

FlexSQL: Flexible Exploration and Execution Make Better Text-to-SQL Agents

Quang Hieu Pham^A Yang He^F Ping Nie^W Canwen Xu^S
 Davood Rafiei^A Yuepeng Wang^F Xi Ye^{*AMP} Jocelyn Qiaochu Chen^{*AMN}

^AUniversity of Alberta ^FSimon Fraser University ^WUniversity of Waterloo

^SSnowflake ^PPrinceton University ^NNew York University

^MAlberta Machine Intelligence Institute (Amii)

{quanghieupham, xi.ye, jocelyn.chen}@ualberta.ca

Abstract

Text-to-SQL over large analytical databases requires navigating complex schemas, resolving ambiguous queries, and grounding decisions in actual data. Most current systems follow a fixed pipeline where schema elements are retrieved once upfront and the database is only revisited for post-hoc repair, limiting recovery from early mistakes. We present FlexSQL, a text-to-SQL agent whose core design principle is flexible database interaction: the agent can explore schema structure, inspect data values, and run verification queries at any point during reasoning. FlexSQL generates diverse execution plans to cover multiple query interpretations, implements each plan in either SQL or Python depending on the task, and uses a two-tiered repair mechanism that can backtrack from code-level errors to plan-level revisions. On Spider2-Snow, using gpt-oss-120b, FlexSQL achieves a 65.4% score, outperforming strong open-source baselines that use stronger, larger models such as gpt-o3 and DeepSeek-R1. When integrated into a general-purpose coding agent (as skills in Claude Code), our approach yields over 10% relative improvement on Spider2-Snow. Further analysis shows that flexible exploration and flexible execution jointly contribute to the effectiveness of our approach, highlighting flexibility as a key design principle. Our code is available at: <https://github.com/StringNLAB/FlexSQL>.

1 Introduction

As analytical databases grow in scale and complexity, text-to-SQL systems face an increasingly difficult reasoning problem (Liu et al., 2025; Hong et al., 2025). The model must navigate large schemas, resolve ambiguities in the question, and ground its decisions in actual data—all before writing a single query. Real-world data warehouses on platforms like Snowflake and BigQuery can contain hundreds of tables organized into multiple schemas (Li et al., 2023; Lei et al., 2025; Huo et al., 2026), making it difficult to even identify which parts of the warehouse are relevant, let alone query them correctly. This challenge is reflected in Spider2.0 (Lei et al., 2025), where nearly 10% of the 152 Snowflake databases exceed 100 tables, and the largest databases contain 60,000 to 72,000 columns.

Most current systems follow a fixed pipeline: retrieve schema elements, generate SQL, then repair errors (Wang et al., 2020; Li et al., 2024; Talaei et al., 2024; Pourreza et al., 2025; Hao et al., 2025; Deng et al., 2025; Li et al., 2025a;b; Lee et al., 2025). While some recent methods incorporate database interaction (e.g., executing candidate queries to check for errors, or retrieving schema elements before generation), these interactions still occur at fixed, predetermined stages. The model explores the schema once upfront, commits to a complete query based on that static snapshot, and only revisits the database to validate or repair the output. This rigidity means downstream repair can catch syntax errors, but

*Equal advising.

it cannot recover when the model chose the wrong tables or misunderstood a column’s meaning in the first place.

We present FlexSQL, a text-to-SQL agent whose core design principle is *flexible database interaction*: the agent incrementally discovers schema structure, grounds its decisions in actual data values, and can revise its approach based on what it finds—at any point during reasoning. For instance, if the agent encounters an error during code generation that stems from a wrong table choice, it can backtrack all the way to schema exploration rather than being trapped in a repair loop.

To make this flexible interaction concrete, FlexSQL equips the agent with a suite of tools designed specifically for incremental database exploration: the agent can navigate schema hierarchies top-down, inspect actual column values, and run exploratory queries against the database. These tools are available throughout the entire reasoning process, so the agent can ground its plans in real data and verify intermediate results at any point.

