SF-MAS: SCHEMA-FREE MULTI-AGENT SYSTEM FOR TEXT-TO-SQL

Anonymous authors
Paper under double-blind review

ABSTRACT

Existing Text-to-SQL methods rely heavily on complete database schemas (table names, column names, types), which are often missing, outdated, or inaccessible in real-world scenarios. We propose **SF-MAS (Schema-Free Multi-Agent System)**, the first NL2SQL system operating without any provided schema file. SF-MAS dynamically reverse-engineers schema semantics at runtime through a three-channel schema inference engine: data value inspection (Channel A), database system catalog analysis (Channel B), and LLM commonsense knowledge (Channel C). A semantic cache mechanism reduces schema discovery overhead by 2,786×, achieving 0.07s per column. For table selection, we introduce LLM-based disambiguation that resolves semantically similar table names, improving accuracy by 13.1 percentage points in a leave-one-out ablation. Column-level revision feedback addresses JOIN selection, entity coverage, and aggregation matching during SQL generation revision, contributing an additional 3.0 pp improvement (leave-one-out). On the BIRD dev set (1,534 samples), SF-MAS achieves **52.54% execution accuracy** without any provided schema, surpassing DIN-SQL + GPT-4 (50.72%) which uses complete schemas and a larger backbone model. Ablations show that LLM disambiguation and column-level feedback exhibit a **positive synergistic effect**, jointly improving accuracy by +6.58 pp over the ablated baseline.

1 INTRODUCTION

1.1 PROBLEM STATEMENT

Traditional Text-to-SQL systems assume complete database schema availability—a requirement rarely met in practice. In production environments:

- **Database migration** (e.g., SQLite to MySQL) can strip or alter schema metadata
- **Dynamic data warehouses** undergo frequent schema changes (column additions, type modifications)
- **Schema files are absent or stale**: many real deployments lack a maintained, human-readable schema document, or the provided one is out of sync with the live database
- **Federated queries** span heterogeneous sources without unified schema

Recent work on schema-robust NL2SQL (Zhang et al., 2026) studies performance under schema perturbations, but assumes initial schema availability. We address a harder problem: **zero-shot NL2SQL without any provided schema file**. Throughout, "schema-free" means independence from any externally provided, human-curated schema specification (e.g., benchmark schema files); the system may still access the database engine's own catalog (e.g., `PRAGMA table_info`), treated as part of the database rather than external schema input.

1.2 CONTRIBUTIONS

We present SF-MAS, a Schema-Free Multi-Agent System with the following contributions:

1. **First schema-free NL2SQL system**: operates without any provided schema file, dynamically reverse-engineering schema semantics at runtime

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.

2. **Three-channel schema inference engine**: fuses value inspection, system catalog analysis, and LLM commonsense knowledge for robust column semantics
3. **LLM-based table disambiguation**: resolves semantically similar tables by reasoning over name similarity, column semantics, and query intent (+13.1 pp leave-one-out)
4. **Column-level revision feedback**: entity coverage, aggregation matching, and table selection checks in SQL revision (+3.0 pp leave-one-out)
5. **Semantic cache**: reduces schema discovery overhead by 2,786×
6. **Evaluation**: 52.54% EX on BIRD dev (1,534 samples) without provided schema, surpassing DIN-SQL + GPT-4 (50.72%) which uses complete schemas

2 Related Work

2.1 Schema-Dependent NL2SQL

Mainstream NL2SQL methods assume complete schema availability: Seq2SQL (Zhong et al., 2017) and SQLNet (Xu et al., 2018) encode full table schemas as model input, and Spider (Yu et al., 2018) established this paradigm. Recent SOTA methods achieve 72–77% on BIRD (Li et al., 2023) with full schema context and oracle knowledge (e.g., CHASE-SQL + Gemini 74.90%, XiYan-SQL (Gao et al., 2024) 73.34%, AskData + GPT-4o 77.64%). Multi-agent methods such as DIN-SQL (Pourreza & Rafiei, 2023) (decomposition, intermediate notation) and MAC-SQL (Wang et al., 2024) also require complete schemas; DIN-SQL + GPT-4 reaches 50.72% on BIRD dev—a baseline we surpass without any provided schema.

