BIRD-PANDAS: DIAGNOSING DIVERGENCE BETWEEN TEXT-TO-SQL AND TEXT-TO-PANDAS

Anonymous authors
Paper under double-blind review

ABSTRACT

Text-to-SQL is a widely adopted paradigm for natural-language access to structured data, but many real-world analytics tasks involve file-based tables and multi-step transformations that are more naturally handled with code-based approaches. Controlled evaluation of Text-to-Pandas and Text-to-SQL under matched task conditions remains largely absent, making it difficult to isolate paradigm-specific challenges from task content. We introduce BIRD-Pandas, a paired benchmark that derives verified Pandas ground truth from BIRD and enables controlled comparison. Benchmark construction revealed that producing equivalent results across paradigms requires different decision logic due to differences in how the two paradigms allocate execution decisions, a phenomenon we term execution-semantic divergence. Text-to-Pandas trails Text-to-SQL in accuracy at smaller and mid-scale models, and this gap narrows substantially as model scale increases. However, the aggregate gap reflects multiple contributing factors, including execution-semantic divergence and task-specification insufficiency, where natural language questions omit information. Disentangling these factors requires supplying the missing task information and observing the resulting performance change. To this end, we introduce the Logic Completion Framework (LCF), a clarification protocol that measures performance recovery when the model requests and receives such information. LCF closes most of the cross-paradigm gap at intermediate scales. At the largest evaluated scale, SQL and Pandas achieve statistically comparable aggregate accuracy, yet SQL benefits more than Pandas from complete task specification. Analysis of post-LCF failures shows that errors attributable to task-specification insufficiency remain within both paradigms' own error sets, as models tend to overlook consequential ambiguities even when invited to ask clarifying questions, while the net cross-paradigm gap is instead driven mainly by execution-semantic divergence, which is concentrated on the Pandas side. Resources are available at `https://anonymous.4open.science/r/Bird_Pandas-3F82/`.

1 INTRODUCTION

The rapid advancement of Large Language Models (LLMs) has accelerated research on natural language interfaces for data access (Qin et al., 2022; Huang et al., 2025). The research community has extensively studied Text-to-SQL, which maps natural-language questions to executable queries over relational databases, benefiting from comprehensive benchmarks such as Spider (Yu et al., 2018) and BIRD (Li et al., 2023), together with advances in semantic parsing, end-to-end generation, and schema linking (Ruan et al., 2023; Kong et al., 2024; Sui et al., 2023; Zhang et al., 2025; Cao et al., 2024; Maamari et al., 2024; Safdarian et al., 2026). However, many analytical workflows operate directly on standalone files such as CSV, Excel, and JSON. While existing Text-to-Code research (Luo et al., 2025; Jiang et al., 2024; Li et al., 2025) primarily focuses on general programming tasks involving basic data structures (Chen et al., 2021; Zhuo et al., 2025), and existing table-oriented benchmarks (Egg et al., 2025; Lei et al., 2025; Huo et al., 2026) typically entangle core reasoning with external factors such as data cleaning or schema inference, the code generation demands of Text-to-Pandas under controlled conditions remain unexplored. In this work, we define Text-to-Pandas as the task of generating an executable Python program that uses Pandas DataFrame operations to answer a natural-language question over tabular data. Under these controlled conditions, a natural

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

comparison arises: how does the choice between SQL and Pandas as the target paradigm affect model reliability, and what mechanisms account for any observed difference?

To enable this comparison, we construct BIRD-Pandas, a paired benchmark derived from the BIRD development set. We rectify inconsistent reference SQL programs and derive verified Pandas programs so that both paradigms are evaluated on the same question, declared schema, and underlying data, with equivalent answer semantics. During SQL-to-Pandas conversion, we observed that even when the source SQL specifies the task intent, the converted Pandas program requires additional decision logic to produce the equivalent result. We term this execution-semantic divergence: the additional decision logic that Pandas generators must encode due to differences in how execution decisions are distributed between the generator and the execution engine. SQL delegates decisions such as null handling and result ordering to a unified database optimizer (Hagedorn et al., 2021), while Pandas distributes these decisions across more than 240 independently designed API methods (Petersohn et al., 2020; Wu, 2019), requiring the generator to encode them explicitly. For example, when the verified SQL ground truth establishes that NULL-key groups should be retained, SQL GROUP BY implements this by default, while Pandas groupby requires the generator to encode dropna=False (Figure 1(a)). In contrast, task-specification insufficiency arises when the natural-language question itself omits information required to infer the task intent, such as a boundary condition or null-value policy (Figure 1(b)), affecting both paradigms until clarification is provided.

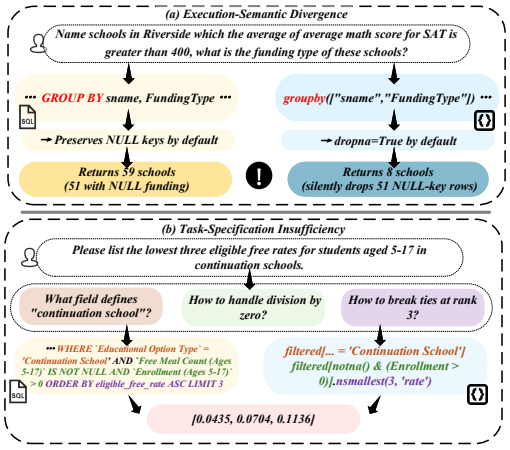


Figure 1: Illustrating task-specification insufficiency and execution-semantic divergence.

Evaluating open-source and proprietary LLMs on BIRD-Pandas, we observe an accuracy gap between Text-to-SQL and Text-to-Pandas whose magnitude tends to decrease as model scale increases. However, the aggregate gap reflects multiple contributing factors, including execution-semantic divergence and task-specification insufficiency, where natural language questions omit information required by the task intent. The aggregate result alone cannot separate these contributions (Huo et al., 2026; Ding et al., 2026).

We introduce the Logic Completion Framework (LCF) to diagnose these sources. LCF is an oracle-assisted clarification protocol in which the model requests one round of task-level clarification before program generation. The Oracle, grounded in verified ground truth, answers only the requested business-logic information, without providing code or target-language operations. Performance recovery after clarification indicates that task-specification insufficiency contributed to the baseline error. At the largest evaluated scale, a residual gap remains after clarification. To characterize this residual, we decompose post-LCF errors. This decomposition shows that task-specification insufficiency still accounts for most of the residual within each paradigm's own error set, since even with an opportunity to ask clarifying questions, models tend to overlook ambiguities that affect the result; when attributing the net cross-paradigm gap, execution-semantic divergence drives the majority of the difference and appears exclusively on the Pandas side. Both sources tend to produce results that deviate from task intent in ways that appear plausible but are incorrect, making such failures difficult to detect without reference answers.

Our contributions are threefold. (1) BIRD-Pandas provides a paired benchmark that enables controlled comparison of Text-to-SQL and Text-to-Pandas. (2) The LCF supplies a diagnostic protocol that measures performance recovery after model-initiated clarification and supports analysis of task-specification insufficiency. (3) Our analysis of post-LCF failures shows that while LCF closes most of the cross-paradigm gap at intermediate scales, a residual gap persists at the largest evaluated scale; task-specification insufficiency accounts for a substantial share of errors within both paradigms' own error sets, while the net cross-paradigm gap is driven mainly by execution-semantic divergence, appearing exclusively on the Pandas side.

2 Related Work

Maturation of Text-to-SQL Systems. Complex analytical tasks increasingly demand the multi-step code generation capability of Text-to-Code approaches (Liu et al., 2024; Ouyang et al., 2025; Zhu et al., 2025), a shift from the rule-based to LLM-driven maturation of Text-to-SQL (Zhong et al., 2017; Wang et al., 2025; Chaturvedi et al., 2025). Existing code generation research heavily skews toward general-purpose programming (Luo et al., 2025; Jiang et al., 2024), largely neglecting tabular data science workflows. SQL gives the database engine the opportunity to understand and mediate physical execution planning through a rigid schema and a unified DBMS execution engine (Hagedorn et al., 2021). Generating Pandas data science code, by contrast, requires models to encode each execution decision explicitly as a sequence of function calls, including orchestrating multi-step DataFrame operations and managing intermediate states, without a unified engine to enforce constraints (Wu, 2019). These characteristics call for domain-specific evaluation in analytical contexts.

Scope of Existing Benchmarks. Early benchmarks focused on structural complexity (e.g., Spider (Yu et al., 2018)) or cross-domain knowledge (e.g., BIRD (Li et al., 2023)). Execution-based metrics in these settings may penalize models for missing implicit constraints rather than actual reasoning failures (Pourreza & Rafiei, 2023; Zhang et al., 2024). General code generation benchmarks such as HumanEval (Chen et al., 2021) and SWE-Bench (Zhuo et al., 2025) assess broad programming capabilities but do not capture the nuances of data-oriented retrieval tasks.

Controllable Evaluation of Multi-Step Code Generation. Recent efforts address complex data workflows through agentic and interactive settings. DaBStep (Egg et al., 2025) and Spider 2.0 (Lei et al., 2025) evaluate multi-step reasoning across heterogeneous enterprise environments, while BIRD-INTERACT (Huo et al., 2026) and AmbiSQL (Ding et al., 2026) focus on clarifying ambiguous requests through dialogue. Although valuable for assessing real-world interactions, these open-ended benchmarks inherently entangle information retrieval and dialogue strategies with core reasoning. Our work provides a controlled environment where both paradigms receive the same natural-language question, declared schema, and underlying data.

3 Experimental Framework

3.1 Problem Formulation

We formulate Text-to-SQL and Text-to-Pandas as paired program generation tasks (Figure 2). Given a natural-language query Q and a database schema $\mathcal{S}$, the model generates a program P that operates on a data instance to produce an answer A. Both paradigms access the same logical data content, instantiated as $\mathcal{D}_{\text{db}}$ (database) for SQL and $\mathcal{D}_{\text{df}}$ (CSV-loaded DataFrame) for Pandas. We evaluate both paradigms on the same tuple $(Q, \mathcal{S}, \mathcal{D}_{\text{db}}, \mathcal{D}_{\text{df}}, A_{gold})$, where A_{gold} is derived from the verified SQL execution result, and assess whether the predicted answer A_{pred} is logically equivalent to A_{gold}.