While these tools address the grounding problem, a second challenge remains: natural language queries are often inherently ambiguous, admitting multiple valid interpretations (Li & Jagadish, 2014; Pourreza & Rafiei, 2023a; Klopfenstein et al., 2026). Rather than committing to one, FlexSQL uses diversity-enforced sampling (Zhang et al., 2025) to produce K plans covering different table choices, join paths, and predicate readings. By exploring a broader hypothesis space, FlexSQL naturally improves its robustness against query ambiguity, and effectively leverages inference-time compute as pass@ K improves with more candidates (Wang et al., 2023; Liu et al., 2026).

The diverse plans also call for flexibility in implementation: some strategies involve multi-step transformations that are cumbersome in pure SQL, so FlexSQL allows each plan to be implemented in either SQL or Python. Python’s more flexible control structures are especially useful for problems that require iterative refinement and branching conditionals (Bhatia et al., 2024), and Python implementations are subsequently translated into SQL. Together, flexible interaction, diverse planning, and bilingual generation allow FlexSQL to cover a broad space of candidate solutions while grounding each one in the actual database.

We evaluate FlexSQL using the open-source gpt-oss models (Agarwal et al., 2025) on two settings from the Spider2.0 benchmark (Lei et al., 2025), covering both cloud data warehouses (Spider2-Snow) and local databases (Spider2-SQLite). With gpt-oss-120b, FlexSQL reaches 65.44% Majority@8 on Spider-Snow, outperforming DSR-SQL with DeepSeek-R1 and ReFoRCE with gpt-o3 despite using a significantly smaller backbone. Under our framework, gpt-oss-20b also either surpasses or remains comparable to baselines using gpt-oss-120b. We further integrate FlexSQL as skills in a general-purpose coding agent (e.g., Claude Code (Anthropic, 2026)), yielding over 10% relative improvement on Spider2-Snow.

Our contributions are:

- A text-to-SQL agent built around flexible database interaction, equipped with a suite of tools for incremental schema navigation, data inspection, and code execution that the agent can use throughout the entire reasoning process.
- A two-stage plan-to-program architecture with a flexible backtracking mechanism that allows the agent to revise not only its code but also its plan and schema assumptions when errors are detected.
- Diversity-enforced plan sampling to handle query ambiguity by covering multiple valid interpretations, combined with bilingual program generation (SQL or Python) to flexibly implement the varied strategies that arise.
- Strong empirical results on Spider2.0, where FlexSQL consistently outperforms baselines across models and exhibits effective test-time scaling with increased sampling budgets.

2 Motivating Example

We illustrate the key challenges of text-to-SQL over complex databases and how FlexSQL addresses them through a concrete example. Consider the following question posed over an

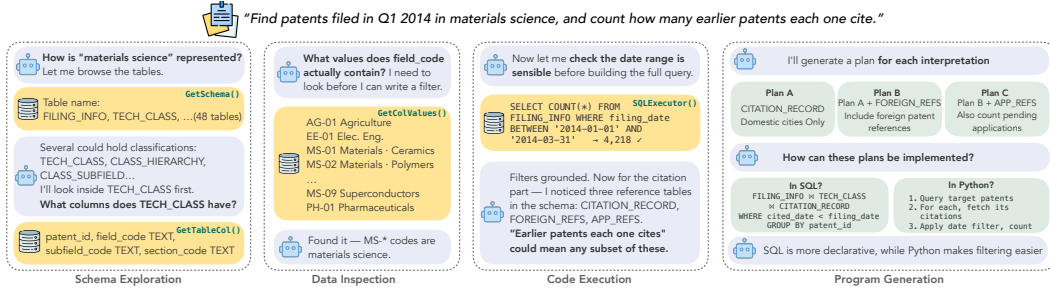


Figure 1: Condensed trace of FlexSQL on the motivating example. The agent progressively discovers the schema, grounds its filters in actual data values, verifies intermediate results, and generates diverse plans covering different interpretations of the query—all through incremental tool interaction with the database.

intellectual property database containing 48 tables spanning filings, inventors, technology classifications, citations, and legal events: *"Find patents filed in Q1 2014 in materials science, and count how many earlier patents each one cites."*

Answering this question requires solving two problems: identifying how "materials science" is represented in the database, and deciding what "earlier patents each one cites" means. Both turn out to be harder than they appear.