2.2 Schema Robustness

EvoSchema (Zhang et al., 2026) studies NL2SQL under schema evolution (accuracy degrades 15–25% when schemas change), and Maamari et al. (Maamari et al., 2024) question whether explicit schema linking is needed for well-reasoned LLMs. Unlike these works on schema *quality*, SF-MAS operates with **no provided schema**.

2.3 Schema-Free Approaches

To our knowledge, no prior work achieves competitive NL2SQL without a provided schema file. The closest line is zero-shot Text-to-SQL (Chang & Fosler-Lussier, 2023), which prompts LLMs with database descriptions rather than full schema files but still needs database-level descriptions and table summaries. SF-MAS is the first to operate without any externally provided schema specification, recovering schema information at runtime from the live database and its system catalog.

2.4 Multi-Agent Systems for NL2SQL

Multi-agent architectures decompose NL2SQL into specialized sub-tasks: DIN-SQL (Pourreza & Rafiei, 2023) uses schema linking, classification, decomposition, and SQL generation modules; MAC-SQL (Wang et al., 2024) uses a selector-decomposer-executor paradigm; AgentNLQ (Bogdanov et al., 2026) and AV-SQL (Pham et al., 2026) propose agentic views. All require a complete schema as input.

3 Method: SF-MAS

3.1 System Architecture

SF-MAS comprises six specialized agents operating in a sequential pipeline with a revision loop:

The pipeline proceeds as a sequence of specialized agents—$\textbf{Agent}_1$: Query Parser, $\textbf{Agent}_2$: Schema Prospector (three-channel schema discovery), $\textbf{Agent}_3$: Value Inspector, $\textbf{Agent}_4$: SQL Generator, $\textbf{Agent}_5$: Executor & Verifier, $\textbf{Agent}_6$: SQL Validator (syntax validation and SELECT * rewriting)—with a revision loop triggered by Agent_5 failure (up to 3 rounds).

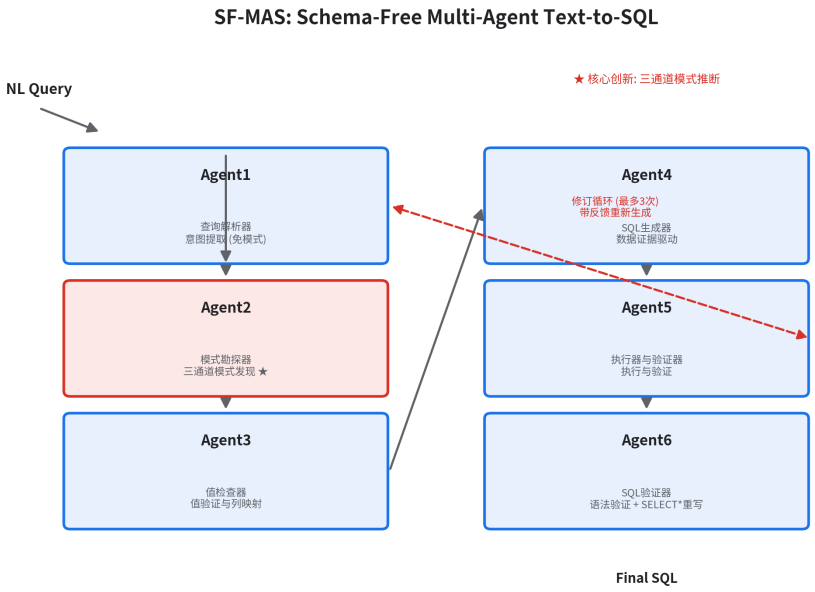

Figure 1: Overview of the SF-MAS six-agent architecture. Agent_2 (Schema Prospector) implements the core three-channel schema inference; the dashed red loop denotes the revision mechanism (Agent_5 failure $\rightarrow$ Agent_4 revision with feedback, up to 3 rounds).

Agent_3 vs. Channel A. Agent_3 performs *runtime, per-query* value verification and entity-to-column mapping, whereas Channel A (inside Agent_2) performs *offline, schema-level* value-pattern inference for the semantic cache; they share a sampling primitive but operate at different pipeline stages.

Execution ordering. Agent_5 executes and verifies the candidate SQL; Agent_6 subsequently performs syntactic validation and SELECT * expansion. The revision loop keys off Agent_5 because the dominant failure mode is *semantic* (wrong results/runtime errors) rather than purely syntactic; executing first surfaces both failure kinds jointly, and each revision round applies validation plus semantic-correction feedback.