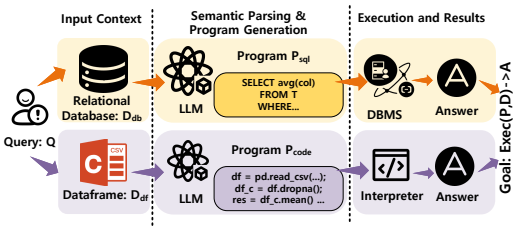
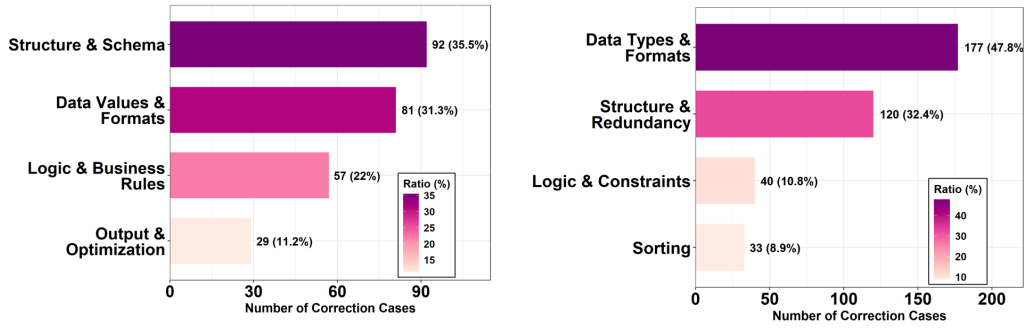


Figure 2: Comparative framework of paradigms.

Two sources of cross-paradigm performance disparity are distinct. *Task-specification insufficiency* arises when the natural-language query Q omits information required to infer the task intent, such as boundary conditions, aggregation scope, or null-value policies. Both paradigms face this challenge. *Execution-semantic divergence* arises from paradigm-specific execution mechanisms, given a specified task intent: SQL delegates execution decisions to a unified database optimizer, while Pandas distributes these decisions across independently designed API methods, requiring explicit encoding by the generator. When the verified SQL ground truth establishes the task intent, producing equivalent results in Pandas may require additional decision logic to match the reference semantics. Because



(a) Difference categories in reference SQL purification.

(b) Difference categories in SQL-to-Pandas code generation.

Figure 3: Correction distributions for (a) dataset rectification and (b) SQL-to-Pandas conversion.

BIRD ground truth is SQL-centric, we derive the reference answer semantics from the verified SQL execution results to enable paired comparison.

3.2 Benchmark Construction

3.2.1 Dataset Correction

To systematically compare the Text-to-Pandas and Text-to-SQL paradigms, we build on the BIRD benchmark (Li et al., 2023), specifically using the BIRD development set, which provides 1,534 query-SQL pairs with domain knowledge annotations. Each question provides expert-verified SQL ground truth for deriving Pandas equivalents, and BIRD's standardized database exports provide the same underlying data for both paradigms. This controls the data source and task content, while the database-backed SQL interface and CSV-based Pandas loading interface remain operationally different.

However, preliminary error analysis revealed that evaluation metrics were affected by annotation inconsistencies, where valid model outputs were marked as incorrect due to inaccuracies in the reference SQL. To establish a reliable evaluation foundation, we implemented a two-stage purification pipeline integrating Model-in-the-Loop verification with expert review (Li et al., 2024; 2023). First, 1,193 queries were automatically accepted when the executed outputs of SQL programs independently generated by three LLMs (Qwen3-238B, DeepSeek-R1, and Qwen-Max) all matched the original BIRD reference SQL execution result. The remaining 341 divergent cases underwent double-blind expert review, resulting in 259 corrections and the confirmation of 82 original annotations as valid.

As illustrated in Figure 3(a), these corrections span four primary dimensions: *Structure & Schema*, *Data Values & Formats*, *Logic & Business Rules*, and *Formatting & Constraints*. Detailed definitions and case studies for each category, along with the rationale for utilizing structurally verified databases over raw schema-less CSVs, are provided in Appendix B.

3.2.2 SQL-to-Pandas Conversion

Building on this rectified foundation, we convert the verified SQL ground truth into Pandas equivalents using the same verification pipeline employed in dataset correction. This process produces verified ground truth for the Text-to-Pandas task while exposing execution-semantic divergences between the two paradigms. Of the instances processed, 370 required manual intervention and were resolved via double-blind expert review; these cases manifest across four dimensions described below (see Figure 3(b)).

Manual Refinement Among the 370 instances resolved via double-blind expert review, mismatches between the converted Pandas program and the verified SQL ground truth arise even though the source SQL query specifies the task intent. Under these conditions, execution-semantic divergence is the operative cause: given a specified task intent, producing equivalent results across paradigms

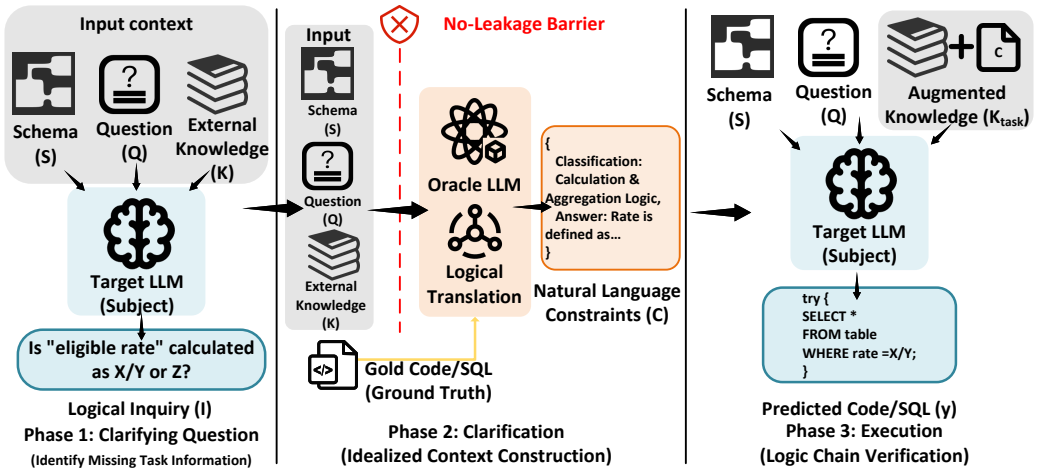


Figure 4: The Logic Completion Framework (LCF).

requires different decision logic. This divergence spans four dimensions: *Data Types & Formats*, where SQL performs implicit type coercion while Pandas requires explicit type handling; *Structure & Redundancy*, where SQL's relational operators implicitly manage overlapping rows while Pandas requires explicit deduplication logic; *Logic & Constraints*, where SQL and Pandas assign different default semantics to missing values (e.g., SQL's `GROUP BY` retains `NULL`-key groups while Pandas's `groupby` removes `NaN`-key groups by default); and *Sorting*, where deterministic output requires a complete sort key specification. When the query underspecifies ordering, SQL may produce results determined by internal storage layout, and Pandas generators must infer additional sorting conditions to match the SQL-derived reference. Detailed case studies for these categories are provided in Appendix C.

Execution Evaluation We evaluate model predictions at the execution-result level. Standard metrics such as Execution Accuracy (EX) rely on exact matches between outputs, but logically correct answers may still differ in column order, numeric type, duplicated rows, or lightweight result structure, especially for Pandas outputs.

Logical Validator We use a structure-agnostic logical validator ($\mathcal{V}_{logic}$) for both SQL and Pandas contexts. Rather than designing separate rule-based normalizers or relying on SQL-specific program-equivalence checks, $\mathcal{V}_{logic}$ compares only the predicted and verified ground truth execution outputs under one shared logical-equivalence criterion. This keeps the evaluation post-execution, format-agnostic, and symmetric across the two paradigms. Following the LLM-as-a-Judge methodology (Zheng et al., 2023; Kim et al., 2025; Huo et al., 2026), $\mathcal{V}_{logic}$ is instantiated with Qwen3-Coder-Flash accessed via the Qwen API and determines whether the predicted result is logically equivalent to the verified ground truth. The full prompt is provided in Appendix D.

Human Verification To establish trust in $\mathcal{V}_{logic}$, we audited its decisions on a stratified sample of $N = 600$ instances, balanced across SQL and Pandas outputs, validator-positive and validator-negative decisions, and three difficulty levels (Simple, Moderate, Hard). Two domain experts independently adjudicated each instance following the same double-blind protocol used in dataset rectification, yielding an observed agreement rate of 96.7% (580/600). The 20 discrepant cases were resolved through subsequent discussion, reaching unanimous consensus. This provides empirical confidence that $\mathcal{V}_{logic}$ decisions are broadly consistent with expert judgment on a representative sample.

3.3 Logic Completion Framework

The benchmark construction process above shows that execution-semantic divergence can arise even when task intent is specified, as in the verified SQL ground truth. However, model-generated

code is conditioned on the original natural-language query, which may still underspecify task intent, introducing task-specification insufficiency as a distinct source of error that affects both paradigms. To examine these factors, we introduce the Logic Completion Framework (LCF), a clarification protocol that measures performance recovery when the model requests and receives additional task information.

LCF operates through a three-phase mechanism (Figure 4). In Phase 1, the evaluated model (Subject) identifies task-specification insufficiency and poses clarifying questions targeting ambiguous definitions, missing task information, or boundary conditions. In Phase 2, an Oracle grounded in verified ground truth provides brief natural-language clarifications that answer the inquiry, constrained to the business-logic level without code or target-language operations. In Phase 3, the Subject generates the final program conditioned on the original query augmented with the clarification.

3.3.1 Mechanism and Execution

Phase 1 and Phase 3 are performed by the evaluated model (Subject); Phase 2 is performed by a domain-expert proxy (Oracle). The detailed design is provided in Appendix E.

Phase 1: Clarifying Question. The Subject analyzes the input query to identify task-specification insufficiency. Instead of immediate code generation, it identifies all suspected information gaps and poses corresponding clarifying questions in a single turn (Q_{clar}) targeting ambiguous definitions, underspecified task requirements, or boundary conditions.