Challenge 1: Schema discovery and data grounding. The phrase "materials science" does not appear as a table or column name in the database. The relevant information is buried in the field_code column of a table called TECH_CLASS, one of 48 tables, several of which have plausible-sounding names like CLASS_HIERARCHY, CLASS_SUBFIELD, and FIELD_SECTION. Even after locating the right table, the correct filter (field_code IN ('MS-01', ..., 'MS-09')) can only be determined by examining actual data values, not schema metadata. A system that performs schema linking once upfront and never revisits the database may fail to discover this mapping. In practice, we observe that one baseline missed TECH_CLASS entirely, returning all patents filed in Q1 2014 instead of only those in materials science.

Challenge 2: Query ambiguity. The phrase "earlier patents each one cites" admits multiple valid readings. The database contains three citation-related tables (CITATION_RECORD, FOREIGN_REFS, APP_REFS), and it is unclear which subset the query intends: (1) using only domestic citations, (2) including foreign references, or (3) also counting pending applications, each yielding different results. A second baseline found TECH_CLASS correctly but unioned all three citation tables, overcounting by an order of magnitude. Again, all candidates shared the same structural choice, leaving no room for voting to recover.

How FlexSQL handles this query. Figure 1 shows a condensed trace of the agent's interaction with the database. Rather than committing to a fixed schema selection, the agent explores incrementally: it calls GetSchema to list all 48 tables, identifies TECH_CLASS as a candidate, inspects its columns via GetTableCol, and then calls GetColValues to discover that field_code values MS-01 through MS-09 correspond to materials science. With the technology domain filter grounded, the agent also needs to verify the date constraint, which is stored in a separate table, FILING_INFO. Before writing the full query, it runs a verification query (SELECT COUNT(*) FROM FILING_INFO with the date filter) to confirm the date range yields a reasonable number of rows.

With the schema grounded, the agent turns to the ambiguity in "earlier patents." Rather than committing to one interpretation, it generates multiple plans that differ in which citation tables to use and how to apply date filters: Plan A uses only CITATION_RECORD with a filter on cited_date; Plan B additionally includes FOREIGN_REFS; Plan C further adds APP_REFS. Each plan can also be implemented in either SQL or Python. For instance, a plan that iterates over each patent's citations and applies per-patent filtering is more naturally expressed as

a Python loop than as a multi-way self-join in SQL. Because all plans discover `TECH_CLASS` through tools, they agree on which patents to retrieve but produce different citation counts; majority voting over execution outputs selects the most supported interpretation.

The rest of this paper describes the three mechanisms at work: tool-based database interaction (§3), diversity-enforced plan generation (§4.1), and program generation (§4.2).

3 Tool Interfaces for Database Querying

We design a set of tools that allow language model agents to interact with relational databases incrementally during text-to-SQL reasoning. Agents in FlexSQL are initialized with only coarse-grained metadata (e.g., a list of schema names) and use these tools to retrieve information on demand as their understanding of the query develops. The tools are organized around three capabilities: navigating schema structure, inspecting data values, and executing code against the database.

Database hierarchy and agent inputs. Analytical databases on platforms like Snowflake and BigQuery are typically organized hierarchically: a *database* contains one or more *schemas*, each grouping a set of related *tables* by business domain. For example, a city government database might have a Transportation schema (with tables for bike-sharing and traffic) and an Environment schema (with tables for air quality and waste management). Our tools are designed around this hierarchy.¹ At initialization, the agent receives the user query, any available external knowledge documents, and a list of schema names in the database. All further information, such as table names, column definitions, and data values, is retrieved through tool calls.

Schema exploration. The first two tools let the agent navigate this hierarchy top-down:

- **GetSchema(schema_name):** Given a schema name (e.g., Transportation), returns all table names within that schema. This is useful because a single database may contain dozens of schemas, and the agent needs to narrow its focus to the relevant domain before examining individual tables.
- **GetTableCol(table_name):** Given a fully qualified table name, returns its column names, data types, and a few sample values. This provides enough information to assess whether a table is relevant and how it might join with other tables.