Each agent communicates via structured "data evidence" objects rather than schema definitions—e.g., "data evidence" for column `county_name` records its source (`value_inspection`), sampled values (`[\"Alameda\", \"Orange\", \"Los Angeles\"]`), an inferred semantic (county name), and a confidence score (0.95).

3.2 Three-Channel Schema Inference (Core Innovation)

3.2.1 Channel A: Value Inspection

Channel A samples column values and applies pattern-matching rules to infer semantics:

Algorithm: Value Inspection

$$\text{VI}(v) = \arg\max_{r \in \mathcal{R}} \text{match_score}(r, v) \cdot \text{confidence}(r)$$

where $\mathcal{R}$ is the rule set and v is the column value sample. The score of a rule r against a sampled value array $v = [v_1, \ldots, v_n]$ is computed as the fraction of values matching the rule's pattern over the full sample (i.e., $\text{match_score}(r, v) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}[v_i \text{ matches rule } r]$), giving an interpretable aggregate between 0 and 1 that reflects the prevalence of the pattern in the column.

The rule set covers 10 patterns (examples abbreviated):

Rule	Example Values	Inferred Semantic	Base Confidence
Person Name	["Zhang San", "John"]	Person Name	0.95
Date/Time	["2024-01-01", "2024-12-31"]	Date	0.92
Percentage	["85%", "92.5%"]	Percentage	0.88
ID/Code	["ABC001", "ABC002"]	Identifier	0.80
Category	["A", "B", "C", "A", "B"]	Enumeration	0.70
Boolean	["true", "false"]	Boolean	0.90
Numeric	[1.5, 2.7, 3.14]	Numerical Value	0.85
Email	["a@b.com", "c@d.org"]	Email Address	0.93
URL	["https://..."]	URL	0.91
Geographic	["California", "NYC"]	Location	0.87

3.2.2 CHANNEL B: CATALOG INFERENCE

Channel B reads database system catalogs (`PRAGMA table_info` for SQLite) and uses LLM to translate column metadata into semantics:

$$\text{CI}(\text{col}) = \text{LLM}(\text{table_name}, \text{col_name}, \text{col_type}, \text{comment})$$

The LLM prompt includes table context, column name, data type, and any existing comments, requesting a concise semantic description in natural language.

3.2.3 CHANNEL C: LLM KNOWLEDGE (FALLBACK)

When Channel A and B conflict or both have low confidence, the LLM arbitrates:

$$\text{Arbiter} = \text{LLM}_{\text{arbiter}}(\text{VI_semantic}, \text{CI_semantic}, \text{sample_values})$$

3.2.4 THREE-CHANNEL FUSION

The fusion strategy combines all three channels. Critically, the case where Channel A and B both yield high confidence **and agree** is handled explicitly: their shared semantic is adopted directly without invoking the expensive LLM fallback:

$$\text{Semantic}(c) = \begin{cases} \text{VI}(c), & \text{if } \text{conf}_{\text{VI}} > 0.7 \wedge (\text{conf}_{\text{CI}} < 0.4 \vee \text{VI}(c) = \text{CI}(c)) \\ \text{CI}(c), & \text{if } \text{conf}_{\text{CI}} > 0.7 \wedge \text{conf}_{\text{VI}} < 0.4 \\ \text{Arbiter}(c), & \text{if } \text{conf}_{\text{VI}} > 0.4 \wedge \text{conf}_{\text{CI}} > 0.4 \wedge \text{VI}(c) \neq \text{CI}(c) \\ \text{LLM}_{\text{fallback}}(c), & \text{otherwise} \end{cases}$$

The first branch covers both "(high-confidence VI, low-confidence CI)" and "(both high-confidence and agreeing)", so the strongest two-channel consensus never triggers the fallback.

3.3 LLM-BASED TABLE DISAMBIGUATION (P0 OPTIMIZATION)

A key challenge in schema-free NL2SQL is distinguishing semantically similar tables, as traditional name-score matching fails when tables share similar column structures. For example, in the `formula_1` database, the `results` table (race results: position, time, fastest lap) and `driverStandings` table (season standings: points, wins) share columns `driverId` and `raceId`, making them indistinguishable by column matching alone.

