DPC: Training-Free Text-to-SQL Candidate Selection via Dual-Paradigm Consistency

Anonymous ACL submission

Abstract

While Large Language Models (LLMs) demonstrate impressive proficiency in generating SQL queries, they fundamentally lack the capability to self-evaluate correctness without an execution oracle. This limitation creates a stark ***Generation-Selection Gap***, where high potential accuracy (*Pass@K*) fails to translate into execution accuracy (*Pass@1*). Although supervised verifiers offer mitigation, they incur prohibitive annotation costs and suffer from domain fragility. Consequently, recent research has pivoted to the *training-free* setting. However, existing methods—such as Self-Consistency or LLM-as-a-Judge—remain hampered by *systematic bias* (consensus on hallucinations) and *symbolic blindness* (inability to simulate execution states). We introduce **DPC** (Dual-Paradigm Consistency), a multi-agent framework that reformulates SQL selection from a probabilistic guessing task on hidden data into a deterministic verification task on visible data. Specifically, **DPC** employs a SLICER and a TESTER agent to collaboratively construct a Minimal Distinguishing Database (MDD)—an adversarial, fully observable micro-environment engineered to expose logical discrepancies between candidates. To break the self-correction bias, a SOLVER agent then verifies the SQL candidates by cross-referencing their execution against a parallel Python/Pandas solution. By validating *execution consistency* between declarative (SQL) and imperative (Python) paradigms, **DPC** robustly discriminates correct logic from systematic hallucinations. Experiments on BIRD and Spider across multiple LLMs demonstrate that our method consistently outperforms existing selection baselines, achieving absolute accuracy improvements of up to 2.2% over strong competitors like Self-Consistency.

1 Introduction

Recent advances in Large Language Models (LLMs) have revolutionized the Text-to-SQL task, enabling systems to generate highly plausible SQL queries for complex questions (Li et al., 2023b; Yu et al., 2018; Zhu et al., 2025). However, due to the inherent stochasticity of LLMs, a single generation is often unreliable. Consequently, state-of-the-art approaches adopt a "***generate-then-select***" paradigm: generating K candidate queries to cover the correct logic (Li et al., 2025b,a). In this work, we focus on the critical downstream task: **SQL Candidate Selection**—the process of autonomously identifying the single correct executable query from a pool of K candidates.

Despite remarkable generative capabilities, a significant ***Generation-Selection Gap*** persists. As shown in Table 2, while the potential *Pass@K* is high (e.g., 58.8% on BIRD), ***real-world users demand a single correct answer***; yet, the actual *Pass@1* significantly lags behind (∼50%). Although *training-based verifiers* (Liu et al., 2025c) attempt to bridge this via fine-tuning, prohibitive annotation costs and poor generalization limit their utility. Consequently, focus has shifted to the **training-free** setting, yet current mechanisms remain inadequate. *Heuristic approaches* like Self-Consistency (Xie et al., 2025) fail under *systematic bias*, where models consistently converge on errors (Figure 1, Top-Left). Similarly, *LLM-as-a-Judge* (Lee et al., 2025) is hindered by *symbolic blindness*: constrained by limited data context and unable to mentally simulate execution, it relies on internal priors and exhibits *intrinsic judgement bias* (Figure 1, Top-Right).

We argue that the failure of current selection methods stems from three intrinsic challenges in the Text-to-SQL paradigm. First, **(C1) Partial Observability**: Real-world databases are typically too massive to fit within the context window. This forces the model to verify logic against a "hidden" data distribution ($\mathcal{D}_{hidden}$) based on probabilistic assumptions rather than concrete evidence, creating an epistemic gap that precludes rigorous verifica-

tion. Second, **(C2) Symbolic Blindness**: Even if the data were visible, LLMs lack an internal interpreter to reliably simulate the state changes of complex SQL operations (e.g., nested JOINs) merely by inspection. Finally, **(C3) Intrinsic Confirmation Bias**: When acting as a selector, the model inherently favors candidates that align with its internal priors—even if flawed. This results in a "conflict of interest" where the model fails to objectively distinguish between its own hallucinations and correct logic.

To address these challenges, we propose **DPC** (Dual-Paradigm Consistency), a multi-agent framework that shifts SQL selection from a probabilistic guessing task on hidden data to a *deterministic reasoning task on visible data*. **DPC** follows a progressive verification pipeline. First, to overcome **Partial Observability (C1)**, we introduce the concept of *Adversarial Environment Synthesis*. Instead of reasoning about the massive, invisible database, **DPC** employs a SLICER and a TESTER agent to construct a **Minimal Distinguishing Database (MDD)**. The MDD is not merely a data sample, but a context-fitting, adversarial micro-environment specifically engineered to yield divergent execution results for conflicting SQLs. This transforms the verification environment from partially observable to **Fully Observable**, enabling the model to ground its decisions in explicit execution evidence.

Second, to mitigate **Symbolic Blindness (C2)** and **Intrinsic Confirmation Bias (C3)**, we introduce **Dual-Paradigm Verification**. Rather than relying solely on SQL generation, we leverage the model's inherent *competency disparity* across programming languages. Extensive research indicates that LLMs exhibit superior reasoning capabilities in widespread imperative languages like Python, attributed to their dominance in pre-training corpora relative to domain-specific query languages (Twist et al., 2025; Lozhkov et al., 2024). Furthermore, translating a user's intent into imperative Python code forces the model to explicitly plan the data manipulation steps, offering a distinct reasoning path from declarative SQL formulation (Gao et al., 2023; Yang et al., 2025). Therefore, **DPC** employs a SOLVER agent to generate a parallel Python solution on the MDD. By treating the higher-confidence Python solution as a *proxy ground truth*, **DPC** pinpoints the correct SQL candidate that aligns with the imperative logic.

Our contributions are summarized as follows:

- We identify *Partial Observability* and *Systematic Bias* as the core constraints in SQL selection and propose the **Minimal Distinguishing Database (MDD)** to facilitate rigorous, fully observable, and adversarial verification.
- We introduce **DPC**, a training-free framework that leverages **Dual-Paradigm Consistency** to identify semantic correctness via complementary reasoning paths.
- We design a **Bipartite Soft-F1 (BS-F1)** metric to handle formatting heterogeneities and row-ordering ambiguity between SQL and Python results, ensuring rigorous alignment.
- Extensive experiments on BIRD and Spider benchmarks demonstrate that **DPC** consistently outperforms state-of-the-art training-free selection methods, establishing new standards in execution accuracy.

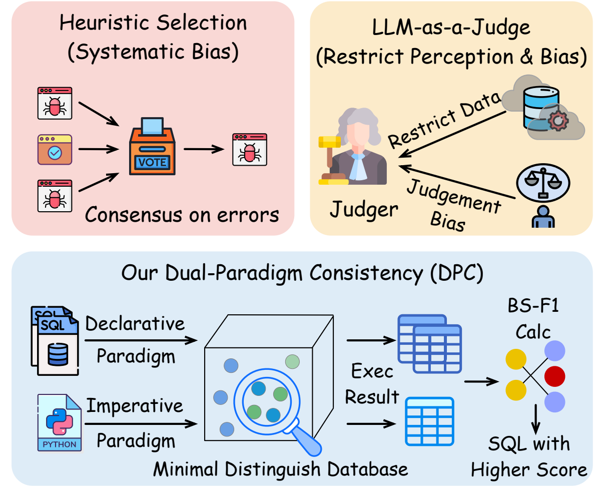

Figure 1: **Comparison of Selection Paradigms.** (Top-Left) *Heuristic Selection* relies on majority voting but fails when models exhibit **systematic bias**, reaching a consensus on errors. (Top-Right) *LLM-as-a-Judge* suffers from **restricted perception** due to the lack of execution feedback, relying solely on internal priors. (Bottom) Our **DPC** framework introduces a *Dual-Paradigm* approach (SQL & Python). It synthesizes a **Minimal Distinguishing Database** to execute both paradigms, determining correctness through deterministic result consistency rather than probability.