Data inspection. Schema metadata alone is often insufficient. As illustrated in Section 2, the agent may examine actual data values to resolve ambiguities. Two tools support this:

- **GetColValues(column_name, table_name):** Returns the distinct values stored in a specific column. This is essential when the query depends on categorical values that are not apparent from column names alone (e.g., discovering that “chemistry” corresponds to classification codes C05–C13).
- **FindRows(term, column_name, table_name, additional_columns):** Performs a keyword search within a column and returns matching rows, optionally including values from other columns in the same table. Here, *term* is the search string (e.g., a product name or abbreviation) the agent wants to locate in the data. This helps verify whether a value exists and how it relates to other columns before committing to a filter condition.

Code execution. The final two tools let the agent run code against the database, supporting iterative development and verification of partial solutions:

- **SQLExecutor(sql_query):** Executes a SQL query and returns the result set or an error message. The agent can use this to test subqueries, verify join logic, or check intermediate results before assembling a full query.

¹For databases without an explicit schema layer (e.g., standard relational databases), the schema level is simply flattened and the agent navigates directly from the database to its tables.

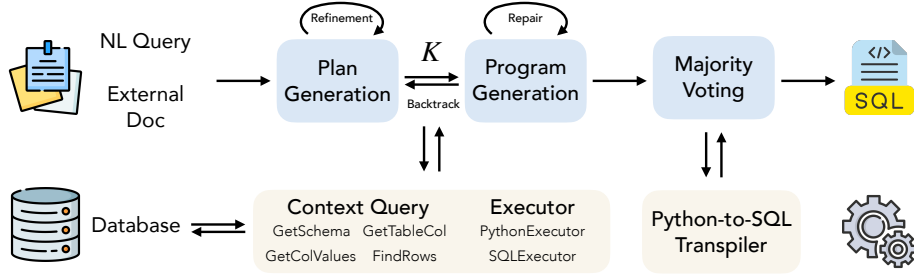


Figure 2: Overview of the FlexSQL framework. The agent takes a natural language query and optional external documents as input, and produces a SQL query through three components: **Plan Generation** explores the database via context query and executor tools to produce K diverse plans, with a refinement loop; **Program Generation** translates each plan into SQL or Python, with a repair loop for code-level errors and backtracking to Plan Generation for plan-level errors; **Majority Voting** selects the consensus answer and transpiles Python solutions to SQL when needed.

- **PythonExecutor(program)**: Executes Python code in a stateful sandbox with database access. This supports multi-step reasoning, such as fetching data with a simple query, transforming it programmatically, and verifying the output, which can be difficult to express in a single SQL statement.

4 FlexSQL Framework

Given a user query q , a database $\mathcal{D}$ with schemas $\{S_1, \dots, S_n\}$, and optional domain documents $\mathcal{K}$, FlexSQL produces a final SQL query a^* and its execution result o^* . As shown in Figure 2, the framework has three components. **Plan Generation** (§4.1) uses the context query tools (GetSchema, GetTableCol, GetColValues, FindRows) and executors (SQLExecutor, PythonExecutor) to explore the database and construct K diverse execution plans, with a refinement loop that iterates on each plan using tool feedback. **Program Generation** (§4.2) translates each plan into executable SQL or Python, with a repair loop for code-level errors and a backtracking mechanism that revises the plan itself when deeper reasoning errors are detected. Finally, **majority voting** groups all programs by execution output regardless of language, selects the consensus answer, and transpiles any Python solution to SQL.

4.1 Plan Generation

The first stage produces K execution plans that cover different interpretations of the query. Each plan is a natural language description of a query strategy, e.g., which tables to use, how to join them, what filters to apply, grounded in the actual database through tool interaction.

Context building. Large analytical databases can contain hundreds of tables, many irrelevant to any given query. Before planning, we apply lightweight preprocessing: pruning columns that are entirely null and grouping tables that share identical schemas but differ only by a time-based suffix (e.g., GA_SESSION_20240101, GA_SESSION_20240202). If external documents $\mathcal{K}$ are provided, we extract a query-relevant summary. Full preprocessing details are provided in Appendix A.