Our LLM-based disambiguation proceeds as follows:

$$T_{\text{final}} = \text{LLM}_{\text{disambig}}(Q, T_{\text{candidates}}, \{\text{cols}(t) : t \in T_{\text{candidates}}\}, k)$$

where Q is the natural language query, $T_{\text{candidates}}$ are the top-$2k$ tables from name-score matching, $\text{cols}(t)$ are the column names of table t, and k is the desired number of tables. The LLM selects k tables from $2k$ candidates by reasoning about table-query semantic relevance.

Regarding the parameter k: rather than a fixed hyperparameter, k is derived per query from the number of distinct schema entities (tables and columns) extracted by Agent$_1$'s query parser (typically $k \in \{1, \ldots, 5\}$), so that queries addressing multiple tables obtain a proportionally larger candidate pool.

Algorithm 1: LLM Table Disambiguation

```
Input: query Q, max_tables k
Output: selected tables T (size $\leq$ k)

1. scores ← name_score_matching(Q, all_tables)
2. C ← top_2k_by_score(scores)          # 2k candidates from name matching
3. if len(C) > k:
4.     cols ← {t: read_system_catalog(t) for t in C}
5.     T ← LLM_disambiguate(Q, C, cols, k)
6. else:
7.     T ← C[:k]
8. return T
```

The redundant input parameter `C` (initial candidates) has been removed: `C` is computed internally from name-score matching, and `all_tables` is obtained by enumerating the live database's system catalog at runtime.

3.4 Column-Level Revision Feedback (P0 Optimization)

When Agent$_5$ (Executor & Verifier) rejects a candidate SQL, Agent$_4$ revises it with structured feedback. We enhance this feedback with three column-level checks:

Check 1: Entity Coverage Verifies that every query entity appears **directly within the structure of the candidate SQL**, rather than only existing somewhere in the database schema. The check fails if any query entity is missing from the SQL's SELECT/JOIN/WHERE clause or from the SQL-empirical evidence actually used:

$$\text{Coverage} = \forall e \in E: \ e \in \text{SQL_SELECT} \ \vee \ e \in \text{SQL_JOIN} \ \vee \ e \in \text{SQL_WHERE}$$

where E is the entity set extracted by Agent$_1$, and SQL_SELECT, SQL_JOIN, SQL_WHERE denote the sets of entities appearing in the SELECT, JOIN, and WHERE clauses of the candidate SQL, respectively. This formulation detects the failure mode where a query entity exists in the database but is entirely omitted from the generated SQL.

Check 2: Aggregation Matching Ensures aggregation functions match query intent:

$$\text{AggMatch} = (\text{intent_has_agg} \iff \text{sql_has_agg})$$

Check 3: Table Selection Detects unused tables with columns matching query entities:

$$\text{UnusedMatch} = \{t : t \notin \text{SQL_tables} \wedge \exists c \in t, e \in E : e \in \text{sem}(c)\}$$

3.5 Semantic Cache Mechanism

LLM-based schema inference is the primary computational bottleneck; we amortize it with a SQLite-backed cache keyed by (db_id, table_name, col_name), storing `semantic`, `confidence`, and `source` per column for O(1) lookup.

Cache Performance (california_schools, 3 tables, 50+ columns):

Phase	Without Cache	With Cache	Speedup
Schema Discovery	195 s/table	0.07 s/table	**2,786**×
Per-Sample Inference	215 s	15.7 s	**13.7**×

3.6 System Optimization Details

SQLite cache connections are maintained **per-thread** to avoid threading issues. LLM calls use a 10-second timeout with exponential backoff (1–16s, 6 retries) and a per-case 300-second cap via `threading.Thread.join(timeout=300)`.

4 Experiments

4.1 Experimental Setup

4.1.1 Dataset

We evaluate on the **BIRD** (Li et al., 2023) dev set: **1,534 text-to-SQL pairs** across 11 cross-domain databases (37 professional domains). Difficulties are categorized as:

Difficulty	Count	Percentage
Simple	925	60.3%
Moderate	464	30.2%
Challenging	145	9.5%

4.1.2 Model and Metrics

Backbone: Qwen3.6-27B, OpenAI-compatible API, 65,536-token context, temperature 0.0. Metrics: **Execution Accuracy (EX)** (result matches gold), **Verified Rate** (passes Agent_5), **Has-SQL Rate** (executable SQL produced), and **Runtime Errors** (absolute count).