Phase 2: Clarification. The Oracle uses the verified ground truth to provide brief natural-language clarifications (K_{task}) that answer the inquiry. The Oracle is constrained to the business-logic level: it expresses task requirements in natural language without code fragments or API-level syntax, answers only what the Subject explicitly asked, and names specific column names or values only when the Subject's question directly requires them, without disclosing execution-level operations.

Phase 3: Execution. The Subject generates the final program (P_{pred}) conditioned on the original query augmented with K_{task}.

4 Experiments

4.1 Experimental Setup

We systematically benchmark the Text-to-Pandas paradigm against Text-to-SQL using the following operational configurations:

Datasets. We evaluate on the BIRD development set (Li et al., 2023), comprising 1,534 instances, under two verified ground truth settings: the original BIRD SQL ground truth for baseline Text-to-SQL evaluation, and our verified Pandas ground truth for paired cross-paradigm comparison. Both configurations share identical natural language questions and underlying data; they differ only in the target execution paradigm.

Evaluation Metrics. We employ Execution Accuracy (EX) as the primary quantitative metric. To reduce false negatives caused by superficial formatting differences in Pandas outputs, we integrate the LLM-based logical validator ($\mathcal{V}_{logic}$) to support format-agnostic assessment. Accuracy scores reported in our results (Tables 1, 2) represent validator-based logical equivalence rather than traditional rule-based exact-match execution accuracy.

Controlled Evaluation Protocol. To isolate core logical reasoning from peripheral data science challenges, we use a controlled setting where both paradigms receive identical inputs: explicit DDL schemas, natural language questions, and external knowledge. Schema representation is kept identical across conditions so that the sole source of systematic variation is the output paradigm. For Pandas generation, we apply prompt-level constraints to support end-to-end executabilit. This matched-input design is intended to attribute performance differences to reasoning capability rather than information asymmetry. Full prompt templates are in Appendix F. An ablation validating the DDL-based input

Model	Reas.	Ver.	Pandas Acc. (%)				SQL Acc. (%)			
			Sim.	Mod.	Hard	Tot.	Sim.	Mod.	Hard	Tot.
StarCoder2 (7B)	✗	✓	1.95	0.86	0.00	1.12	0.97	1.08	0.00	0.91
		✗	1.62	0.43	0.00	0.85	0.65	0.86	0.69	0.72
AutoCoder (7B)	✗	✓	8.65	4.31	2.07	5.45	13.84	3.88	7.59	10.23
		✗	7.50	3.88	1.72	4.88	12.54	3.23	6.90	9.19
TableGPT2 (7B)	✗	✓	25.46	17.24	13.10	19.82	40.97	25.00	18.62	34.03
		✗	23.15	15.09	11.03	17.45	39.68	22.84	18.62	32.59
Qwen2.5-Coder (7B)	✗	✓	41.91	26.51	28.28	35.95	61.41	42.67	31.72	52.93
		✗	39.89	18.97	23.45	31.94	60.65	41.16	31.03	51.96
Qwen3 (7B)	✓	✓	59.46	45.04	39.31	53.19	68.98	52.92	38.19	61.22
		✗	54.70	36.21	34.48	47.20	62.15	46.00	32.64	54.48
Qwen3 (14B)	✓	✓	54.70	37.93	37.24	47.98	65.84	48.28	42.76	58.34
		✗	50.27	32.33	29.66	42.89	59.46	44.18	39.31	52.93
Qwen2.5-Coder (14B)	✗	✓	60.76	45.69	40.69	54.30	63.78	51.29	50.34	58.74
		✗	51.24	33.84	35.86	44.52	63.24	46.55	44.14	56.39
Qwen2.5-Coder (32B)	✗	✓	62.16	49.78	42.76	56.58	66.70	50.65	41.38	59.45
		✗	58.38	41.81	37.93	51.43	64.11	47.84	39.31	56.84
Qwen3-Coder (30B)	✗	✓	49.51	41.59	35.86	45.83	67.35	55.82	43.45	61.60
		✗	46.92	34.05	32.41	41.53	64.11	51.29	43.54	58.28
Qwen3 (32B)	✓	✓	65.33	51.94	54.48	60.10	69.08	54.09	40.00	61.80
		✗	60.22	42.24	45.52	53.32	61.84	47.84	35.17	55.08
Qwen-Max	✗	✓	65.73	53.66	52.41	60.82	68.32	51.72	40.00	59.20
		✗	59.57	45.04	42.76	53.59	63.14	45.47	34.48	55.08
Qwen3-Coder-Plus	✗	✓	65.95	55.82	**53.10**	61.60	65.30	**56.25**	50.34	61.15
		✗	60.32	45.26	42.07	53.98	62.92	53.45	50.34	58.87
DeepSeek-R1	✓	✓	**69.95**	53.02	45.52	62.52	67.54	53.48	**57.93**	62.32
		✗	62.38	45.04	48.28	55.74	62.81	44.40	40.00	55.08
Qwen3-Max	✓	✓	67.89	**59.91**	46.21	**63.43**	**69.41**	55.82	46.21	**63.10**
		✗	62.27	43.75	47.59	55.28	65.51	52.59	52.41	60.37

Table 1: Single-execution accuracy on the BIRD-Pandas benchmark and original BIRD benchmark. "✓" denotes our *Verified* dataset; "✗" denotes the original BIRD set.

choice is provided in Appendix G. Detailed token usage and inference cost statistics are in Appendix H.

4.2 BASELINES

We evaluate a diverse set of open-source and closed-source LLMs under two deployment settings. All open-source models are deployed locally on a server equipped with six NVIDIA A6000 GPUs, using a temperature of 0.001 and a top-p of 0.95. Our baseline selection includes: (1) StarCoder2 (Lozhkov et al., 2024) (7B), AutoCoder (Lei et al., 2024) (7B), and TableGPT2 (Su et al., 2024) (7B), evaluated locally; (2) the Qwen family, covering the Qwen-Coder-2.5 series (Hui et al., 2024), the Qwen3 series (Yang et al., 2025) (including both reasoning and coder variants), and the latest closed-source Qwen3-Max, all accessed via the Alibaba Cloud API; and (3) DeepSeek-R1 (Guo et al., 2025), accessed via its official API endpoint.

4.3 MAIN RESULTS

Table 1 presents the accuracy of Text-to-SQL and Text-to-Pandas models across varying model scales. Under the Verified setting, Text-to-Pandas trails Text-to-SQL at smaller and mid-scale models. For Qwen2.5-Coder, the SQL-Pandas accuracy gap decreases from 16.98 percentage points at the 7B scale to 2.87 percentage points at the 32B scale, indicating that the performance advantage of SQL over Pandas code generation diminishes as model capacity increases. Reasoning-augmented models, including Qwen3-Max and DeepSeek-R1, reduce the cross-paradigm gap to

Model	Setting	Pandas Acc. (%)				SQL Acc. (%)			
		Sim.	Mod.	Hard	Tot.	Sim.	Mod.	Hard	Tot.
Qwen3-7B	LCF	**79.46**	**61.64**	**48.97**	**71.19**	**80.65**	**60.99**	**45.52**	**71.38**
	Standard	59.46	45.04	39.31	53.19	68.98	52.92	38.19	61.22
Qwen3-32B	LCF	**78.16**	**63.15**	**66.21**	**72.49**	**80.54**	**62.93**	**54.48**	**72.75**
	Standard	65.33	51.94	54.48	60.10	69.08	54.09	40.00	61.80
Qwen3-Max	LCF	**77.84**	**66.16**	**66.90**	**73.21**	**85.95**	**72.63**	**66.21**	**80.05**
	Standard	67.89	59.91	46.21	63.43	69.41	55.82	46.21	63.10

Table 2: Single-execution accuracy with and without LCF on the verified BIRD-Pandas dataset.

within 2 percentage points, suggesting that enhanced reasoning capability helps address both task-specification insufficiency and execution-semantic divergence. Beyond scale effects, cross-paradigm transferability differs between model types: specialized task-specific models such as AutoCoder and TableGPT2 perform well on SQL but exhibit larger accuracy drops on Pandas compared to general-purpose models such as Qwen3 and DeepSeek-R1, which maintain more balanced accuracy across the two paradigms. Comparing the Verified and Original settings further reveals the effect of ground truth reliability on evaluation. Leading models, such as Qwen3-Max and DeepSeek-R1, improve accuracy by 8.15 and 6.78 percentage points, respectively, on the Verified set, indicating that corrected ground truth reduces annotation-induced variance in performance measurement.

4.4 Error Analysis

We analyze execution failures using the fine-grained categories defined in our framework. Figure 5 summarizes the distribution of labels within each paradigm. Detailed definitions are provided in Appendix I. At the fine-grained level, *Filter Condition Errors* are the most frequent category in both SQL (32.7%) and Pandas (31.8%). Under our taxonomy, this label captures incorrect predicate specification, including wrong comparison values, wrong operators, and omitted conditions. This recurring pattern suggests that translating user constraints into executable conditions remains a shared difficulty across paradigms. Beyond this common pattern, the secondary error profile differs by paradigm. In SQL, *Row Selection* (21.2%) and *Column Selection* (16.6%) are the next most frequent categories, suggesting that many SQL failures arise from selecting the wrong records or fields. In Pandas, *Logic Errors* (17.5%) are comparatively frequent, indicating that Pandas failures more often involve the choice of computation strategy rather than isolated predicate or projection mistakes. However, these labels describe which decision went wrong, e.g., which table, rows, or filter condition, not why: the same error could stem from the task description leaving that decision underspecified, from the paradigm defaulting to a different treatment of it, or from other contributing factors, and error labels alone cannot distinguish between them.

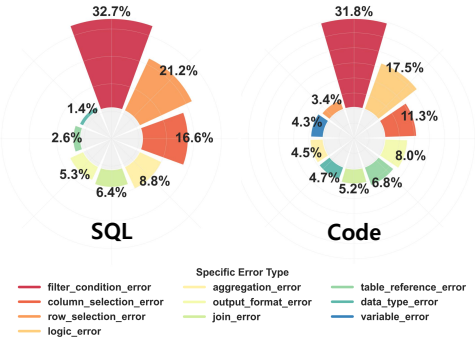


Figure 5: Distribution of execution error.