2 Problem Formulation

2.1 The SQL Selection Task

Let Q denote a natural language question and $S = (\mathcal{T}, \mathcal{C}, \mathcal{R})$ denote a relational database schema, where $\mathcal{T}$ is a set of tables, $\mathcal{C}$ is a set of columns, and $\mathcal{R}$ represents foreign key relations. Given Q and S, an LLM generates a set of K candidate SQL queries, denoted as $\mathcal{Y} = \{y_1, y_2, \dots, y_K\}$.

The objective of SQL Selection is to identify

the optimal candidate $y^* \in \mathcal{Y}$ that is semantically equivalent to the ground truth query y_{gt}. Semantic equivalence is defined by execution correctness on the database instance:

$$\textsc{Exec}(y^*, \mathcal{D}) = \textsc{Exec}(y_{gt}, \mathcal{D}) \quad (1)$$

where $\mathcal{D} = (S, \mathcal{V})$ represents the database instance, consisting of the schema S and the actual data values (records) $\mathcal{V}$.

In the **training-free** setting, we seek a selection function f that maximizes the likelihood of identifying y^* under fixed model parameters θ:

$$f^* = \underset{f}{\operatorname{argmax}}\, P\big(f(Q, S, \mathcal{Y}; \theta) \equiv y_{gt}\big) \quad (2)$$

subject to the constraint that f operates without access to the ground truth y_{gt} or gradient updates.

2.2 The Observability Gap

A critical constraint in real-world deployment is that the data instance is *partially observable*. Let W denote the context window capacity of the LLM. Typically, the full data values $\mathcal{V}$ far exceed this limit ($|\mathcal{V}| \gg W$). While standard prompting strategies may provide a small set of sample values $\mathcal{V}_{example} \subset \mathcal{V}$ (e.g., 3-shot row samples) to the model, the vast majority of the data distribution remains unseen, denoted as $\mathcal{V}_{unseen} = \mathcal{V} \setminus \mathcal{V}_{example}$.

This partial observability creates a *verification gap*. The provided $\mathcal{V}_{example}$ is typically static and sparse, lacking the *boundary cases* necessary to verify complex logic. For instance, distinguishing between `INNER JOIN` and `LEFT JOIN` requires specific records (e.g., unmatched keys) that are likely absent in a random sample. Consequently, execution consistency on the example subset is a necessary but insufficient condition for correctness:

$$\textsc{Exec}(y_i, \mathcal{D}_{example}) = \textsc{Exec}(y_{gt}, \mathcal{D}_{example}) \not\Rightarrow y_i \equiv y_{gt} \quad (3)$$

where $\mathcal{D}_{example} = (S, \mathcal{V}_{example})$. The model is thus forced to select y^* based on probabilistic priors rather than definitive execution evidence.

2.3 Objective Reformulation

Since verification on partial $\mathcal{D}_{example}$ is unreliable, we reformulate the selection task by constructing a **Minimal Distinguishing Database (MDD)**, denoted as $\mathcal{D}_{MDD} = (S', \mathcal{V}_{MDD})$. To serve as a valid verification ground, the MDD must satisfy two core properties: **(i) Contextual Feasibility**, requiring the synthetic instance to fit within the LLM's context window ($|\mathcal{D}_{MDD}| \leq W$); and **(ii) Discriminative Validity**, where $\mathcal{V}_{MDD}$ is adversarially engineered to expose semantic discrepancies. Formally, for any pair of semantically distinct candidates $y_i, y_j \in \mathcal{Y}$ (where $y_i \not\equiv y_j$), their execution on the MDD must diverge:

$$y_i \not\equiv y_j \implies \textsc{Exec}(y_i, \mathcal{D}_{MDD}) \neq \textsc{Exec}(y_j, \mathcal{D}_{MDD}) \quad (4)$$

Under these constraints, **DPC** transforms the task from probabilistic guessing on hidden data to deterministic consistency checking on visible data.

3 The DPC Framework

3.1 Framework Overview

DPC is a multi-agent system bridging the epistemic gap in SQL selection. As shown in Figure 2, it reformulates the task into a *progressive verification pipeline*, prioritizing external execution evidence over internal priors. It coordinates three agents (SLICER, TESTER, SOLVER) across four phases (see Appendix A for detailed prompts):

Stage 1: Candidate Clustering & Pairing. Instead of evaluating all candidates individually, **DPC** first condenses the selection space by clustering semantically identical queries. It then samples a *Champion-Challenger* pair to represent the most significant logical conflict, maximizing the efficiency of subsequent verification.

Stage 2: Adversarial Environment Synthesis. To resolve the conflict, the SLICER and TESTER agents collaborate to synthesize the **Minimal Distinguishing Database (MDD)**. Unlike random sampling, the MDD is adversarially generated to be *context-fitting* (small enough for the context window) yet *discriminative* (guaranteeing divergent execution results for conflicting logic).

Stage 3: Dual-Paradigm Execution. With the environment fully observable, **DPC** introduces a complementary reasoning path. The SOLVER agent generates an imperative Python script to solve the question on the MDD. This Python execution serves as a high-confidence *reference anchor* for verifying the declarative SQL candidates.

Stage 4: Consistency Verification. Finally, **DPC** determines the winner by comparing the SQL execution results against the Python reference. We employ a novel **Bipartite Soft-F1** metric to robustly quantify semantic equivalence, overcoming both formatting heterogeneities and row-ordering ambiguity inherent to cross-paradigm verification.

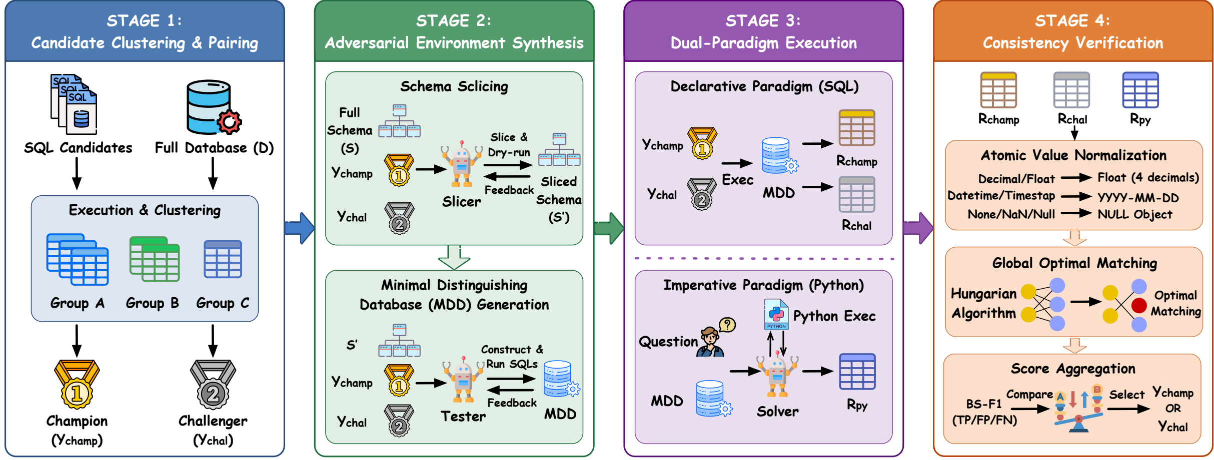

Figure 2: **Overview of the DPC framework.** The pipeline transforms selection into verification by identifying conflicting candidates (Champion vs. Challenger) and synthesizing an adversarial **Minimal Distinguishing Database (MDD)**. By executing candidates alongside a generated Python reference on this fully observable environment, DPC utilizes the **BS-F1** to deterministically identify the correct logic via cross-paradigm consistency.

3.2 Stage 1: Candidates Clustering & Pairing

The primary objective of this stage is to distill the noisy candidate set into two representative SQLs for verification. Since LLMs generate SQL queries probabilistically, the raw output often contains syntactic variations that are logically equivalent (e.g., varying alias names or keyword capitalization).