Even after preprocessing, the database may remain too large for direct context grounding. We address this with a routing step: the agent examines the list of schema names and uses GetSchema and GetTableCol to identify a relevant subset of schemas $S_{\text{rel}} \subseteq \mathcal{D}$, exploiting the database $\rightarrow$ schema $\rightarrow$ table hierarchy described in Section 3.

Diversity-enforced sampling. Given S_{rel} , the agent generates K plans in parallel batches of size M using verbalized sampling (Zhang et al., 2025): each batch is prompted to produce

plans that differ from those already generated, encouraging coverage of genuinely different query interpretations. During planning, the agent actively explores the database using the full tool suite such as inspecting schema structure, examining column values, and running test queries, to ground its plans in the actual data.

Plan refinement. Each generated plan is reviewed by the agent against the information gathered through tool interaction. If the review identifies issues (e.g., a referenced table does not exist, or a filter contradicts the actual column values), the agent refines the plan by re-exploring the database and regenerating, as shown by the refinement loop in Figure 2.

4.2 Program Generation

The second stage translates each plan into an executable program, revises it if needed, and selects the final answer.

Bilingual program synthesis. For each plan p_i , the agent generates a program a_i in either Python or SQL, choosing the language that best fits the task. SQL is well-suited for declarative relational queries, while Python is better suited for procedural workflows involving multi-step transformations, iterations, or computations that are cumbersome in a single SQL statement. During synthesis, the agent can use `SQLExecutor` and `PythonExecutor` to test partial implementations by running subqueries, checking intermediate outputs, and debugging, before committing to a final program.

Repair. After execution, a review step evaluates whether the program’s output o_i is consistent with the original query and plan. If a *code-level* error is detected, such as syntactic errors, wrong aggregation, or incorrect column references, the program is regenerated with the error message as feedback (see the repair loop in Figure 2). We allow up to R repair rounds (set to 3 in our experiments).

Plan backtracking. If the review instead identifies a *plan-level* error, such as wrong tables, missing joins, or an incorrect interpretation of the query, repair alone cannot fix the problem. In this case, the system backtracks to Plan Generation (Figure 2): the agent revises the plan itself, potentially re-exploring the database with tools, before re-attempting program synthesis. This allows the system to recover from deeper reasoning errors, not just surface-level implementation mistakes.

Majority voting and transpilation. After all K programs have been generated—some in SQL, some in Python—we group them by execution output regardless of language and select the largest equivalence class $\mathcal{A}^*$ as the consensus answer. This allows SQL and Python solutions to reinforce each other during voting, which is important because some query strategies are easier to implement correctly in Python. If the consensus answer was produced by a SQL query, we return it directly; otherwise, we transpile a Python program from $\mathcal{A}^*$ into SQL so that the final output is always a SQL query. The transpiler generates a candidate query, executes it, and verifies that its output matches the consensus before returning. We include our heuristic transpilation rules in Appendix F.

5 Experiments

Datasets. We evaluate FlexSQL on two subsets of Spider2.0 (Lei et al., 2025): Spider2-Snow (544 questions over Snowflake databases) and Spider2-SQLite (135 questions over SQLite databases). While Spider2-Snow contains enterprise-scale databases spanning hundreds of tables, Spider2-SQLite features moderately sized schemas averaging 14.4 tables (with a maximum of 38). Together, this benchmark suite tests performance across distinct SQL dialects and varying schema complexities.

Baselines. We evaluate our method against two strong open-source baselines on the Spider2.0 leaderboard: ReFoRCE (Deng et al., 2025) and DSR-SQL (Deng et al., 2025). For a

Model	Metric	Spider2-Snow			Spider2-SQLite		
		DSR-SQL	ReFoRCE	Ours	DSR-SQL	ReFoRCE	Ours
gpt-oss-120b	Pass@1	33.27	44.12	55.15	48.15	45.19	57.78
	Majority@8	50.37	48.90	59.74	51.85	54.07	64.44
	Pass@8	63.24	62.32	78.68	57.78	71.11	78.52
	Micro Acc.	28.38	28.11	44.07	25.79	32.11	49.69
gpt-oss-20b	Pass@1	32.54	36.76	43.20	34.81	42.96	50.37
	Majority@8	42.65	43.01	50.92	37.04	46.67	54.07
	Pass@8	52.39	59.01	71.14	39.26	68.15	78.52
	Micro Acc.	18.55	24.91	37.85	20.44	29.52	45.34

Table 1: Performance comparison (in %) on Spider2-Snow and Spider2-SQLite datasets.