4.5 LCF Evaluation

To quantify the contribution of task-specification insufficiency, we apply LCF to the verified BIRD-Pandas dataset. We evaluate the top-performing Qwen3 models at three scales: 7B, 32B, and Max, selected from Table 1. Using a single model family controls for architectural differences across scales, so that LCF gains reflect clarification efficacy rather than family-specific biases. LCF provides brief, business-logic–level clarifications.

Phase (Qwen3-Max)	SQL	Pandas
Phase 1: Identification	69.3%	76.9%
Task-specification insufficiency	69.3%	55.0%
Execution-semantic divergence	0.0%	21.9%
Phase 2: Communication	1.0%	0.5%
Phase 3: Code generation	29.7%	22.6%

Table 3: Decomposition of post-LCF residual failures by phase and subtype.

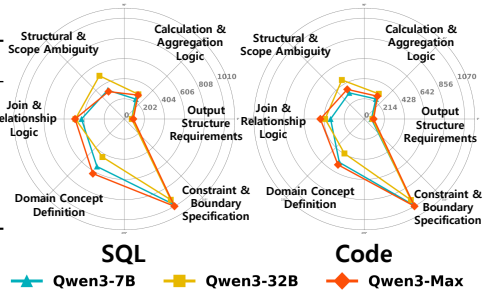


Figure 6: Distribution of inquiry categories.

4.5.1 LCF Effectiveness Results

Table 2 presents a performance comparison between the Standard and LCF settings. Clarification yields accuracy gains across evaluated models and paradigms, closing the gap between paradigms at the 7B and 32B scales. At the largest evaluated scale, SQL and Pandas achieve statistically comparable aggregate accuracy, yet a residual gap emerges under LCF. This pattern indicates that task-specification insufficiency accounts for a substantial share of the cross-paradigm difference at these scales. At the largest evaluated scale, SQL and Pandas achieve statistically comparable aggregate accuracy, yet the gap observed after clarification reflects the combined effect of execution-semantic divergence and any clarification limitations, including cases where models do not surface all consequential ambiguities.

4.5.2 Analysis of Post-LCF Failures

To quantify the composition of the gap observed after LCF clarification, we classify Qwen3-Max post-LCF failures according to the three-phase LCF taxonomy. Phase 1 (identification) splits into two subtypes: task-specification insufficiency, where required task information was not identified or surfaced, and execution-semantic divergence, where Pandas API defaults diverge from SQL execution semantics. Phase 2 covers clarification communication errors. Phase 3 covers code generation errors under correct clarification. Table 3 reports these subtypes as a share of each paradigm's own remaining errors, where task-specification insufficiency remains prominent on both sides while execution-semantic divergence appears only on the Pandas side. This suggests that single-turn model-generated questioning does not fully surface all missing information, regardless of paradigm.

4.5.3 Analysis of Clarifying Questions

Figure 6 shows the distribution of inquiry categories as classified by the Oracle in Phase 2. The most frequent category is "Constraint & Boundary Specification" and the least frequent is "Output Structure Requirements." Notably, larger models such as Qwen3-Max pose clarifying questions in similar proportions to smaller models, suggesting that task information ambiguity rather than syntactic complexity is the primary source of confusion in clarifying questions.

5 Conclusion

This work introduces the BIRD-Pandas benchmark and the LCF to quantify the relative contribution of task-specification insufficiency and execution-semantic divergence between paradigms. LCF closes the gap at intermediate scales. At the largest evaluated scale, SQL and Pandas achieve statistically comparable aggregate accuracy, yet SQL benefits more than Pandas from complete task specification. Analysis of post-LCF failures shows that task-specification insufficiency accounts for a substantial share of errors within both paradigms' own error sets, as models tend to overlook ambiguities with consequential impact on results when allowed to ask clarifying questions, while the net cross-paradigm gap is driven mainly by execution-semantic divergence, which appears exclusively on the Pandas side. Future work could explore multi-turn clarification mechanisms and verify generalizability across other code-generation paradigms.

AI USE STATEMENT

In this work, we used generative AI tools for the following tasks with required disclosure: (1) *Designing research ideas and experiments*: We used LLMs (Qwen3-238B, DeepSeek-R1, Qwen-Max) in a Model-in-the-Loop verification pipeline to identify annotation inconsistencies in the benchmark construction process (Section 3.2); (2) *Executing research*: We employed an LLM-based structure-agnostic validator ($\mathcal{V}_{logic}$, instantiated with Qwen3-Coder-Flash) following the LLM-as-a-Judge methodology to evaluate logical equivalence of execution outputs (Section 3.2); (3) *Generating datasets*: We used LLMs to assist with initial SQL-to-Pandas conversion and synthetic test case generation, with all outputs subsequently verified through double-blind expert review; (4) *Data annotation*: We used Gemini 3.1 Pro to produce an initial classification of post-LCF failures, which was subsequently validated through independent human annotation with 96.2% consistency (Appendix J). Additionally, we used generative AI tools for refining writing. All LLM-assisted work was reviewed through systematic human verification: LLM-generated code and annotations underwent double-blind expert review with documented inter-annotator agreement rates (96.7% for the validator audit; Section 3.2), and all 370 execution-semantic divergence cases were manually resolved by domain experts. We take full responsibility for the final content of this work, including all artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This research focuses on technical evaluation of code generation systems using publicly available benchmark data. Our work does not involve human subjects, personally identifiable information, or sensitive data.

REPRODUCIBILITY STATEMENT

To support reproducibility, we provide resources at `https://anonymous.4open.science/r/Bird_Pandas-3F82/`. In addition, we also provide: (1) Prompt templates and LCF pipeline specifications (Appendices F and E); (2) The structure-agnostic validator ($\mathcal{V}_{logic}$) prompt (Appendix D); (3) Error taxonomy and case studies for all 370 execution-semantic divergence instances (Appendices B and C). The complete code for dataset processing, model evaluation is available in the anonymous repository.

REFERENCES

Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. Rsl-sql: Robust schema linking in text-to-sql generation. *arXiv preprint arXiv:2411.00073*, 2024.

Saumya Chaturvedi, Aman Chadha, and Laurent Bindschaedler. Sql-of-thought: Multi-agentic text-to-sql with guided error correction. *arXiv preprint arXiv:2509.00581*, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Zhongjun Ding, Yin Lin, Tianjing Zeng, Rong Zhu, Bolin Ding, and Jingren Zhou. Ambisql: Interactive ambiguity detection and resolution for text-to-sql. In *Companion of the International Conference on Management of Data*, SIGMOD Companion, pp. 26–29, Bengaluru, India, 2026. ACM. doi: 10.1145/3626246.3654685.

Alexander David Egg, Martin Iglesias Goyanes, Andreu Mora, Friso H. Kingma, Thomas Wolf, and Leandro Von Werra. Dabstep: Data agent benchmark for multi-step reasoning. In *Advances in Neural Information Processing Systems 38: Datasets and Benchmarks Track*, 2025. arXiv preprint arXiv:2506.23719.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025. doi: 10.1038/s41586-025-09422-z.

Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. Putting pandas in a box. In *Proceedings of the Conference on Innovative Data Systems Research*, 2021.

Chen Huang, Yang Deng, Wenqiang Lei, Jiancheng Lv, Tat-Seng Chua, and Jimmy Xiangji Huang. How to enable effective cooperation between humans and nlp models: A survey of principles, formalizations, and beyond. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 466–488, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.22.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Nan Huo, Xiaohan Xu, Jinyang Li, Per Jacobsson, Shipei Lin, Bowen Qin, Binyuan Hui, Xiaolong Li, Ge Qu, Shuzheng Si, Linheng Han, Edward Alexander, Xintong Zhu, Rui Qin, Ruihan Yu, Yiyao Jin, Feige Zhou, Weihao Zhong, Yun Chen, Hongyu Liu, Chenhao Ma, Fatma Ozcan, Yannis Papakonstantinou, and Reynold Cheng. BIRD-INTERACT: Re-imagining text-to-SQL evaluation via lens of dynamic interactions. In *Proceedings of the Fourteenth International Conference on Learning Representations*, 2026.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024. doi: 10.1145/3672456.

Heegyu Kim, Jeon Taeyang, SeungHwan Choi, Seungtaek Choi, and Hyunsouk Cho. 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)*, pp. 4448–4475, Albuquerque, New Mexico, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.naacl-long.228.

Yilun Kong, Jingqing Ruan, Yihong Chen, Bin Zhang, Tianpeng Bao, Shiwei Shi, Guoqing Du, Xiaoru Hu, Hangyu Mao, Ziyue Li, Xingyu Zeng, Rui Zhao, and Xueqian Wang. Tptu-v2: Boosting task planning and tool usage of large language model-based agents in real-world industry systems. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 371–385, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-industry.27.

Bin Lei, Yuchen Li, and Qiuwu Chen. Autocoder: Enhancing code large language model with aiev-instruct. *arXiv preprint arXiv:2405.14906*, 2024.

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 Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *Proceedings of the Thirteenth International Conference on Learning Representations*, 2025.

Chunhui Li, Yifan Wang, Zhen Wu, Zhen Yu, Fei Zhao, Shujian Huang, and Xinyu Dai. Multisql: A schema-integrated context-dependent text2sql dataset with diverse sql operations. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 13857–13867, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.823.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23, 2025. doi: 10.1145/3690635.

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. 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, 2023.

Fang Liu, Yang Liu, Lin Shi, Zhen Yang, Li Zhang, Xiaoli Lian, Zhongqi Li, and Ma Yuchi. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Ziyang Luo, Kaixin Li, Hongzhan Lin, Yuchen Tian, Mohan Kankanhalli, and Jing Ma. Tree-of-evolution: Tree-structured instruction evolution for code generation in large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 297–316, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.14.

Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. The death of schema linking? text-to-sql in the age of well-reasoned language models. In *Table Representation Learning Workshop at NeurIPS 2024*, 2024. Oral presentation.

Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. An empirical study of the non-determinism of chatgpt in code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–28, 2025. doi: 10.1145/3697010.

Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. Towards scalable dataframe systems. *Proceedings of the VLDB Endowment*, 13(11):2033–2046, 2020. doi: 10.14778/3407790.3407807.