Execution-Consistency Clustering. We adopt an execution-based strategy to simplify the candidate set. Let $\mathcal{D}$ denote the original database instance available for execution. We execute every candidate $y_k \in \mathcal{Y}$ on $\mathcal{D}$ to obtain its execution result E_k. Candidates are then grouped into clusters $\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_m$ such that all queries within a cluster produce identical execution results:

$$y_a, y_b \in \mathcal{C}_i \iff \text{EXEC}(y_a, \mathcal{D}) = \text{EXEC}(y_b, \mathcal{D}) \quad (5)$$

Champion-Challenger Selection. To maximize the efficiency of the subsequent verification, we select the two most dominant SQLs for a pairwise duel (Li et al., 2025a; Sheng and Xu, 2025). We sort the clusters by size $|\mathcal{C}_i|$ in descending order to identify two candidates: the **Champion** (y_{champ}), selected from the largest cluster $\mathcal{C}_{max}$, which represents the model's "Majority Vote" choice and serves as a strong baseline; and the **Challenger** (y_{chal}), from the second-largest cluster, representing the most probable alternative hypothesis that conflicts with the consensus.

3.3 Stage 2: Adversarial Environment Synthesis

To resolve the conflict between the Champion (y_{champ}) and the Challenger (y_{chal}), DPC constructs a **Minimal Distinguishing Database (MDD)**. This synthetic environment is engineered to satisfy the contextual feasibility and discriminative validity defined in Section 2.3. See Appendix B for a detailed running example.

The SLICER Agent. The full database schema S often contains numerous tables and columns irrelevant to the current query, introducing noise and consuming context window space. The SLICER Agent iteratively distills a focused schema subgraph $S' \subseteq S$ that contains only the tables and columns necessary for the candidate SQLs.

To ensure the structural integrity of the sliced schema (e.g., ensuring no referenced columns or foreign keys are missing), we implement a *Dry-Run Validation* loop. In each iteration t, the agent predicts a schema candidate S'_t. We then attempt to execute (or EXPLAIN) both y_{champ} and y_{chal} on an empty database instance structured with S'_t. If execution fails, the database error message e_t is fed back to the agent for self-correction:

$$S'_{t+1} = \text{SLICER}(Q, S, \{y_{champ}, y_{chal}\}, e_t) \quad (6)$$

The process terminates upon dry-run success or reaching the maximum iteration limit T_{max}.

The TESTER Agent. Based on the validated slice S', the TESTER Agent constructs the synthetic data values $\mathcal{V}_{MDD}$ to populate the MDD. Unlike random data generation, this process is adversarial: the goal is to generate specific records that expose the semantic discrepancy between the two candidates.

We employ a *Discriminative Feedback* loop to guarantee effectiveness. In iteration t, the agent generates a data sample $\mathcal{V}_t$. We execute both candi-

dates on $\mathcal{D}_t = (S', \mathcal{V}_t)$ and compare their outputs. If the outputs are identical, it implies the data is insufficient to distinguish the logic (e.g., lacking a specific boundary case for a `JOIN`). The agent receives a penalty signal and regenerates the data:

$$\mathcal{V}_{t+1} = \text{TESTER}(Q, S', y_{champ}, y_{chal}, \mathbb{I}_{equal}) \quad (7)$$

where $\mathbb{I}_{equal}$ is the feedback indicator. The loop continues until the condition is met or the maximum iteration limit T_{max} is reached:

$$\text{EXEC}(y_{champ}, \mathcal{D}_{MDD}) \neq \text{EXEC}(y_{chal}, \mathcal{D}_{MDD}) \quad (8)$$

3.4 Stage 3: Dual-Paradigm Execution

With the fully observable MDD established, **DPC** performs dual-paradigm execution. We execute the candidate SQLs (y_{champ} and y_{chal}) on the $\mathcal{D}_{MDD}$ to obtain their deterministic outputs, denoted as E_{champ} and E_{chal} respectively. Simultaneously, to verify these results against an independent reasoning path, this stage employs a SOLVER Agent to solve the user question Q using Python (Pandas).

The Solver Agent. To establish a reliable verification standard, we leverage the model's *competency disparity* between paradigms. Imperative Python code benefits from superior pre-training coverage (Twist et al., 2025; Lozhkov et al., 2024) and explicit step-by-step logic derivation (Gao et al., 2023; Yang et al., 2025), offering a higher-confidence reasoning path than declarative SQL. Consequently, we treat the Python execution result as a robust *reference anchor* (proxy ground truth) to validate the SQL candidates.

To guarantee the executability of the generated logic, we implement a *Runtime Self-Correction* loop. The agent initially generates a Python script ρ_0 based on the schema and data in $\mathcal{D}_{MDD}$. In each iteration t, we execute ρ_t within a sandboxed Python environment. If the execution raises an exception (e.g., `KeyError`, `SyntaxError`), the traceback message e_t is captured and fed back to the agent to guide the debugging process:

$$\rho_{t+1} = \text{SOLVER}(Q, \mathcal{D}_{MDD}, \rho_t, e_t) \qquad (9)$$

This iterative refinement continues until the script executes successfully or the retry limit T_{max} is reached, yielding the final execution result E_{py}.

3.5 Stage 4: Consistency Verification

The final stage is to determine which candidate—the Champion or the Challenger—aligns with the verified Python reference anchor E_{py}.

Algorithm 1 Bipartite Soft-F1 (BS-F1) Calculation

Input: SQL Result E_{sql}, Python Proxy E_{py}
Output: Consistency Score $s \in [0, 1]$

1: **Preprocessing:**
2: $R_{sql} \leftarrow \text{NORMALIZE}(E_{sql})$
3: $R_{py} \leftarrow \text{NORMALIZE}(E_{py})$
4: **Global Optimal Matching:**
5: Let $N = |R_{sql}|$, $M = |R_{py}|$
6: Initialize Cost Matrix C of size $N \times M$
7: **for** $i \in 0 \dots N-1$, $j \in 0 \dots M-1$ **do**
8:     $m_{ij} \leftarrow$ column overlap ratio between $R_{sql}[i]$ and $R_{py}[j]$
9:     $C_{i,j} \leftarrow 1.0 - m_{ij}$ ▷ *Minimize cost ≡ Maximize overlap*
10: **end for**
11: ▷ *Solve Assignment Problem via Hungarian Algorithm*
12: $\mathcal{A} \leftarrow \text{HUNGARIANALGORITHM}(C)$
13: Initialize $TP, FP, FN \leftarrow 0, 0, 0$
14: **Aggregation:**
15: **for** assigned pair (i, j) in $\mathcal{A}$ **do**
16:     Accumulate TP, FP, FN based on match of $(R_{sql}[i], R_{py}[j])$
17: **end for**
18: ▷ *Handle Unmatched Rows (Penalties)*
19: $FP \mathrel{+}= (N - |\mathcal{A}|)$ ▷ *Unmatched SQL rows*
20: $FN \mathrel{+}= (M - |\mathcal{A}|)$ ▷ *Unmatched Python rows*
21: **Final Calculation:**
22: $P \leftarrow \frac{TP}{TP+FP}$, $\quad R \leftarrow \frac{TP}{TP+FN}$
23: **return** $2 \cdot \frac{P \cdot R}{P+R}$

The Challenge of Heterogeneity. In standard Text-to-SQL evaluation, **Execution Accuracy (EX)** checks for strict identity between result sets. However, this rigid metric fails in our cross-paradigm setting (E_{sql} vs. E_{py}) due to: **(i) Type Incompatibility:** SQL and Python/Pandas utilize different internal representations for semantically identical values (e.g., SQL `DECIMAL` vs. Python `float`, SQL `NULL` vs. Python `NaN`), causing standard EX to reject valid matches. **(ii) Ordering Ambiguity:** SQL query results without `ORDER BY` are unordered sets, whereas Pandas operations often produce implicitly ordered indices, leading to false negatives despite identical content.

To bridge this gap without introducing human intervention, we propose the **Bipartite Soft-F1 (BS-F1)**, a robust deterministic metric grounded in combinatorial optimization.

BS-F1 Calculation. As detailed in Algorithm 1, the BS-F1 metric addresses heterogeneity through a three-step pipeline: **(i) Atomic Value Normalization:** We apply type-agnostic mapping (Table 1) to unify diverse data types into a canonical format, resolving type incompatibilities. **(ii) Global Optimal Matching:** To handle ordering ambiguity, we model result alignment as a *Maximum Weight Bipartite Matching* problem. By constructing a

Table 1: Atomic Value Normalization used in DPC.