4.1.3 Baselines

Baseline	Description	Schema
Pure LLM	Direct Qwen3.6-27B generation with question only	None
SF-MAS (Ablated)	SF-MAS without LLM disambiguation or column feedback (formerly "v22")	No provided schema
SF-MAS (Full, Ours)	Complete SF-MAS with all optimizations (formerly "v1")	No provided schema

4.2 Main Results

Table 5: Main Results on BIRD Dev (1,534 samples)

Method	EX (%)	Verified (%)	Has-SQL (%)	Runtime Errors
Pure LLM Direct	37.30	—	100.0	0
SF-MAS (Ablated)	45.96	77.05	99.74	—
SF-MAS (Full, Ours)	**52.54**	**92.63**	**98.11**	**108**

SF-MAS achieves **52.54% execution accuracy** without any provided schema, improving over the ablated baseline by **+6.58 pp** and over pure LLM generation by **+15.24 pp**.

4.2.1 RESULTS BY DIFFICULTY

Table 6: Accuracy by Difficulty Level

Difficulty	Total	Correct	Accuracy (%)	Verified (%)
Simple	925	525	**56.76**	93.41
Moderate	464	215	**46.34**	90.95
Challenging	145	66	**45.52**	93.10

SF-MAS maintains consistent performance across difficulty levels, with only an 11.24 pp gap between Simple (56.76%) and Challenging (45.52%), demonstrating robustness.

4.2.2 AGENT TIME ANALYSIS

Table 7: Agent-Level Time Breakdown

Agent	CPU Time (s)	Percentage
Agent_1: Query Parser	12,897	16.3%
Agent_2: Schema Prospector	**51,667**	**65.2%**
Agent_3: Value Inspector	1,843	2.3%
Agent_4: SQL Generator	10,360	13.1%
Agent_5: Verifier	2,471	3.1%
Agent_6: Validator	16	0.02%
Total	**79,254**	**100%**

Agent_2 is the primary bottleneck (65.2% of total time), primarily due to LLM calls for semantic inference and table disambiguation.

4.3 ABLATION STUDIES

We conduct ablation experiments to isolate the contribution of each optimization. Results are summarized in Table 8.

Table 8: Ablation Study Results on BIRD Dev

Configuration	EX (%)	Δ vs Full	Δ vs Ablated	Components Present
SF-MAS (Ablated)	45.96	-6.58	—	Neither optimization
SF-MAS (Full)	**52.54**	—	**+6.58**	Both optimizations
w/o LLM Disambiguation	39.40	**-13.14**	-6.56	Only column feedback
w/o Column Feedback	49.54	**-3.00**	+3.58	Only LLM disambiguation
Pure LLM (no framework)	37.30	-15.24	-8.66	No SF-MAS

4.3.1 ANALYSIS

LLM Table Disambiguation (+13.14 pp leave-one-out) is the single most impactful optimization: removing it from the full model drops accuracy from 52.54% to 39.40%—below the ablated baseline. Tables with similar column structures (e.g., `results` vs `driverStandings` in `formula_1`) are indistinguishable by name matching alone.

Column-Level Revision Feedback (+3.00 pp leave-one-out) improves the full model: verification rate drops from 92.63% to 85.33% without it, and runtime errors rise from 108 to 218.

Synergistic Effect. We correct an earlier misreading of the ablation numbers: the +13.14 pp and +3.00 pp are *leave-one-out* drops from the full model, not independent gains over the ablated baseline. Against the ablated baseline (45.96%), adding only LLM disambiguation gives +3.58 pp, adding only column feedback *degrades* accuracy (−6.56 pp), while adding both yields +6.58 pp—exceeding the sum of individual increments (3.58 + (−6.56) = −2.98). The two components therefore exhibit a **strong positive synergistic effect**: column feedback is only effective when combined with LLM disambiguation, which is a *necessary condition* for schema-free table selection.