Mohammadreza Pourreza and Davood Rafiei. Evaluating cross-domain text-to-sql models and benchmarks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 1601–1611. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.99.

Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, Fei Huang, and Yongbin Li. A survey on text-to-sql parsing: Concepts, methods, and future directions. *arXiv preprint arXiv:2208.13629*, 2022.

Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Guoqing Du, Shiwei Shi, Hangyu Mao, Ziyue Li, Xingyu Zeng, and Rui Zhao. Tptu: Task planning and tool usage of large language model-based ai agents. In *NeurIPS 2023 Workshop on Foundation Models for Decision Making*, 2023.

AmirHossein Safdarian, Milad Mohammadi, Ehsan Jahanbakhsh Bashirloo, Mona Shahamat Naderi, and Heshaam Faili. Schemagraphsql: Efficient schema linking with pathfinding graph algorithms for text-to-sql on large-scale databases. In *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 2585–2599, Rabat, Morocco, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-eacl.134.

Aofeng Su, Aowen Wang, Chao Ye, Chen Zhou, Ga Zhang, Gang Chen, Guangcheng Zhu, Haobo Wang, Haokai Xu, Hao Chen, et al. Tablegpt2: A large multimodal model with tabular data integration. *arXiv preprint arXiv:2411.02059*, 2024.

Guanghu Sui, Zhishuai Li, Ziyue Li, Sun Yang, Jingqing Ruan, Hangyu Mao, and Rui Zhao. Reboost large language model-based text-to-sql, text-to-python, and text-to-function–with real applications in traffic domain. *arXiv preprint arXiv:2310.18752*, 2023.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. Mac-sql: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 540–557, Abu Dhabi, UAE, 2025. Association for Computational Linguistics.

Yifan Wu. Is a dataframe just a table? In *10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019)*, volume 76 of *Open Access Series in Informatics*, pp. 6:1–6:10. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi: 10.4230/OASIcs.PLATEAU.2019.6.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

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. 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*, pp. 3911–3921. Association for Computational Linguistics, 2018. doi: 10.18653/v1/D18-1425.

Bin Zhang, Yuxiao Ye, Guoqing Du, Xiaoru Hu, Zhishuai Li, Chi Harold Liu, Zhiwei Xu, Guoliang Fan, Rui Zhao, Ziyue Li, and Hangyu Mao. Sqlbench: A comprehensive evaluation for text-to-sql capabilities of large language models. *arXiv preprint arXiv:2403.02951*, 2024.

Qinggang Zhang, Hao Chen, Junnan Dong, Shengyuan Chen, Feiran Huang, and Xiao Huang. Structure-guided large language models for text-to-sql generation. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 74671–74691, Vancouver, Canada, 2025. PMLR.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46595–46623, 2023.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.

Yuqi Zhu, Ge Li, Xue Jiang, Jia Li, Hong Mei, Zhi Jin, and Yihong Dong. Uncertainty-guided chain-of-thought for code generation with llms. *arXiv preprint arXiv:2503.15341*, 2025.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *Proceedings of the Thirteenth International Conference on Learning Representations*, 2025.

A LIMITATIONS

This work has five main limitations. First, LCF provides clarification only when the model requests it, so the task-specification insufficiency subtype (Phase 1) may be underestimated if the model fails to identify or formulate the relevant question; consequently, the execution-semantic divergence share (21.9%; Table 3) represents a diagnostic upper bound rather than a clean causal decomposition. Second, the protocol compares SQL over a database against Pandas with CSV loading, introducing data-interface asymmetry; some failures may arise from CSV type inference and format handling rather than logical program generation. Third, the benchmark derives reference answer semantics from verified SQL execution results to enable paired comparison. This design choice reflects BIRD's SQL-centric ground truth and SQL's unified DBMS execution semantics, but it means that cross-paradigm equivalence is defined relative to SQL behavior rather than evaluating each paradigm's native idiomatic patterns independently. Task-specification insufficiency and execution-semantic divergence are framed within this paired-comparison context. Fourth, boundary cases such as tied ordering (Table 9) can reflect task underspecification rather than paradigm-specific requirements, and the classification depends partly on annotation judgment grounded in verified SQL semantics. Fifth, benchmark verification accepted 1,193 instances automatically based on model consensus (Section 3.2) without task-intent audit, which may reinforce rather than detect reference errors; divergent cases received double-blind expert review but consistent cases did not undergo random sampling audit. Future work could provide exhaustive clarifications independent of model questions, match data-loading interfaces across paradigms, and audit task intent on a random sample of accepted cases.

Case Study: Structural Alignment & Join Logic
Natural Language Query: Of the schools that offers a magnet program serving a grade span of Kindergarten to 8th grade, how many offers Multiple Provision Types? List the number of cities that offers a Kindergarten to 8th grade span and indicate how many schools are there serving such grade span for each city.
Original SQL: `SELECT T2.City, COUNT(T2.CDSCode) FROM frpm AS T1 INNER JOIN schools AS T2 ON T1.CDSCode = T2.CDSCode WHERE T2.Magnet = 1 AND T2.GSoffered = 'K-8' AND T1.`NSLP Provision Status` = 'Multiple Provision Types' GROUP BY T2.City` *Defect:* The use of `INNER JOIN` with the `WHERE` filter restricts the result set to cities containing schools with "Multiple Provision Types", excluding cities that satisfy the K-8 Magnet condition but lack this specific provision. Additionally, the query omits the calculation for the total school count per city.
Rectified SQL (Gold): `SELECT T2.City, COUNT(DISTINCT CASE WHEN T1.`NSLP Provision Status` = 'Multiple Provision Types' THEN T2.CDSCode END), COUNT(DISTINCT T2.CDSCode) FROM schools AS T2 LEFT JOIN frpm AS T1 ON T2.CDSCode = T1.CDSCode WHERE T2.GSoffered = 'K-8' AND T2.Magnet = 1 GROUP BY T2.City` *Correction:* Uses `LEFT JOIN` to retain all cities meeting the primary K-8 Magnet criteria. Applies conditional aggregation to calculate both the subset (Multiple Provision) and the total school count, addressing the dual-quantification requirement of the prompt.

Table 4: Comparison of Original and Rectified SQL for a complex aggregation task. The rectification resolves structural misalignment caused by incorrect filtering and join selection.

B Dataset Rectification Details and Case Studies

This appendix documents the rectification pipeline used to derive the verified benchmark and presents representative correction cases. Both paradigms are evaluated on matched underlying data through verified databases and their standardized CSV exports. The rectification process corrects annotation inconsistencies affecting execution-based evaluation.

We categorized the dataset corrections into four dimensions. First, *Structure & Schema* errors involve correcting column hallucinations and missing joins so that the SQL structure correctly reflects entity relationships. Second, *Data Values & Formats* refinements adjust filter conditions to match actual database values and explicitly handle NULLs, resolving literal mismatches between the query and the stored data. Third, *Logic & Business Rules* rectification restructures nested queries and set operations to resolve cases where the annotation incorrectly implemented the query's intended logic, for instance mistranslating a universal quantifier ("all bonds must be single") as an existential check ("at least one bond is single"). Fourth, *Formatting & Constraints* standardization adds missing `ORDER BY` and `LIMIT` clauses to enforce deterministic output ordering.

We present representative case studies for these four error categories. These examples show how the original SQL differs from the rectified SQL and how such discrepancies affect automated evaluation.

B.1 Category 1: Structure & Schema

The "Structure & Schema" category accounts for 35.5% of errors. It involves cases where the SQL structure does not match the logic required by the query, particularly regarding join logic (e.g., `INNER` vs. `LEFT` joins) and aggregation scope.

Table 4 shows a representative instance. The original annotation used an `INNER JOIN` combined with a `WHERE` clause. This structure removed cities that satisfied the primary condition (K-8 Magnet schools) but lacked the secondary attribute (Multiple Provision Types), and failed to include the "Total Schools" column. The rectified SQL uses a `LEFT JOIN` and conditional aggregation (`CASE WHEN`), retaining all relevant cities and calculating the required statistics.

Case Study: Data Consistency & Value Alignment
Natural Language Query: How many schools in merged Alameda have number of test takers less than 100?
Original SQL: `SELECT COUNT(T1.CDSCode) FROM schools AS T1 INNER JOIN satscores AS T2 ON T1.CDSCode = T2.cds WHERE T1.StatusType = 'Merged' AND T2.NumTstTakr < 100 AND T1.County = 'Lake'` *Defect:* The query filters by `'Lake'` instead of `'Alameda'`, using a literal value that does not appear in the prompt.
Rectified SQL (Gold): `SELECT COUNT(T1.CDSCode) FROM schools AS T1 INNER JOIN satscores AS T2 ON T1.CDSCode = T2.cds WHERE T1.StatusType = 'Merged' AND T2.NumTstTakr < 100 AND T1.County = 'Alameda'` *Correction:* Maps the token "Alameda" to the corresponding database value `'Alameda'`, so that the query filters the correct data partition.

Table 5: Comparison of Original and Rectified SQL demonstrating a Data Consistency error. The rectification resolves the mismatch between the user's constraint and the SQL literal.

B.2 Category 2: Data Values & Formats

The "Data Values & Formats" category (31.3%) involves cases where the SQL logic incorrectly maps natural language entities to database values. Common issues include incorrect literals, string matching errors, or format mismatches.

Table 5 presents an example of a value mismatch. The user queries for "Alameda" county, but the original SQL filters for "Lake", a value not present in the query. This error likely stems from incorrect entity mapping. The rectification process aligns the SQL literal with the information provided in the natural language hints.

B.3 Category 3: Logic & Business Rules

The "Logic & Business Rules" category (22.0%) covers errors where the SQL implementation does not match the query's business intent, particularly regarding conditional logic and scope. These errors often involve incorrect translation of natural language quantifiers (e.g., "all", "only") into SQL predicates like `NOT EXISTS` or `HAVING`.

Table 6 shows a "Quantification Scope" error. The user queries for "single bond molecules," implying that *all* bonds in the molecule must be of the single type. However, the original SQL uses a `JOIN` with a filter, which functions as an existential check (selecting molecules with *at least one* single bond). This logic incorrectly includes molecules with mixed bond types. The rectification uses `NOT EXISTS` to exclude molecules containing non-single bonds, satisfying the universal quantifier.