Original Type/Value	Normalized Target
`Decimal`, `float`	`float` (rounded to 4 decimals)
`datetime`, `Timestamp`	ISO String (`YYYY-MM-DD`)
`None`, `NaN`, `"null"`	None (Null Object)
String w/ whitespace	Stripped String

cost matrix of semantic distances between SQL and Python rows, we utilize the **Hungarian Algorithm** (Kuhn, 2010; Munkres, 1957; Crouse, 2016) to find the global optimal assignment maximizing column-level overlap. **(iii) Score Aggregation:** Based on the optimal assignment, we aggregate True Positives (TP) while unmatched rows (extra or missing data) strictly penalize the score as FP or FN. The harmonic mean of the resulting Precision and Recall yields the final BS-F1 score.

The candidate with the highest BS-F1 score is selected as the final prediction y^*:

$$y^* = \underset{y \in \{y_{champ}, y_{chal}\}}{\operatorname{argmax}} \text{BS-F1}(E_y, E_{py}) \quad (10)$$

By grounding verification in a metric that is robust to format heterogeneity yet sensitive to semantic logic, DPC effectively mitigates the intrinsic bias of LLM self-correction.

4 Experiments

We conduct comprehensive experiments to verify the effectiveness and efficiency of DPC, benchmarking it against both general-purpose base LLMs and state-of-the-art NL2SQL specialized solutions.

4.1 Experimental Settings

Datasets. We evaluate DPC on two benchmarks: **BIRD** (Li et al., 2023b), a large-scale dataset focusing on real-world database complexity and external knowledge, and **Spider** (Yu et al., 2018), the standard for evaluating SQL structural generalization. We use the official BIRD Mini-Dev split (500 queries) for efficient agentic reasoning evaluation and the standard Spider Test split (2,147 quries).

Base LLMs. To assess adaptability, we employ three backbones: **GPT-5** (OpenAI, 2025) as the frontier proprietary model, and **DeepSeek-V3.2** (Liu et al., 2025a) alongside **Qwen2.5-Coder-7B-Instruct** (Hui et al., 2024) as state-of-the-art open-source representatives.

Selection Baselines. We compare DPC against two categories of selection strategies: (1) **Heuristic-based**: *Random* selection; *Execution-Guided Selection* (Li et al., 2023a), which selects the first candidate that executes without errors; and *Self-Consistency (SC)* (Li et al., 2025b), which selects the SQL yielding the most frequent execution result. (2) **LLM-based**: *Multiple-Choice Selection (MCS)* (Lee et al., 2025), where an LLM is prompted to select the best candidate from the pool.

Integration with Text-to-SQL Systems. To demonstrate plug-and-play capability, we integrate DPC into: (1) **Prompting-based** systems, including *DAIL-SQL* (Gao et al., 2024) and *CHESS* (Talaei et al., 2024); and (2) **Fine-tuning-based** models, including *OmniSQL-7B* (Li et al., 2025c) and *XiYanCoder-7B* (Liu et al., 2025c).

Evaluation Metrics. We adopt a multi-dimensional protocol focusing on *Effectiveness* and *Efficiency*. We report **Execution Accuracy (EX)** (Li et al., 2023b) as the primary metric, alongside **Upper Bound (Pass@N)** (Pourreza et al., 2025) which represents the theoretical maximum performance within the candidate pool. For practical overhead, we measure the average **Token Cost** and **Latency** per query to assess the efficiency of the selection process.

Implementation Details. For local inference, we deploy **Qwen2.5-Coder-7B-Instruct**, **XiYanCoder-7B**, and **OmniSQL-7B** using the `vLLM` library (Kwon et al., 2023) on a single NVIDIA A800 (80GB) GPU. Meanwhile, **DeepSeek-V3.2** and **GPT-5** are accessed via their official APIs. Throughout the DPC process, we set the sampling temperature to 0.7 and the self-correction limit to $T = 3$.

4.2 Main Results and Analysis

DPC Consistently Outperforms Baselines. As shown in Table 2, DPC achieves state-of-the-art performance across all base LLMs on both BIRD and Spider. On the challenging BIRD benchmark, DPC significantly boosts the performance of open-source models; for instance, it improves the EX of **DeepSeek-V3.2** from 51.2% (Self-Consistency) to **53.4%**. Even for the frontier **GPT-5** model, which already exhibits high baseline accuracy, DPC manages to extract further gains, pushing its EX from 50.4% (Multiple-Choice Selection) to **51.2%**. Furthermore, compared to heuristic-based methods like Execution-Guided Selection, DPC demonstrates a superior ability to identify semantically correct SQLs that might otherwise be overlooked by simple execution checks. We provide a detailed case

Table 2: Comparison of different SQL selection methods using multiple base LLMs on BIRD and Spider datasets. **Upper Bound (Pass@N)** indicates the theoretical maximum accuracy within the candidate pool ($N = 5$). The best results among selection methods are **bolded**, and the second-best results are underlined.

Selection Methods	BIRD EX (%)			Spider EX (%)		
	Qwen	DeepSeek	GPT	Qwen	DeepSeek	GPT
Upper Bound (Pass@5)	*57.6*	*58.8*	*56.4*	*84.8*	*77.6*	*76.6*
Heuristic-based						
Random Selection (Baseline)	37.0	48.2	46.0	72.3	70.9	71.9
Execution-Guided Selection (Li et al., 2023a)	44.4	49.8	50.2	75.0	72.6	72.4
Self-Consistency (Li et al., 2025b)	46.4	51.2	49.4	76.5	71.6	72.2
LLM-based						
Multiple-Choice Selection (Lee et al., 2025)	43.6	51.0	50.4	74.8	72.8	72.9
Dual-Paradigm Consistency (Ours)	**47.6**	**53.4**	**51.2**	**77.5**	**73.2**	**73.3**

Table 3: Performance enhancement of `DPC` integrated into SOTA Text-to-SQL systems on BIRD ($N = 5$).

Method / Selection	Execution Accuracy (%)			
	Sim.	Mod.	Cha.	All
Prompting-based				
DAIL-SQL (Gao et al., 2024)				
+ SC	**64.2**	46.0	32.3	48.6
+ **DPC (Ours)**	62.9	**49.2**	**35.3**	**50.4**
CHESS (Talaei et al., 2024)				
+ SC	78.4	63.2	55.9	66.2
+ **DPC (Ours)**	**78.4**	**64.8**	**56.9**	**67.2**
Fine-tuning-based				
OmniSQL-7B (Li et al., 2025c)				
+ SC	64.8	43.6	40.2	49.2
+ **DPC (Ours)**	**68.9**	**44.0**	**41.2**	**50.8**
XiYanCoder-7B (Liu et al., 2025c)				
+ SC	70.9	49.2	36.3	53.0
+ **DPC (Ours)**	**71.6**	**52.0**	**40.2**	**55.4**

Table 4: Robustness analysis against systematic bias on BIRD using Qwen2.5-Coder-7B-Instruct.

Set Category	SC (%)	DPC (%)	Gain
Overall (Full Set)	46.4	47.6	+1.2
Majority-Correct	**100.0**	97.0	-3.0
Majority-Incorrect	0.0	**21.4**	**+21.4**

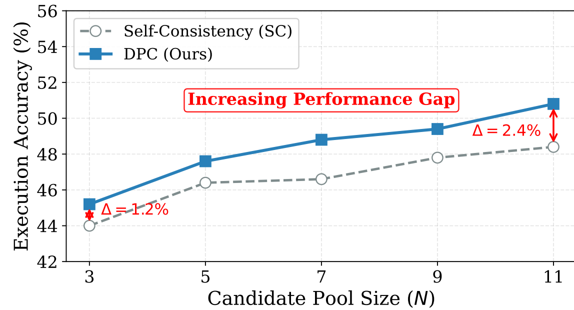

Figure 3: Impact of candidate pool size N on BIRD.

study comparing `DPC` with the Self-Consistency baseline in Appendix B.