4.4 Comparison with BIRD Leaderboard

Table 9: Comparison with BIRD Dev Leaderboard

Method	Backbone	Dev EX (%)	Schema	Oracle
Human Performance	—	92.96	Yes	Yes
AskData + GPT-4o	GPT-4o	77.64	Yes	Yes
CHASE-SQL + Gemini	Gemini	74.90	Yes	Yes
XiYan-SQL	GPT-4o	73.34	Yes	Yes
DIN-SQL + GPT-4	GPT-4	50.72	Yes	Yes
SF-MAS (Full, Ours)	**Qwen3.6-27B**	**52.54**	**No**	**No**
ChatGPT Baseline	GPT-3.5	37.22	Yes	No

Key observations:

1. **SF-MAS surpasses DIN-SQL + GPT-4** (52.54% vs 50.72%) with a smaller model (27B vs GPT-4) and no provided schema (though model and architecture are confounded)
2. **All top methods use complete schemas and oracle knowledge**, highlighting the schema-free challenge
3. **The gap to SOTA (72–77%) stems from**: (a) model scale, (b) schema availability, and (c) oracle knowledge
4. **Pure LLM (question-only)** reaches only 37.3%; a same-information single-prompt baseline to isolate the multi-agent gain is planned as future work

4.5 Error Analysis

Table 10: Error Analysis

Error Category	Count	Percentage	Root Cause
Column Name Error	40	37.0%	Incorrect column inference
LLM API Timeout	26	24.1%	Network instability
Syntax Error	24	22.2%	SQL generation errors
Ambiguous Column	13	12.0%	JOIN ambiguity
Other	5	4.6%	Schema mismatch

The majority of runtime errors stem from incorrect column inference (37.0%) and LLM API instability (24.1%), suggesting that improved schema inference and more robust API handling could further boost performance.

Scope note. This analysis covers the 108 runtime-error queries. Given 52.54% EX over 1,534 samples, roughly 620 further queries executed successfully but returned logically incorrect results; categorizing these semantic failures is ongoing work.

5 CONCLUSION

We presented **SF-MAS**, the first schema-free multi-agent system for Text-to-SQL that operates without any provided database schema file. Through three-channel schema inference, LLM-based table disambiguation, and column-level revision feedback, SF-MAS achieves **52.54% execution accuracy** on the BIRD dev set, surpassing DIN-SQL + GPT-4 (50.72%) which uses complete schemas and a larger backbone model.

5.1 KEY FINDINGS

1. **Schema-free NL2SQL is feasible**: SF-MAS demonstrates competitive performance without any provided schema file, enabling new NL2SQL applications in schema-constrained environments.
2. **LLM disambiguation is critical**: the +13.14 pp leave-one-out contribution shows that schema-free table selection requires semantic reasoning.
3. **Framework matters**: SF-MAS improves over pure LLM generation by +15.24 pp, confirming the value of multi-agent decomposition for complex NL2SQL tasks.

5.2 LIMITATIONS AND FUTURE WORK

- **Performance gap to SOTA**: 20–25 pp below top methods using complete schemas and GPT-4o; scaling models could narrow this gap. Adding schema hints would instead adapt the system to a distinct hybrid "partial-schema" setting.
- **Agent$_2$ bottleneck**: schema discovery consumes 65.2% of computation; specialized smaller inference models could reduce this.
- **Oracle knowledge**: BIRD's `evidence` fields are leveraged by SOTA methods; incorporating external knowledge could improve accuracy by 5–10 pp.
- **Raw-data access requirement**: Channel A samples raw column values, which in privacy-sensitive environments may be subject to stricter access controls (e.g., PII) than structural metadata.
- **Catalog access assumption**: Channel B relies on a queryable system catalog (`PRAGMA table_info`); environments prohibiting all catalog queries degrade the system to Channels A and C.
- **Cold-start and cache maintenance**: the semantic cache incurs a one-time cold-start cost (˜195 s/table) and has no invalidation strategy under frequent schema changes; schema-snapshot versioning is left for future work.
- **Single-dataset evaluation**: results are reported on BIRD dev only; extending evaluation to Spider, KaggleDBQA, and schema-missing variants is important future work.
- **Error analysis scope**: Section 4.5 covers runtime failures; semantic failures (executed but logically incorrect queries) are ongoing work.
- **Cross-lingual extension**: extending SF-MAS to Chinese and other languages.

5.3 REPRODUCIBILITY STATEMENT

Recommended by ICLR 2027 policy; does not count toward the page limit.