B.4 Category 4: Formatting & Constraints

The "Formatting & Constraints" category (11.2%) covers cases where the SQL logic is correct but the output formatting does not match requirements, such as column ordering or tie-breaking. These issues can lead to false negatives in automated evaluation.

Table 7 shows a "Column Ordering" error. The user requests the address components in the sequence: "Street, City, Zip and State". The original SQL retrieves the correct fields but swaps the last two, returning "State" before "Zip". This discrepancy affects evaluation accuracy. The rectification reorders the projection clause to match the requested sequence.

Case Study: Complex Logic & Quantification Scope

Natural Language Query:
Among the single bond molecule id, which molecules are not carcinogenic?

Original SQL:
```
SELECT DISTINCT T1.molecule_id FROM bond AS T1 INNER JOIN molecule AS T2
ON T1.molecule_id = T2.molecule_id WHERE T2.label = '-' AND T1.bond_type
= '-'
```
Defect: The query uses an existential filter (`bond_type = '-'`), selecting molecules with *at least one* single bond instead of those composed *exclusively* of single bonds.

Rectified SQL (Gold):
```
SELECT T2.molecule_id FROM molecule T2 WHERE T2.label = '-' AND NOT
EXISTS (SELECT 1 FROM bond WHERE molecule_id = T2.molecule_id AND
bond_type != '-')
```
Correction: Uses `NOT EXISTS` so that no non-single bonds exist for the molecule, matching the definition of a "single bond molecule."

Table 6: Comparison of Original and Rectified SQL illustrating a Complex Logic error. The rectification fixes the quantification scope from existential ("any") to universal ("all") using negative existential logic.

Case Study: Formatting & Output Determinism

Natural Language Query:
What is the complete address of the school with the lowest excellence rate? Indicate the Street, City, Zip and State.

Original SQL:
```
SELECT T2.Street, T2.City, T2.State, T2.Zip FROM satscores AS T1 INNER
JOIN schools AS T2 ON T1.cds = T2.CDSCode ORDER BY CAST(T1.NumGE1500 AS
REAL) / T1.NumTstTakr ASC LIMIT 1
```
Defect: The query projects columns in the order `(Street, City, State, Zip)`, inconsistent with the user instruction to present `Zip` before `State`.

Rectified SQL (Gold):
```
SELECT T2.Street, T2.City, T2.Zip, T2.State FROM satscores AS T1 INNER
JOIN schools AS T2 ON T1.cds = T2.CDSCode ORDER BY CAST(T1.NumGE1500 AS
REAL) / T1.NumTstTakr ASC LIMIT 1
```
Correction: Reorders the `SELECT` clause to match the requested output format `(Street, City, Zip, State)`.

Table 7: Comparison of Original and Rectified SQL demonstrating a Formatting error. The rectification fixes the column permutation to comply with the output order requirement.

C Execution-Semantic Divergence Case Studies

This appendix presents representative cases illustrating execution-semantic divergence during SQL-to-Pandas conversion. Each case demonstrates how identical task intent can produce different results under default execution behaviors of the two paradigms.

The cases below instantiate the main-text contrast between SQL and Pandas execution. In SQL, details such as NULL behavior, result ordering, and relational execution are mediated by the DBMS. In Pandas, equivalent behavior often depends on explicit parameter choices and intermediate operations. The case studies therefore focus on concrete points where execution can diverge even when the underlying task intent is aligned.

Representative Cases Table 8 illustrates null-value handling divergence. SQL `GROUP BY` preserves `NULL` keys by default, whereas Pandas `groupby` defaults to `dropna=True`. When the task requires retaining NULL-key groups, the Pandas generator must encode `dropna=False`, reflecting the additional decision logic required to match the specified task semantics.

Type	Content
Question	Name schools in Riverside where the average math score is greater than 400, and what is the funding type?

Gold SQL
`WHERE T2.District LIKE 'Riverside%'`
`GROUP BY T1.sname, T2.Funding`
`HAVING ... > 400`
	`SELECT T1.sname, T2.Funding FROM ...`

Prediction (Pandas)
`# Implicitly drops NaNs in grouping keys`
`grouped = df_merged.groupby(['sname', 'Funding'])`
`result = grouped['AvgScrMath'].mean()`
`# ... (subsequent filtering/projection omitted)`
	`df_merged = pd.merge(df_sat, df_frpm, ...)`

Table 8: A case study of execution mismatch between SQL and Pandas. While the SQL `GROUP BY` retains `NULL` keys, the generated Pandas code uses the default `groupby` which drops `NaN` values, causing valid records to be excluded.

Type	Content
Question	Which cities have the top 5 lowest enrollment number for students in grades 1 through 12?
Gold SQL **`ORDER BY SUM(T1.`Enrollment`) ASC LIMIT 5`** *Execution Result (Top 5):* Coulterville', 'Pinecrest', 'Shaver Lake', **'Emigrant Gap', 'Hyampom'**>Coulterville', 'Pinecrest', 'Shaver Lake', **'Emigrant Gap', 'Hyampom'**	`SELECT T2.City FROM ...`
Prediction (Pandas) `grouped = df_merged.groupby('City')['Enrollment'].sum()` **`df_sorted = grouped.sort_values(ascending=True).head(5)`** *Execution Result (Top 5):* Coulterville', 'Pinecrest', 'Shaver Lake', **'Hyampom', 'Woody'**>Coulterville', 'Pinecrest', 'Shaver Lake', **'Hyampom', 'Woody'** *(Note: 'Emigrant Gap' is excluded due to tie-breaking)*	`df_merged = pd.merge(...)`

Table 9: Example of Nondeterministic Sort Stability. Both SQL and Pandas correctly sorted by enrollment, but the absence of a secondary sort key caused divergent results at the cutoff point ($N = 5$) for tied values.

Table 9 demonstrates discrepancies arising from underspecified ordering criteria. With identical values at the limit boundary ($N = 5$) and no secondary sort key, the tied subset is not uniquely determined by the query. For paired evaluation, we derive the reference answer from the verified SQL execution result, which may reflect database-internal storage order. Matching this reference requires Pandas generators to infer additional sorting conditions beyond those stated in the query.

Table 10 illustrates type coercion divergence. CDSCode fields stored as text in the database may be read with leading-zero truncation when Pandas infers numeric types from the CSV export, causing join key mismatches that alter the filtered set and subsequent ranking.

Type	Content
Question	Which active district has the highest average score in Reading?

Gold SQL
 	
Result: **Palo Alto Unified**
	`SELECT T1.District FROM schools AS T1`
	`INNER JOIN satscores AS T2`
	`ON T1.CDSCode = T2.cds`
	`WHERE T1.StatusType = 'Active'`
	`ORDER BY T2.AvgScrRead DESC LIMIT 1`

Prediction (Pandas)
 	
Result: **Fremont Unified**
	`merged = schools.merge(satscores,`
	`  left_on='CDSCode', right_on='cds')`
	`active = merged[merged['StatusType']`
	`== 'Active']`
	`result = active.sort_values('AvgScrRead',`
	`  ascending=False).iloc[0]['District']`

Table 10: Example of type coercion divergence. CDSCode fields contain leading zeros stored as text in the SQLite database. When Pandas reads the CSV export, numeric inference may truncate leading zeros, causing join key mismatches that alter the filtered record set and subsequent ranking.

D Structure-Agnostic Validator Specifications

We evaluate executed outputs rather than generated programs, so SQL-specific program-level validators are not directly applicable in our setting. We also do not rely on hand-written output normalizers, because the comparison includes Pandas outputs whose answers may appear as scalars, lists, or tables, and may differ in ordering, numeric type, duplication, or lightweight structural wrappers while remaining logically equivalent. Covering these cases with rules would require many output-specific heuristics and different logic for SQL and Pandas. We therefore use one structure-agnostic judge that sees execution outputs only and applies the same criterion to both paradigms. Table 11 lists the prompt used for $\mathcal{V}_{logic}$.

E Logic Completion Framework (LCF) Specifications

This section documents the operational configuration, role assignments, and pipeline specifications used to reproduce LCF.

E.1 Diagnostic Scope and Experimental Role

LCF is a controlled diagnostic protocol designed to separate task-specification insufficiency from execution-semantic divergence. LCF shares a surface interaction pattern with interactive ambiguity-resolution settings such as AmbiSQL (Ding et al., 2026) and BIRD-INTERACT (Huo et al., 2026), but its role is fundamentally different. AmbiSQL studies fine-grained ambiguity resolution and automatic rewriting; BIRD-INTERACT evaluates stateful CRUD-oriented dialogue in an agentic loop. Both measure end-to-end task completion under realistic interaction dynamics. In contrast, LCF keeps the task fixed, restricts clarification to a single turn, and evaluates only the final generated program. This design reduces variation from feedback quality, user simulator fidelity, dialogue policy, and evolving task goals, allowing the contribution of task-specification insufficiency to be estimated as an isolated variable.

E.2 Role Configuration

Subject (Evaluated Model). The Subject performs both Phase 1 (formulating a clarifying question) and Phase 3 (generating the final program). Using the same model for both tasks keeps the evaluation

Structure-Agnostic Validator Instruction

System Role:
You are a professional output-equivalence validator. Decide whether Output 1 and Output 2 are logically and materially equivalent. Follow these rules strictly:

Core Equivalence Rules:
1. **Content Over Format:** Ignore data presentation (e.g., CSV vs. JSON, whitespace) and metadata (e.g., Pandas Index). Focus on the actual content.
2. **Structural Invariance:** Flatten nested structures. A scalar, a single-element list, and a 1xN array are equivalent if they contain the same value (e.g., ['apple'] == 'apple').
3. **Order Sensitivity:** If the ground truth query contains an ORDER BY clause or a LIMIT clause, ordering is required and the ranked sequence must match. Otherwise, compare outputs as unordered multisets regardless of row order.
4. **Type Normalization:** Normalize numeric types (1.0 == 1), booleans (True/yes/1), and date formats before comparison.
5. **Superset Validity:** If one output contains extra descriptive columns (labels) and the other is value-only, they are equivalent if the value projection matches exactly.
6. **Numeric Tolerance:** Numerically equivalent representations are equivalent (e.g., 1.0 == 1, $19.99 == $20.00 within rounding). Apply the same tolerance to values that are semantically identical but represented differently.
7. **De-duplication:** If one output contains duplicates and the other is unique, compare the sets of unique values.