Versatility in System Integration. Table 3 demonstrates `DPC` integrated into existing SOTA Text-to-SQL systems. To ensure a rigorous and cost-effective evaluation, we employ **Qwen2.5-Coder-7B-Instruct** as the uniform backbone for both the candidate generation in prompting-based systems and the logic execution in the `DPC` verification agent. We observe that `DPC` serves as a powerful "plug-and-play" enhancement layer across all tested systems. For prompting-based frameworks, it boosts **DAIL-SQL** and **CHESS** by **+1.8%** and **+1.0%** respectively. More importantly, for fine-tuning-based models like **XiYanCoder-7B**, `DPC` yields a significant improvement of **+2.4%**.

4.3 Robustness Analysis

Resilience to Systematic Bias. Traditional SC fails completely (**0%** accuracy) on the *Majority-Incorrect Set*, where model-internal biases lead to consistent but erroneous outputs (Table 4). In contrast, `DPC` recovers **21.4%** of these hard cases by leveraging cross-paradigm verification. Although a minor regression (-3.0%) occurs in the Majority-Correct set, the significant recovery in biased scenarios proves that `DPC` serves as a critical safety net when the majority consensus is flawed (see Appendix C for detailed error analysis).

Scaling with Candidate Size. As shown in Figure 3, increasing the candidate pool size ($N \in \{3, \dots, 11\}$) widens `DPC`'s lead over SC from **1.2%** to **2.4%**. This trend validates `DPC`'s superior scalability and proficiency in filtering high-confidence noise within dense search spaces.

Table 5: Efficiency and cost-benefit analysis on BIRD using Qwen2.5-Coder-7B-Instruct. Token cost and latency are reported as average values per question.

Selection Methods	EX (%)	Tokens (K)	Latency (s)
Heuristic-based			
Random Selection	37.0	-	∼0.0
Execution-Guided	44.4	-	∼0.3
Self-Consistency	46.4	-	∼0.8
LLM-based			
Multiple-Choice	43.6	∼4.2	∼5.1
DPC (Ours)	**47.6**	**∼3.8**	**∼4.2**

Table 6: Ablation study of DPC components on the BIRD dataset. The baseline is the full DPC framework equipped with Qwen2.5-Coder-7B-Instruct.

Components	EX (%)	Δ EX (%)
DPC (Full)	**47.6**	**–**
w/o Tester Agent	46.2	-1.4
w/o Slicer Agent	46.8	-0.8
w/o Python Agent	47.0	-0.6
w/o BS-F1 Metric	47.0	-0.6
w/o Self-Correction	47.2	-0.4

520 ### 4.4 Efficiency and Cost-Benefit Analysis

521 **Efficiency vs. LLM Baselines.** Compared to
522 MCS, DPC achieves higher accuracy (**+4.0%** EX)
523 with lower overhead, reducing token consumption
524 (4.2k→**3.8k**) and latency (5.1s→**4.2s**), as shown
525 in Table 5. This efficiency stems from the SLICER
526 Agent, which proactively prunes irrelevant schema,
527 avoiding the redundancy inherent in full-schema
528 selection methods.

529 **Accuracy-Efficiency Trade-off.** While heuristic
530 baselines (e.g., SC) are near-instantaneous, they
531 compromise precision. DPC outperforms SC by
532 **1.2%** on BIRD. In high-stakes domains (e.g., fi-
533 nance) where data integrity is paramount, the
534 marginal latency increase (∼3.6s) is a justifiable
535 investment for the substantial gain in reliability.

536 ### 4.5 Ablation Study

537 Table 6 dissects component-wise contributions on
538 the BIRD set. The TESTER Agent proves most crit-
539 ical (**-1.4%**), validating the necessity of adversarial
540 feedback for effective MDD construction, followed
541 by the SLICER Agent (**-0.8%**) which mitigates
542 schema noise. The remaining modules (SOLVER
543 Agent, BS-F1, and Self-Correction) collectively
544 contribute to the robustness, confirming the holistic
545 efficacy of the dual-paradigm design.

5 Related Works 546

Text-to-SQL Generation and Selection. Re- 547
cent prompting (Gao et al., 2024; Talaei et al., 548
2024) to fine-tuning (Liu et al., 2025c; Li et al., 549
2025c) systems achieved impressive potential ac- 550
curacy (Pass@K) on benchmarks like BIRD and 551
Spider (Liu et al., 2025b). However, a critical 552
Generation-Selection Gap persists: models often 553
struggle to identify correct candidates (Wang et al., 554
2025). To bridge this, training-based verifiers (Liu 555
et al., 2025c; Pourreza et al., 2025) employ dis- 556
criminative models but suffer from high annotation 557
costs and domain fragility. Conversely, training- 558
free heuristics like Self-Consistency (Xie et al., 559
2025) and LLM-as-a-Judge (Lee et al., 2025) are 560
hampered by *systematic bias* (consensus on errors) 561
and *Symbolic Blindness* (inability to verify execu- 562
tion). In contrast, DPC reformulates selection as a 563
deterministic verification task within an adversari- 564
ally synthesized environment. 565

Data Synthesis for Text-to-SQL. To mitigate 566
data scarcity (Hu et al., 2023), recent research 567
leverages LLMs to synthesize training corpora. 568
Frameworks like OmniSQL (Li et al., 2025c) and 569
SQL-Factory (Li et al., 2025d) prioritize scale via 570
multi-agent collaboration, while others like SQL- 571
Forge (Guo et al., 2025), SQLord (Cheng et al., 572
2025), and SING-SQL (Caferoglu et al., 2025) fo- 573
cus on reliability and domain adaptation through 574
"reverse generation" pipelines. However, these 575
methods target *offline training augmentation*, pro- 576
ducing static datasets that lack the *discriminative* 577
validity required for inference-time verification. In 578
contrast, DPC repurposes synthesis for *online verifi-* 579
cation, dynamically constructing a query-specific 580
Minimal Distinguishing Database (MDD) to ex- 581
pose precise semantic discrepancies. 582

6 Conclusion 583

In this paper, we introduce Dual-Paradigm Consis- 584
tency (DPC), a training-free framework that bridges 585
the generation-selection gap in Text-to-SQL. By 586
synthesizing an adversarial Minimal Distinguish- 587
ing Database (MDD) and leveraging the comple- 588
mentary reasoning of SQL and Python, DPC trans- 589
forms selection from probabilistic guessing into 590
deterministic verification. Experimental results 591
on BIRD and Spider demonstrate that DPC signifi- 592
cantly mitigates systematic bias and achieves state- 593
of-the-art execution accuracy. 594

Limitations

Despite significant gains, **DPC** incurs higher inference latency than direct generation due to its multi-agent synthesis and execution pipeline. Future work will explore *adaptive verification*—triggering DPC only under high uncertainty—to optimize efficiency. Additionally, under complex implicit constraints, synthesized environments may structurally diverge from real distributions; thus, improving the *semantic fidelity* of adversarial synthesis remains a critical direction for future exploration.

References

Hasan Alp Caferoglu, Mehmet Serhat Çelik, and Özgür Ulusoy. 2025. SING-SQL: A synthetic data generation framework for in-domain text-to-sql translation. *CoRR*, abs/2509.25672.

Song Cheng, Qiannan Cheng, Linbo Jin, Lei Yi, and Guannan Zhang. 2025. Sqlord: A robust enterprise text-to-sql solution via reverse data generation and workflow decomposition. In *Companion Proceedings of the ACM on Web Conference 2025, WWW 2025, Sydney, NSW, Australia, 28 April 2025 - 2 May 2025*, pages 919–923. ACM.

David Frederic Crouse. 2016. On implementing 2d rectangular assignment algorithms. *IEEE Trans. Aerosp. Electron. Syst.*, 52(4):1679–1696.

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. *Proc. VLDB Endow.*, 17(5):1132–1145.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: program-aided language models. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.

Yu Guo, Dong Jin, Shenghao Ye, Shuangwu Chen, Jianyang Jianyang, and Xiaobin Tan. 2025. Sqlforge: Synthesizing reliable and diverse data to enhance text-to-sql reasoning in llms. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 8441–8452. Association for Computational Linguistics.

