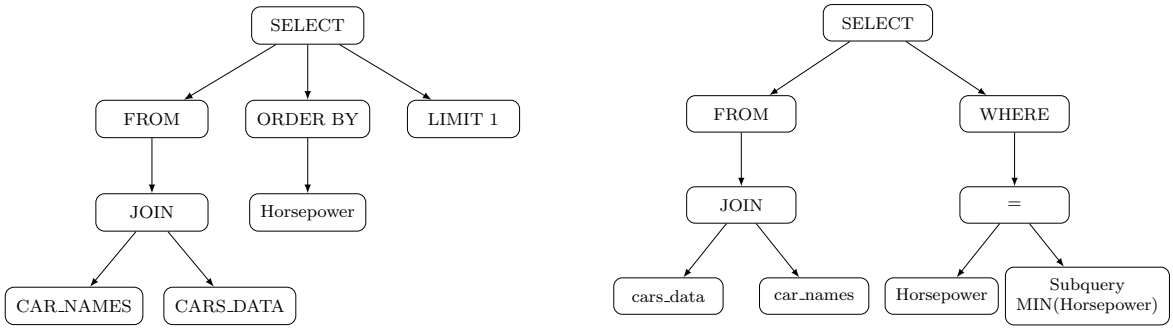
```
SELECT country, COUNT(*)
FROM singer
GROUP BY country;
```

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;
```

These generated queries illustrate *ECSD* across equivalent phrasings: both return the correct result on the evaluated database, but their ASTs differ in the aggregation argument (`COUNT(*)` vs. `COUNT(Singer_ID)`).



(a) AST of gold SQL query

(b) AST of generated SQL query

Figure 4: AST comparison for the minimum-horsepower example in Appendix F. The reference uses ordering and a row limit; the generated query uses an aggregate subquery. Both return the same result on the evaluated database.

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

The illustrated baseline output 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`. The displayed compiled SQL contains the aggregation structure and closely matches the gold AST.

G 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.

G.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"}
  ]
}
```

G.2 LLM Output: Structured JSON Intermediate Representation

In compile-style generation, the model generates a structured JSON representation that encodes the query components for compilation into SQL.

The two entries in `select` preserve the requested output attributes, `Name` and `Capacity`. The ranking attribute, `Average`, appears only in `order_by`; it need not be returned as an output column.

Descending order and a limit of one implement the superlative in the question. Here, "average attendance" refers to the stored `Average` column, so the representation contains no aggregation or grouping operation.

Although the simplified schema also includes `concert`, every required attribute belongs to `stadium`. The empty `joins` list therefore makes the single-table choice explicit. The empty predicate lists indicate that the question requests no additional filtering.

```
{
  "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`.

G.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.

G.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.

G.5 Reading and Checking the Pipeline

The example separates decisions made during generation from their realization in SQL. The model chooses the table, output attributes, ordering expression, and row limit in the JSON representation. The compiler carries these choices into the displayed AST and SQL. A deterministic translation makes a fixed representation reproducible, but a wrong table or ordering choice in that representation can still yield an incorrect query.

Component correspondence. Each populated field has a visible counterpart. The selected expressions become `Column` nodes and the SQL projection; the source becomes the `Table` node and `FROM` clause. The ordering field becomes an `OrderBy` node and `ORDER BY stadium.Average DESC`, while the row limit remains one. Empty lists introduce no join, predicate, or grouping clauses. This correspondence supports inspection beyond the execution result.

Semantic boundary cases. The query returns at most one row. If several stadiums share the largest average attendance, the displayed ordering contains no secondary key to select a particular stadium among them. A formulation that filters on the maximum value can instead return multiple rows in that case. Thus, agreement on the evaluated database does not establish equivalence under changes to the data. Testing tied values would help distinguish these constructions; no such additional experiment is reported here.

Connection to structural evaluation. The displayed internal AST explains compilation. Structural scoring subsequently applies the shared SQL parsing and canonicalization procedure to the compiled SQL, as described in Appendix D. Direct and compiled SQL thus enter the same evaluator; JSON formatting differences are not counted as SQL structures.