Evaluation Task:
Ground Truth Query: {ground_truth_query}
Output 1: {predicted_output}
Output 2: {ground_truth}
Respond with exactly one word: "Correct" if equivalent, "Incorrect" otherwise.

Table 11: Prompt specifications for the Structure-Agnostic Validator ($\mathcal{V}_{logic}$). The system applies normalization rules (summarized above) to distinguish logical correctness from format variations.

within one system and aims to prevent confounds from model-specific questioning or generation styles.

Oracle (Domain-Expert Proxy). The Oracle (implemented via Qwen3-Max) receives both the verified ground truth and the Subject's clarifying questions, and returns brief natural-language clarifications. The Oracle's role is strictly diagnostic: it provides an upper-bound estimate of how much performance can be recovered when task-specification insufficiency is resolved correctly.

Oracle Scope Constraint. The Oracle provides task-level guidance (business requirements), which may include the specific column names or values needed to answer a question precisely, but withholds execution-level details (API calls, function parameters, code fragments). This constraint ensures LCF measures genuine recovery from task-specification insufficiency rather than artificial injection of execution decisions. This scoping also means the clarification's marginal utility differs across paradigms: business-logic information such as column names and filter values maps directly onto SQL's declarative syntax, whereas Pandas generation may still require resolving procedural sequencing and API-level defaults beyond what the Oracle specifies.

E.3 Pipeline Specification

The LCF pipeline consists of three sequential phases:

1. **Clarifying Question (Phase 1):** The Subject analyzes the input query and schema to identify task-specification insufficiency, then identifies all suspected information gaps and poses corresponding clarifying questions in a single turn (Q_{clar}).
2. **Clarification (Phase 2):** The Oracle processes Q_{clar} alongside the verified ground truth and returns a brief natural-language clarification (K_{task}) that answers the inquiry without leaking execution-level operations (e.g., API calls or code).
3. **Execution (Phase 3):** The Subject generates the final program (P_{pred}) conditioned on the original query augmented with K_{task}, using the same generation pipeline as the standard setting.

Phase 1 Prompt: Clarifying Question (System Instruction)
– As an expert Data Analyst, identify ambiguity in the following question regarding the data schema or task information. – Task: You are a logical analyst. Identify any ambiguity in the question regarding the database schema, task information, or output structure requirements (e.g., exact columns to return, handling of NULLs). Return ONLY a clarifying question that addresses the ambiguity. Do not generate SQL or Pandas code.

Table 12: System prompt for the Subject model in Phase 1 (Clarifying Question).

Phase 2 Prompt: Clarification (Oracle Instruction)
– Task: You are a Data Analyst Proxy. Based strictly on the Ground Truth provided below, answer the Model's Inquiry by explaining the task information or requirements used in the verified ground truth implementation. – Ground Truth: {GROUND_TRUTH_QUERY} – Model's Inquiry: {PHASE1_INQUIRY} Return your response in JSON format with "classification" and "answer". – Classification: Classify the Model's Inquiry into ONE or more of the following categories (use exact category names): [Domain Concept Definition]: Questions about what specific terms/concepts map to in the database schema [Calculation & Aggregation Logic]: Questions about formulas, ranking rules, or computational methods [Structural & Scope Ambiguity]: Questions about data scope, table relationships, or entity boundaries [Constraint & Boundary Specification]: Questions about filters, ranges, null handling, or boundary conditions [Join & Relationship Logic]: Questions about how tables connect or relate to each other [Output Structure Requirements]: Questions about exact columns to return, sorting order, or NULL handling in the output – Constraints: 1. Do NOT output code. 2. Extract the task information into implementation-agnostic natural language hints. Do not leak implementation details or syntactic tokens. 3. Answer ONLY what is explicitly asked in the Inquiry. 4. Explicitly mention specific Column Names and Values required.

Table 13: System prompt for the Oracle model in Phase 2 (Clarification).

The single-turn design is a deliberate methodological choice. Multi-turn interactive settings, while realistic, introduce several confounding variables: (i) **Feedback quality variability**: later turns may compensate for poor earlier responses, obscuring whether a single piece of information is sufficient; (ii) **Dialogue policy effects**: models may adopt different questioning strategies across turns, introducing variation in what information is requested; (iii) **Evolving task goals**: follow-up sub-tasks in multi-turn benchmarks shift the target, making it difficult to attribute performance changes to task-specification insufficiency alone.

E.4 LCF Prompt Templates

Table 12 and Table 13 present the system prompts for Phase 1 and Phase 2. Phase 3 uses the standard generation prompt (Appendix F) augmented with the clarification from Phase 2.

F Baseline Prompt Specifications

This appendix lists the prompt templates used in the baseline Text-to-SQL and Text-to-Pandas evaluations and records the implementation choices needed to reproduce the matched-input setup. Phase 3 of LCF uses the same Text-to-Pandas template.

Text-to-SQL Prompt. We follow the standard BIRD benchmark protocol. The system instruction and prompt template are defined in Table 14 and Table 15, respectively.

Text-to-Pandas Prompt. To evaluate explicit logical reasoning under matched inputs, we require a Pandas-based generation format. The prompt excludes SQL syntax and requires the use of

Text-to-SQL System Instruction
You are a SQL assistant. Only return the SQL query without any explanation.

Table 14: System instruction for Text-to-SQL generation.

Text-to-SQL Prompt Template
/* [Schema Info: DDL] */ CREATE TABLE ‘table_name‘ (...); /* ... – External Knowledge: {knowledge} – Additional Clarification (from Phase 2): {clarification} – Using valid SQLite and understanding External Knowledge, – answer the following questions for the tables provided above. – {question} SELECT

Table 15: Complete Text-to-SQL prompt template including schema information.

Text-to-Pandas System Instruction
You are an expert Pandas code generator specializing in pandas data analysis. Return runnable Pandas code only. No explanations or markdown. Strictly use ’import pandas as pd’ and pd.read_csv(’<table>.csv’). Do NOT use or mention SQL/SELECT/JOIN/CREATE/WHERE/etc. Do NOT define functions or classes (no ’def’, ’lambda’, ’class’).

Table 16: System instruction for Text-to-Pandas generation.

`pd.read_csv` to standardize data loading across examples. To ensure executable code without complex control flow, we prohibit user-defined functions and lambda expressions, requiring models to express conditional logic through vectorized operations (e.g., `np.where()`, boolean indexing, `.mask()`). The system instruction and prompt template are provided in Table 16 and Table 17.

Generated code undergoes standard preprocessing before execution: extraction from markdown code blocks when present, and verification that required imports and output statements are included as specified in the prompt. Execution proceeds in isolated per-instance environments with standard library timeouts. These implementation choices are consistent with established code generation evaluation protocols and do not introduce paradigm-specific advantages.

G Input Representation Ablation

To validate the matched-input design choice, we evaluate four Pandas input conditions while holding SQL fixed at DDL, using Qwen3-Max on the full evaluation set of 1,534 instances (Table 18).

Changing schema representation leaves Pandas accuracy unchanged. The raw CSV condition, which omits DDL entirely, achieves 63.75% against 63.43% for DDL only, and adding df.dtypes remains flat at 63.10%. If DDL were a systematic disadvantage for Pandas, these alternative representations should raise accuracy, yet they do not. The only condition that improves performance is DDL with df.head() sample rows at 66.43%, but this gain reflects exposure to actual data values rather than a fairer schema format, and providing such information to Pandas alone would break the matched-input control.

We chose DDL as the shared representation because it declares column types identically for both paradigms. When Pandas instead infers types from raw CSV, the inference can silently diverge from the declared schema. Table 19 illustrates this with real BIRD tables, where leading zeros are dropped, integer columns become float through NaN propagation, and date types are lost.

Text-to-Pandas Prompt Template

```
/* [Schema Info: Same as Text-to-SQL] */
...

# Task Description:
# Generate runnable pandas code only. No explanations,
# no markdown, no JSON.
# Requirements:
# 1) Use: import pandas as pd
# 2) Read tables strictly via pd.read_csv('<table>.csv')
# 3) Do NOT use or mention SQL/ SELECT/ JOIN/ CREATE/ WHERE/ etc.
# 4) Do NOT define functions or classes (no 'def', 'lambda', 'class')
# 5) Prefer clear variable names; keep code executable end-to-end
# 6) Use result to record the final result, and finally
# print(result) to print the final result.

# External Knowledge: {knowledge}
# Additional Clarification (from Phase 2): {clarification}
# Question: {question}
CODE
```

Table 17: Complete Text-to-Pandas prompt template with pandas-specific requirements. The "{clarification}" field shows where the Oracle's Phase 2 output is injected.

Pandas input condition	Accuracy	Correct
DDL only (paper setting)	63.43%	973/1534
DDL + df.dtypes	63.10%	968/1534
raw CSV (no DDL)	63.75%	978/1534
DDL + df.head() sample	66.43%	1019/1534

Table 18: Ablation of Pandas input representation. The df.head() condition exposes actual data values, which were not provided to SQL and would break matched-input control.

Column (table)	DDL type	CSV stores	Inferred dtype	Effect on Pandas
CDSCode (schools)	TEXT	"01100170000000"	int64	leading zero dropped, key becomes 1100170000000
NCESDist (schools)	TEXT	"0691051"	float64	leading zero dropped, value becomes 691051.0
AvgScrRead (satscores)	INTEGER	"" (null)	float64	integer column becomes float through NaN
OpenDate (schools)	DATE	"2005-08-29"	object	date type lost, read as plain text

Table 19: Type inference inconsistencies when Pandas reads raw CSV without DDL constraints. These silent divergences introduce mismatches in join keys and aggregation logic that reflect input inconsistency rather than generation ability.

Giving both paradigms the same DDL removes this input-level noise. The ablation supports the choice that DDL does not favor either paradigm, and the type-declaration consistency it provides helps isolate execution-semantic divergence from schema-representation artifacts.