Yiqun Hu, Yiyun Zhao, Jiarong Jiang, Wuwei Lan, Henghui Zhu, Anuj Chauhan, Alexander Hanbo Li, Lin Pan, Jun Wang, Chung-Wei Hang, Sheng Zhang, Jiang Guo, Mingwen Dong, Joseph Lilien, Patrick Ng, Zhiguo Wang, Vittorio Castelli, and Bing Xiang. 2023. Importance of synthesizing high-quality data for text-to-sql parsing. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 1327–1343. Association for Computational Linguistics.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

Harold W. Kuhn. 2010. The hungarian method for the assignment problem. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 29–47. Springer.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025. MCS-SQL: leveraging multiple prompts and multiple-choice selection for text-to-sql generation. In *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025*, pages 337–353. Association for Computational Linguistics.

Boyan Li, Chong Chen, Zhujun Xue, Yinan Mei, and Yuyu Luo. 2025a. Deepeye-sql: A software-engineering-inspired text-to-sql framework. *CoRR*, abs/2510.17586.

Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025b. Alpha-sql: Zero-shot text-to-sql using monte carlo tree search. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025c. Omnisql: Synthesizing high-quality text-to-sql data at scale. *Proc. VLDB Endow.*, 18(11):4695–4709.

Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023a. RESDSQL: decoupling schema linking and skeleton parsing for text-to-sql. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 13067–13075. AAAI Press.

Jiahui Li, Tongwang Wu, Yuren Mao, Yunjun Gao, Yajie Feng, and Huaizhong Liu. 2025d. Sql-factory: A

multi-agent framework for high-quality and large-scale SQL generation. *CoRR*, abs/2504.14837.

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 Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023b. 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 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025a. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025b. A survey of text-to-sql in the era of llms: Where are we, and where are we going? *IEEE Trans. Knowl. Data Eng.*, 37(10):5735–5754.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2025c. Xiyan-sql: A novel multi-generator framework for text-to-sql. *CoRR*, abs/2507.04701.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 38 others. 2024. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173.

James Munkres. 1957. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38.

OpenAI. 2025. GPT-5 System Card. Technical report, OpenAI. Technical report.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Ö. Arik. 2025. CHASE-SQL: multi-path reasoning and preference optimized candidate selection in text-to-sql. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Lei Sheng and Shuai-Shuai Xu. 2025. CSC-SQL: corrective self-consistency in text-to-sql via reinforcement learning. *CoRR*, abs/2505.13271.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHESS: contextual harnessing for efficient SQL synthesis. *CoRR*, abs/2405.16755.

Lukas Twist, Jie M. Zhang, Mark Harman, Don Syme, Joost Noppen, and Detlef D. Nauck. 2025. Llms love python: A study of llms' bias for programming languages and libraries. *CoRR*, abs/2503.17181.

Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, Mengdie Chu, Yingqi Gao, Xiang Qi, Peng Zhang, and Ying Yan. 2025. Agentar-scale-sql: Advancing text-to-sql through orchestrated test-time scaling. *CoRR*, abs/2509.24403.

Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment. *Proc. ACM Manag. Data*, 3(3):194:1–194:24.

Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, and Julian J. McAuley. 2025. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms. *CoRR*, abs/2502.19411.

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, Brussels, Belgium, October 31 - November 4, 2018*, pages 3911–3921. Association for Computational Linguistics.

Yizhang Zhu, Runzhi Jiang, Boyan Li, Nan Tang, and Yuyu Luo. 2025. Elliesql: Cost-efficient text-to-sql with complexity-aware routing. *CoRR*, abs/2503.22402.

A Prompts for DPC Agents

To facilitate reproducibility and provide transparency into the **DPC** framework, we present the detailed prompt templates used for the three core agents: SLICER, TESTER, and SOLVER. Each agent operates with a distinct system prompt that defines its specific role, reasoning instructions, and strict output formats (e.g., JSON or Python code), ensuring robust communication and error handling within the multi-agent pipeline.

A.1 Prompts for SLICER Agent

The SLICER Agent performs *schema pruning* to extract a Minimal Schema Slice, identifying only the tables and columns strictly necessary for the subsequent verification pipeline.

System Prompt: SLICER Agent

```
You are a database expert. Your task is to
     identify the minimum, non-duplicate set of
     tables and columns required to execute the
     provided Candidate SQL Queries.

You should:
1. Analyze the provided Candidate SQL Queries.
2. Identify all tables and columns from the Full
     Database Schema that are actually used in
     these SQLs (SELECT, JOIN, WHERE, GROUP BY,
     etc.).
3. Ensure you only use table and column names
     exactly as they appear in the Full Database
     Schema.
4. Provide a concise thinking process before the
     final result.

Your output MUST follow this format:
<thinking>
[Your step-by-step analysis here]
</thinking>
<result>
{
    "relevant_schema": [
        {
            "table": "table_name",
            "columns": ["column1", "column2"]
        },
        ...
    ]
}
</result>
IMPORTANT: The content inside <result> MUST be a
     valid, standard JSON string that can be
     parsed by `json.loads()`. DO NOT include any
      comments (like // or /* */) or extra text
     inside the JSON block.
```

User Prompt Template: SLICER Agent

```
Full Database Schema:
{full_schema}

Candidate SQL Queries:
{candidate_sqls}

Please identify the relevant tables and columns
     used in the Candidate SQL Queries, following
      the output format defined in the system
     prompt.
```

Retry Prompt Template: SLICER Agent

```
The previously identified Schema Slice has issues
     . Error details:
---
{error_message}
---

Please analyze the error (it could be a format
     issue, missing tables/columns, or incorrect
     names) and provide the corrected, complete
     Schema Slice.
Ensure your response strictly follows the output
     format defined in the system prompt (
     including <thinking> and <result> tags).
The content inside <result> MUST be a valid,
     standard JSON string without any comments.
```

A.2 Prompts for TESTER Agent

The TESTER Agent is responsible for generating adversarial data (Minimal Distinguishing Database) to expose logical discrepancies.

System Prompt: TESTER Agent

```
You are a database QA engineer. Your task is to
     generate a minimal set of synthetic test
     data (rows) that will cause two different
     SQL queries to return DIFFERENT results.

You should:
1. Analyze the Natural Language Question and the
     Candidate SQLs.
2. Identify the logical difference between SQL 1
     and SQL 2 (e.g., a filter condition, a join
     type, or an aggregation).
3. Generate a sufficient but minimal set of data
     that specifically triggers this logical
     difference.
4. Ensure the data adheres to the Sliced Database
      Schema (correct table/column names, types,
     and foreign key relationships).
5. Leverage metadata in the Schema: Use column
     descriptions, value descriptions, and
     example values provided in the schema to
     ensure the generated test data is realistic
     and follows the expected data distribution/
     format of the original database.
6. Provide a concise thinking process before the
     final result.

Your output MUST follow this format:
<thinking>
[Your analysis of why the SQLs differ and how
     your data will expose that]
</thinking>
<result>
{
    "test_data": {
        "table_name1": [
            {"column1": value1, "column2": value2
        },
            ...
        ],
        "table_name2": [...]
    }
}
</result>
IMPORTANT: The content inside <result> MUST be a
     valid, standard JSON string that can be
     parsed by `json.loads()`. DO NOT include any
      comments (like // or /* */) or extra text
     inside the JSON block.
```

User Prompt Template: TESTER Agent

```
Sliced Database Schema:
{sliced_schema}

Natural Language Question: {question}
{evidence_str}
```

```
Candidate SQL 1 (Champion):
{sql_1}

Candidate SQL 2 (Challenger):
{sql_2}

Please generate the test data that makes SQL 1
     and SQL 2 yield different results, following
      the output format defined in the system
     prompt.
```

Retry Prompt Template: TESTER Agent

```
The previously generated test data has issues or
     is INEFFECTIVE. Error details:
---
{error_message}
---

Please analyze the logic difference between SQL 1
      and SQL 2 again, and provide a corrected
     set of test data that yields DIFFERENT
     results.
Ensure your response strictly follows the output
     format defined in the system prompt (
     including <thinking> and <result> tags).
The content inside <result> MUST be a valid,
     standard JSON string without any comments.
```

A.3 Prompts for SOLVER Agent

The SOLVER Agent functions as the reference anchor, executing Python logic on the synthesized MDD to determine the correct result.