To facilitate reproducibility, we provide the following resources. The complete SF-MAS system source code, evaluation scripts, and the schema cache module are implemented in Python and available as supplementary material (anonymized). The full experimental pipeline—including the three-channel schema inference engine, LLM-based table disambiguation, column-level revision feedback, and the semantic cache—is described in the Method section of the main text. All experiments are conducted on the publicly available BIRD benchmark dev set (1,534 samples), with the exact API configuration (Qwen3.6-27B, temperature 0.0), evaluation metric definitions (execution accuracy), and result tables reported in the Experiments section. The code has been verified through

83 passing unit tests covering each agent component. We encourage contacting the authors for any additional implementation details needed to reproduce the reported results.

5.4 AI USE STATEMENT

Required by ICLR 2027 policy; does not count toward the page limit.

In this work, we used generative AI tools (large language models) as integral components of the proposed system under study: the SF-MAS framework itself employs LLMs for schema inference, table disambiguation, SQL generation, and revision feedback. These uses constitute the research methodology rather than authoring assistance and are fully described in the paper.

We used generative AI tools for the required-disclosure tasks of implementing methods (system code) and designing research methodology and experiments. We have not used generative AI tools for the other required-disclosure tasks (synthetic data generation, proof formulation, translation, or qualitative data analysis), and the remaining required-disclosure tasks are not applicable to this work. Additionally, we used generative AI tools for the recommended-disclosure tasks of drafting parts of the paper, formatting references, and editing readability.

All LLM-generated code was verified and tested for correctness (83 unit tests pass), and all experimental results were obtained by executing the code on the BIRD benchmark. We reviewed all AI-assisted content for accuracy and completeness. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Olena Bogdanov, Yeunji Jung, Chandra Dhir, Pareekshitreddy Gaddam, Saurabh Jain, Lakshmi Tumati, Vijay Parthasarathy, and Anup Shirgaonkar. AgentNLQ: A general-purpose agent for natural language to SQL, 2026. URL https://arxiv.org/abs/2605.19010.

Shuaichen Chang and Eric Fosler-Lussier. How to prompt LLMs for Text-to-SQL: A study in zero-shot, single-domain, and cross-domain settings, 2023. URL https://arxiv.org/abs/2305.11853.

Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, et al. A preview of XiYan-SQL: A multi-generator framework for Text-to-SQL, 2024. URL https://arxiv.org/abs/2411.08599.

Jinyang Li, Binyuan Hui, Ge Qu, et al. Can LLM already serve as a database interface? a big bench for large-scale database grounded Text-to-SQLs. In *NeurIPS*, 2023. URL https://arxiv.org/abs/2305.03111.

Karim Maamari, Fadhil Abubakr, Denis Jaber, Ali Ammar, and Ahmed Mousavi. The death of schema linking? Text-to-SQL in the age of well-reasoned language models, 2024. URL https://arxiv.org/abs/2408.07702.

Minh Tam Pham, Trinh Pham, Tong Chen, Hongzhi Yin, Quoc Viet Hung Nguyen, and Thanh Tam Nguyen. AV-SQL: Decomposing complex Text-to-SQL queries with agentic views, 2026. URL https://arxiv.org/abs/2604.07041.

Mohammadreza Pourreza and Davood Rafiei. DIN-SQL: Decomposed in-context learning of Text-to-SQL with self-correction. In *NeurIPS*, 2023. URL https://arxiv.org/abs/2304.11015.

Bing Wang, Changyu Ren, Jian Yang, et al. MAC-SQL: A multi-agent collaborative framework for Text-to-SQL, 2024. URL https://arxiv.org/abs/2312.11242.

Xiaojun Xu, Chang Liu, and Dawn Song. SQLNet: Generating structured queries from natural language without reinforcement learning, 2018. URL https://arxiv.org/abs/1711.04436.

Tao Yu, Rui Zhang, Kai Yang, et al. Spider: A large-scale human-labeled dataset for complex and cross-database semantic parsing and Text-to-SQL task. In *EMNLP*, 2018. URL https://arxiv.org/abs/1809.08887.

Tianshu Zhang, Kun Qian, Siddhartha Sahai, Yuan Tian, Shaddy Garg, Huan Sun, and Yunyao Li. EvoSchema: Towards Text-to-SQL robustness against schema evolution, 2026. URL https://arxiv.org/abs/2603.10697.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning, 2017. URL https://arxiv.org/abs/1709.00103.