H Detailed Efficiency and Inference Cost Statistics

To expand upon the efficiency analysis, Table 20 presents a detailed quantitative breakdown of token consumption and inference latency. These statistics are derived from executing 89 query instances of varying difficulty on the `california_schools` database, with all models evaluated via their respective API endpoints under unified inference configurations. Token counts reflect API-reported values from each provider's endpoint; prompt token counts are identical across models because all queries operate on the same database schema. Text-to-Pandas consistently uses more completion tokens than Text-to-SQL, because Pandas requires explicitly sequencing DataFrame operations (including intermediate states and data transformations), while SQL delegates physical

Model	Reason.	Text-to-SQL					Text-to-Pandas				
		Prompt	Comp.	Total	Reas.	Lat(s)	Prompt	Comp.	Total	Reas.	Lat(s)
Qwen2.5-Coder (7B)	✗	1055	36	1091	0	1.78	1155	110	1265	0	3.59
Qwen2.5-Coder (14B)	✗	1055	39	1094	0	1.89	1155	160	1315	0	4.79
Qwen2.5-Coder (32B)	✗	1055	39	1094	0	1.71	1155	105	1260	0	3.04
Qwen3 (7B)	✓	1055	39	1808	714	12.96	1155	81	2241	1005	16.80
Qwen3 (14B)	✓	1055	36	1502	411	13.38	1155	84	2470	1231	33.43
Qwen3 (32B)	✓	1055	40	1967	872	25.06	1155	90	2800	1555	57.38
Qwen3-Coder (30B)	✗	1055	40	1095	0	0.89	1155	86	1241	0	1.51
Qwen-Max	✗	1055	35	1090	0	3.35	1155	137	1292	0	14.71
Qwen3-Coder-Plus	✗	1055	42	1097	0	1.72	1155	97	1252	0	2.84
DeepSeek-R1	✓	1055	45	2852	1752	42.06	1155	109	3902	2638	59.25
Qwen3-Max	✓	1055	45	5925	4825	127.91	1155	104	5553	4294	120.12

Table 20: Token usage and latency per model for Text-to-SQL and Text-to-Pandas generation. Prompt, Comp., Total, Reason. denote prompt_tokens, completion_tokens, total_tokens, reasoning_tokens; Lat(s) denotes latency in seconds.

Specific error type	Classification basis	Main error manifestations
`table_reference_error`	Wrong data source chosen	Wrong table name or selection
`join_error`	Multiple tables connected incorrectly	Missing joins, wrong join type
`filter_condition_error`	Predicate doesn't match intent	Wrong operator or value
`row_selection_error`	Final records don't match	Extra or missing rows
`aggregation_error`	Grouping/aggregate incorrect	Wrong function or scope
`logic_error`	Overall computation strategy wrong	Wrong method or granularity
`data_type_error`	Type handling changes logic	Incorrect casting
`column_selection_error`	Fields don't match requested	Wrong or missing columns
`output_format_error`	Answer presented in wrong form	Incorrect structure or sorting
`variable_error`	Intermediate state mismanaged	Variable reference errors

Table 21: Fine-grained execution error taxonomy used for Figure 5.

execution planning to the DBMS. This longer token usage leads to higher latency for non-reasoning models on Pandas tasks. Reasoning models incur notably higher latency in both paradigms due to chain-of-thought generation.

I Execution Error Taxonomy

To support the failure analysis in Figure 5, we annotate each failed prediction with one or more fine-grained error labels. The taxonomy is organized around shared logical decisions in SQL and Pandas: choosing the correct data source and relational structure, specifying the correct constraints and answer scope, applying the required computation, and returning the requested answer form.

J Residual Gap Decomposition: Annotation Methodology

Section 4.5.2 reports the decomposition of the post-LCF residual gap into the three LCF phases, with Phase 1 further split into input-level and execution-semantic subtypes (Table 3). This appendix describes how those labels were obtained and validated.

An initial classification of Qwen3-Max post-LCF failures was produced by Gemini 3.1 Pro according to the three-phase LCF taxonomy. Two graduate students then independently labeled a stratified sample of 25 instances per paradigm per phase, plus all Phase 2 cases (n=6). Initial inter-annotator agreement reached $\kappa = 0.94$. All disagreements (n=6 instances, primarily on the Phase 1 subtype boundary) were discussed and resolved to consensus. Post-consensus labels showed 96.2% consistency with the initial Gemini 3.1 Pro classification, and Table 3 reports the distribution derived from that classification.

Model	Setting	SQL Acc.	Pandas Acc.	Gap	95% CI
Qwen3-32B	w/o Ver.	55.08%	53.32%	1.76 pp	[−1.77, 5.29]
Qwen3-32B	Verified	61.80%	60.10%	1.70 pp	[−1.75, 5.15]
Qwen3-32B	LCF	72.75%	72.49%	0.26 pp	[−2.90, 3.42]
Qwen3-Max	w/o Ver.	60.37%	55.28%	5.09 pp	[1.60, 8.58]
Qwen3-Max	Verified	63.10%	63.43%	−0.33 pp	[−3.74, 3.08]
Qwen3-Max	LCF	80.05%	73.21%	6.84 pp	[3.85, 9.83]

Table 22: Confidence intervals for SQL-minus-Pandas accuracy gaps. Intervals crossing zero indicate that the observed difference is consistent with random sampling variation and does not support a paradigm claim.

Subset	Agreement	Proportion	Wilson 95% CI
Overall	580/600	96.7%	[94.9%, 97.8%]
SQL instances	293/300	97.7%	[95.3%, 98.9%]
Pandas instances	287/300	95.7%	[92.7%, 97.5%]

Table 23: Human-validator agreement rates. The SQL and Pandas Wilson intervals overlap, indicating that the 2.0 pp difference is not statistically resolved.

K Statistical Validation of Performance Gaps

To assess whether reported performance differences reflect genuine paradigm distinctions rather than measurement noise, we provide three complementary lines of statistical evidence: confidence intervals for the gaps themselves, validator reliability audit results, and paired McNemar tests across all evaluation settings.

K.1 Confidence Intervals for Paradigm Gaps

Table 22 reports 95% confidence intervals for the SQL-minus-Pandas gap under all six model-setting combinations. We compute intervals for the gap directly rather than for individual accuracies, using a two-proportion normal approximation that treats the two paradigms as independent, which yields wider intervals than a paired approach and is therefore conservative.

The confidence intervals separate two cases. All three Qwen3-32B settings and the Qwen3-Max Verified setting have intervals that cross zero, indicating that the observed differences do not exceed sampling variation. Only two Qwen3-Max settings exclude zero: w/o Ver. at 5.09 pp [1.60, 8.58] and LCF at 6.84 pp [3.85, 9.83], with LCF showing the largest and cleanest gap once clarification noise is removed.

The independent two-proportion approximation is conservative; paired confidence intervals that account for within-instance correlation would be narrower. For the two significant gaps, Newcombe's paired score intervals are [1.12, 8.77] for w/o Ver. and [4.23, 9.36] for LCF, consistent with the independent intervals but tighter due to the positive correlation between paradigms (McNemar test contingencies in Table 24 show substantial shared success).

K.2 Validator Reliability Audit

The logical validator ($\mathcal{V}_{logic}$) applies one shared equivalence criterion to both paradigms. To verify that validator error does not account for reported gaps, we conducted a double-blind audit on a stratified sample of $N = 600$ instances (300 SQL, 300 Pandas), with two domain experts independently adjudicating each case. Table 23 reports the observed agreement rates.

The validator achieves 96.7% overall agreement with expert judgment. The difference between SQL agreement (97.7%) and Pandas agreement (95.7%) is 2.0 percentage points, but the two Wilson confidence intervals overlap, so this difference is not statistically distinguished. Even if all 2.0 pp of differential disagreement acted in the direction that widens the gap, it is far too small to produce the 6.84 pp LCF residual observed for Qwen3-Max.

Model	Setting	a	b	c	d	χ^2	p
Qwen3-32B	w/o Ver.	581	264	237	452	1.46	0.23
Qwen3-32B	Verified	692	256	230	356	1.39	0.24
Qwen3-32B	LCF	902	214	210	208	0.04	0.85
Qwen3-Max	w/o Ver.	656	270	192	416	13.17	2.9e-4
Qwen3-Max	Verified	749	219	224	342	0.06	0.81
Qwen3-Max	LCF	988	240	135	171	29.40	5.9e-8

Table 24: Paired McNemar contingency tables and test statistics. Cell notation: a = both correct, b = SQL only, c = Pandas only, d = both incorrect.

K.3 Paired McNemar Tests

Because both paradigms answer the same $n = 1{,}534$ instances, the comparison is paired. For each setting, we construct a 2×2 contingency table from question-level logs, where a denotes both correct, b denotes SQL only correct, c denotes Pandas only correct, and d denotes both incorrect. The McNemar test operates on the discordant cells b and c. Table 24 reports the results.

The paired tests are consistent with the confidence-interval reading. All Qwen3-32B gaps are non-significant ($p \geq 0.23$), and the Qwen3-Max Verified near-tie is non-significant ($p = 0.81$). The two settings with large observed gaps survive paired testing: Qwen3-Max LCF yields $\chi^2 = 29.4$ ($p < 1 \times 10^{-7}$), and Qwen3-Max w/o Ver. is significant at $\chi^2 = 13.2$ ($p < 1 \times 10^{-3}$).

The contingency table structure reveals how the two paradigms distribute success and failure across instances. For Qwen3-32B under LCF, the near-symmetric discordant cells ($b = 214$, $c = 210$) indicate that both paradigms handle the same set of instances with comparable reliability once task information is supplied. For Qwen3-Max under LCF, the asymmetry ($b = 240$, $c = 135$) indicates that SQL succeeds on 105 more instances than Pandas when both face identical clarified task specifications. This divergence concentrates in cases where execution-semantic differences manifest: Pandas requires explicit handling of API defaults such as null-key retention in grouping and deterministic tie-breaking in sorting, while SQL delegates these behaviors to the database engine. The substantial shared-success cell ($a = 988$) shows that most instances do not trigger paradigm-specific divergence, but the persistent imbalance in discordant outcomes isolates execution-semantic divergence as the distinguishing factor under complete task specification.