System Prompt: SOLVER Agent

```
You are a Python data scientist expert in Pandas.
      Your task is to write a Python script to
     answer a natural language question based on
     provided database tables (loaded as Pandas
     DataFrames).

You should:
1. Analyze the schema and the provided test data.
2. Use the provided DataFrames (already available
      in the namespace with their table names).
3. Write clean, efficient Pandas code to compute
     the answer.
4. IMPORTANT: Store the final result in a
     variable named 'result'.
5. The 'result' MUST ALWAYS be a pandas DataFrame
     . Even for single values or lists, wrap them
      in a DataFrame.
6. COLUMN SELECTION: The final DataFrame MUST
     ONLY contain columns that are explicitly
     asked for in the question. Do not include
     extra or redundant columns.
7. COLUMN ORDERING: The order of columns in the
     final DataFrame MUST strictly follow the
     order mentioned in the natural language
     question.

Your output MUST follow this format:
<thinking>
[Your step-by-step logic for solving the problem
     using Pandas]
</thinking>
<result>
[Your Python code here]
</result>
```

User Prompt Template: SOLVER Agent

```
{test_data_with_types}

### Database Relationships (PK/FK):
{relationships}

### Available DataFrames (Pandas Variables):
{df_names}

### Natural Language Question:
{question}
{evidence_str}

Please write the Pandas code to solve the
     question.
Remember:
- Only return columns explicitly asked for in the
      question.
- The column order must match the question's
     order.
- Ensure the final answer is stored in the `
     result` variable as a DataFrame.
```

Retry Prompt Template: SOLVER Agent

```
The previously generated code has issues. Error
     details:
---
{error_message}
---

Please analyze the error and provide the
     corrected Python code.
Ensure your response strictly follows the output
     format defined in the system prompt (
     including <thinking> and <result> tags).
```

B Case Study: Identifying Missing Constraints

We present a detailed case study (QID: 1500, Source: *BIRD Mini-Dev Split*) to demonstrate how **DPC** leverages dual-paradigm execution to detect missing logical constraints (e.g., missing JOINs) that heuristic methods often overlook.

B.1 The Conflict: Missing Table Constraint

Question: *"Please list the product description of the products consumed in September, 2013."*

The baseline SC method selects a **Champion SQL** that relies solely on the `transactions_1k` table, assuming the transaction date is sufficient. In contrast, **DPC** selects a **Challenger SQL** that introduces an additional JOIN with the `yearmonth` table, implying that a valid consumption record requires a corresponding monthly entry.

Step 1: Logical Divergence

Champion SQL (Incorrect):

```
SELECT p.Description
FROM transactions_1k
     t
JOIN products p ON
     ...
WHERE t.Date LIKE '
     201309%'
```

$\rightarrow$ *MISSING constraint: 'yearmonth' table.*

Challenger SQL (Correct):

```
SELECT p.Description
FROM products p
JOIN transactions_1k
     t ...
JOIN yearmonth ym ON
     ...
WHERE ym.Date LIKE '
     201309%'
```

$\rightarrow$ *ENFORCES constraint: 'yearmonth' table.*

B.2 Adversarial Synthesis (MDD)

The TESTER agent identifies that the Champion SQL is a *superset* of the Challenger. To expose the error, it constructs a "trap" scenario in the *Minimal Distinguishing Database* (MDD):

1. It creates a valid transaction for **Customer 100** in September 2013 (Product: 'LPG').
2. Crucially, it **omits** Customer 100 from the `yearmonth` table for that period.

This data distribution creates a decisive split:

- **Champion Result:** Returns `['LPG']` (Matches transaction date).
- **Challenger Result:** Returns `Empty` (Fails INNER JOIN with `yearmonth`).

Table 7: **Synthesized MDD.** The agent creates a "Ghost Transaction" (Row 1) that lacks a parent record in the YearMonth table (Bottom).

Table: transactions_1k

TxID	CustID	Date	Product
1	**100**	20130915	**LPG (ID:2)**

Table: yearmonth

CustID	Date
200	201309
300	201309
– Cust 100 is Missing! –	

B.3 Dual-Paradigm Verification

The SOLVER agent executes the verification logic using Pandas. As shown below, the Python agent explicitly merges the `yearmonth` dataframe, validating the structural necessity of this table.

Step 3: Python Logic Alignment

```
# Solver Agent's Logic:
# 1. Filter yearmonth table first.
# Note: Python explicitly uses the yearmonth
    constraint.
mask = yearmonth['Date'].str.startswith('
    201309')
filtered_ym = yearmonth[mask]

# 2. explicit MERGE acts as a filter (Inner
    Join)
# This line kills the result because Customer
     100 is NOT in filtered_ym
merged_tx = pd.merge(filtered_ym,
    transactions_1k, on='CustomerID')

result = pd.merge(merged_tx, products, ...)[[
    'Description']]
```

Selection Result: The Python execution returns an **Empty DataFrame**, perfectly aligning with the Challenger SQL's result. **DPC** determines that the Champion SQL hallucinated a simplification (ignoring the `yearmonth` table) that contradicts the cross-paradigm consensus. Thus, **DPC** correctly identifies the missing constraint and selects the Challenger SQL.

C Error Analysis

To deliver a thorough analysis, we present a systematic error analysis comparing our **DPC** framework against the **Self-Consistency (SC)** baseline, based on execution results from the BIRD Mini-Dev split. We first introduce our error taxonomy used throughout the analysis, then provide a statistical overview of error disagreements, followed by detailed case studies illustrating how **DPC** 's dual-paradigm verification resolves or fails to resolve specific error types.

C.1 Error Taxonomy

We adopt a two-level error taxonomy to categorize discrepancies between **DPC** and SC, adapted from prior studies on Text-to-SQL errors. Errors are classified into two primary categories:

1. **Semantic-Level Errors (A)** Arise from misunderstandings of query semantics, such as incorrect filtering, aggregation, ordering, or result representation. These errors often stem from natural language ambiguity or misinterpretation of query intent.
2. **Structural/Schema-Level Errors (B)** Involve incorrect schema linking, join path selection, or table/column references. These errors reflect failures in mapping the question to the underlying database structure.

A detailed breakdown of error subcategories, along with representative examples, is provided in Appendix Table 8. Based on our validation on the BIRD Mini-Dev split, among the cases where **DPC** succeeds while SC fails, three error subtypes dominate the distribution, as follows: **Result Representation Error (A5)** constitutes **41.7%** of the observed failures, followed by **Schema Linking Error (B3)** at **25%**, and **Predicate Misalignment Filter (A1)** accounting for **16.7%**.

Table 8: Error Taxonomy for DPC vs. SC Comparison

Category	Code	Error Type	Description
Semantic (A)	A1	Predicate Misalignment	Incorrect filtering logic, value hallucination, misuse of operators (e.g., LIKE, IN, BETWEEN), or logical connectors (AND/OR).
	A2	Aggregation Misuse	Misapplication of aggregate functions (COUNT, SUM, AVG, etc.).
	A3	Grouping Error	Missing or incorrect GROUP BY columns.
	A4	Ordering & Superlative	Incorrect ORDER BY or LIMIT usage, especially in queries involving "most," "least," or ranking.
	A5	Result Format Error	Issues in the SELECT clause, including wrong data types, missing or extra columns, and formatting errors (e.g., rounding, casting).
Structural (B)	B1	Wrong Join Path	Incorrect table joins, including missing, extra, or incorrect tables.
	B2	Join Condition Error	Incorrect ON conditions in join operations.
	B3	Schema Linking Error	Correct semantic understanding but failure in composing multi-table logic, often due to incorrect column or table mapping.

C.2 Case Studies: Semantic-Level Errors

C.2.1 Predicate Misalignment

(QID: 1464 from BIRD Mini-Dev Split)

SC SQL (Incorrect):

```
SELECT m.first_name,
    m.last_name, i.
    amount
FROM member AS m
JOIN income AS i ON m
    .member_id = i.
    link_to_member
WHERE i.date_received
    = '9/9/2019';
```

→ *INCORRECT predicate value: uses colloquial date format not supported by the database schema.*

DPC SQL (Correct):

```
SELECT T2.first_name,
    T2.last_name,
    T1.amount
FROM income AS T1
JOIN member AS T2 ON
    T1.
    link_to_member =
    T2.member_id
WHERE STRFTIME('%Y-%m
    -%d', T1.
    date_received) =
    '2019-09-09';
```

→ *CORRECT predicate grounding: enforces ISO date format consistent with schema constraints.*

Predicate Misalignment (Value Hallucination). Case ID 1464 illustrates a representative predicate-level error where SC relies on natural language priors to generate a colloquial date literal ('9/9/2019') that does not conform to the database's expected format, resulting in execution failure despite correct intent detection.

Schema-Grounded Correction. DPC successfully aligns the predicate value with schema constraints by explicitly normalizing the date representation to the ISO format ('2019-09-09'), ensuring compatibility with the underlying database.

Robust Literal Alignment. This case demonstrates that effective text-to-SQL generation requires grounding literal value predictions in database-specific formatting rules rather than surface-level textual patterns, highlighting DPC 's advantage over SC in handling **A1 Predicate Misalignment** errors.

C.2.2 Result Representation Error

(QID: 227 from BIRD Mini-Dev Split)

SC SQL (Incorrect):

```
SELECT CAST(SUM(CASE
    WHEN label = '+'
     THEN 1 ELSE 0
    END) AS REAL)
      * 100 / COUNT
         (*) AS
         percent
FROM molecule;
```

→ *INCORRECT result representation: missing* ROUND(..., 3)*, does not satisfy the required precision.*

DPC SQL (Correct):

```
SELECT ROUND(
    CAST(SUM(CASE
         WHEN label =
          '+' THEN 1
         ELSE 0 END)
         AS REAL)
    * 100 / COUNT(
         molecule_id)
         ,
     3
) AS percent
FROM molecule;
```

→ *CORRECT result representation: explicit casting and rounding ensure alignment with the question.*

Result Representation Awareness. In this case, SC correctly captures the underlying computation for estimating the proportion of carcinogenic molecules, but fails to align with the question's output specification by omitting explicit precision control (i.e., ROUND(..., 3)).

Question-Aligned Refinement. DPC refines the SELECT clause by enforcing both numeric type casting and result formatting, ensuring that the returned value strictly adheres to the requirement of three decimal places.

Consistency Selection. By favoring SQL candidates whose execution results match the question-defined output format, DPC successfully corrects result-level mismatches on top of a logically correct foundation. This demonstrates its advantage over SC in handling **A5 Result Representation Errors**.

C.3 Case Studies: Structural/Schema-Level Errors

C.3.1 Wrong Join Path

(QID: 1340 from BIRD Mini-Dev Split)

SC SQL (Incorrect):

```
SELECT
SUM(CASE WHEN SUBSTR(
      T1.event_date,
      1, 4)
= '2019' THEN T2.
      spent ELSE 0 END
      ) -
SUM(CASE WHEN SUBSTR(
      T1.event_date,
      1, 4)
= '2020' THEN T2.
      spent ELSE 0 END
      ) AS difference
FROM event AS T1
INNER JOIN
budget AS T2 ON T1.
      event_id = T2.
      link_to_event
INNER JOIN
expense AS T3 ON T2.
      budget_id = T3.
      link_to_budget
WHERE T1.status = '
      Student_Club'
```

→*Incorrect Join Path: Utilized unnecessary extra tables for redundant filtering.*

Challenger SQL (Correct):

```
SELECT SUM(CASE WHEN
SUBSTR(T1.event_date,
       1, 4)
= '2019' THEN T2.
      spent ELSE 0 END
      )
- SUM(CASE WHEN
SUBSTR(T1.event_date,
       1, 4)
= '2020' THEN T2.
      spent ELSE 0 END
      )
AS difference
FROM event AS T1
JOIN budget AS T2
ON T1.event_id = T2.
      link_to_event
```

→ *CORRECT Join Path: Understand the natural language query and identifying the appropriate tables.*

Adversarial Environment Synthesis DPC constructs a Minimal Distinguishing Database (MDD) tailored to expose logical discrepancies between candidate queries. The MDD is engineered to be small enough to fit within the model's context window yet rich enough to differentiate between semantically distinct SQLs. This allows DPC to execute both candidates in a fully observable, controlled environment.

Dual-Paradigm Verification Instead of relying solely on SQL execution or internal model priors, DPC leverages a Python/Pandas reference generated by the Solver agent. The Python solution, derived from the same MDD, serves as a high-confidence proxy ground truth. By comparing the execution results of the SQL candidates against this independent imperative reference, DPC can objectively determine which SQL aligns with the correct logic.

Avoidance of Systematic Bias Unlike SC, which tends to reinforce the model's internal biases, DPC 's cross-paradigm verification breaks the cycle of self-confirmation. By grounding the decision in explicit execution evidence from an adversarial environment and an independent reasoning path, DPC avoids falling into consensus traps on erroneous logic.

In this case, DPC correctly selects the simpler and accurate query that joins only event and budget tables, avoiding the unnecessary and erroneous inclusion of the expense table and the extraneous status filter.

C.3.2 Schema Linking Error

(QID: 1134 from BIRD Mini-Dev Split)

SC SQL (Incorrect):

```
SELECT (SELECT
      jumping
FROM
      Player_Attributes

WHERE player_api_id =
      6) -
(SELECT jumping
FROM
      Player_Attributes

WHERE player_api_id =
      23) AS
      difference;
```

→ *INCORRECT column mapping: uses* `player_api_id` *instead of* `id`.

Challenger SQL (Correct):

```
SELECT (SELECT
      jumping
FROM
      Player_Attributes

WHERE id = 6) -
(SELECT jumping
FROM
      Player_Attributes

WHERE id = 23) AS
      difference;
```

→ *CORRECT column mapping: uses* `id` *as player identifier.*

Constructing Minimum Differentiating Database (MDD): The SLICER and TESTER of DPC collaborate to synthesize a minimal execution environment in which the two candidate column mappings (`player_api_id` vs. `id`) produce different results, thereby exposing the underlying semantic discrepancy.

Dual Paradigm Verification: DPC reimplements the SQL logic using Python scripts on the MDD, producing a high-confidence reference output that serves as an execution-level oracle.

Consistency Alignment: By comparing SQL execution results against the Python reference using the **BS-F1** metric, DPC selects the semantically consistent and correct SQL formulation (i.e., using `id` instead of `player_api_id`).

Therefore, DPC successfully overcomes the symbolic blind spot and system bias inherent in SC, achieving precise alignment with user intent.

C.4 Discussion: How DPC Addresses These Errors

DPC 's dual-paradigm design systematically targets two fundamental error classes prevalent in SC and other training-free methods:

1. **Schema Mapping Errors (Category B3)** LLMs suffer from *partial observability*—they

only see a schema snippet during inference. **DPC** 's SLICER agent extracts a relevant schema subgraph and validates it via *dry-run execution* on an empty database. If joins or column references fail, the error feedback iteratively corrects the schema linking before any data synthesis. This acts as *schema-level unit testing*, eliminating B3 errors before execution.

2. **Result Representation Errors (Category A5)** LLMs struggle to infer exact output format (columns, types, rounding). **DPC** 's SOLVER generates a parallel Python/Pandas script on the same micro-database, providing a high-confidence reference output E_{py}. The **BS-F1** metric then compares SQL results against E_{py}, penalizing formatting mismatches, extra/missing columns, or incorrect data types. This cross-paradigm consistency check enforces output alignment.

SC's failures often stem from *systematic bias* and lack of execution grounding. **DPC** mitigates these by enforcing schema validation and cross-paradigm consistency, exposing errors through adversarial database construction and deterministic matching.