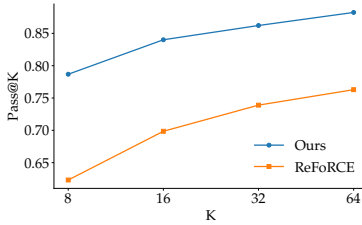


Figure 3: Pass@K test-time scaling of FlexSQL and ReFoRCE.

Method	Model	Majority@K
DSR-SQL	DeepSeek-R1	63.80
ReFoRCE	gpt-o3	62.89
Ours	gpt-oss-20b, $K = 8$	50.92
	gpt-oss-120b, $K = 8$	59.74
	gpt-oss-120b, $K = 16$	65.44

Table 2: Majority@K on Spider2-Snow compared against leading open-source systems on the Spider2.0 leaderboard.

fair comparison, all experiments use gpt-oss-120b and gpt-oss-20b (Agarwal et al., 2025) with identical sampling and repair budgets and equal access to database tools. At the time we develop FlexSQL, only the two baselines provided open-source codebases. Our inference configurations are in Appendix G. We also demonstrate the broader generalization of FlexSQL by integrating our flexible interaction paradigm to Claude Code, a widely adopted general-purpose coding agent.

Evaluation metrics. We consider three evaluation dimensions. (1) **Execution accuracy**, measured by Pass@1 against the released golden execution results. (2) **Test-time scalability**, assessed via Pass@8 and majority-vote accuracy over eight samples (**Majority@8**). (3) **Answer diversity**, measured by **micro accuracy**: since Spider2.0 includes multiple gold answers per question, micro accuracy assigns fractional credit (e.g., $1/N$ for N gold answers) based on whether the model recovers any valid solution. We also evaluate schema linking quality at the table level using precision and recall.

5.1 Main results

Overall performance. Table 1 compares FlexSQL against DSR-SQL and ReFoRCE. Across both Spider2-Snow and Spider2-SQLite, our method consistently achieves higher execution accuracy (Pass@1), test-time scalability (Pass@8, Majority@8), and answer diversity (micro accuracy) within each model size class. With gpt-oss-120b, FlexSQL reaches a Pass@1 of 55.15% on Spider2-Snow and 57.78% on Spider2-SQLite, yielding absolute improvements of 11.03% and 9.63% over the strongest respective baselines.

Parameter efficiency. FlexSQL remains competitive even with the smaller gpt-oss-20b backbone. On Spider2-SQLite, our 20b model achieves a Pass@1 of 50.37%, outperforming both 120b baselines (ReFoRCE: 45.19%, DSR-SQL: 48.15%). On Spider2-Snow, our 20b Pass@1 of 43.20% remains comparable to the strongest 120b baseline (44.12%).

Test-time scaling. Figure 3 shows Pass@K as we scale the sampling budget from $K=8$ to $K=64$ with gpt-oss-120b on Spider2-Snow. FlexSQL maintains a consistent lead over

Model Setup	Majority@8	Pass@8	Micro Acc.
gpt-oss-120b	64.44	78.52	49.69
w/o Python (SQL Only)	52.59	68.89	41.10
w/o diverse planning	55.56	83.70	50.23
w/o plan backtracking	61.48	80.00	48.64
w/o all repair	57.04	79.26	46.79
gpt-oss-20b	54.07	78.52	45.34
w/o Python (SQL Only)	42.22	60.00	36.08
w/o diverse planning	49.63	74.07	43.32
w/o plan backtracking	52.59	73.33	42.21
w/o all repair	51.85	75.37	44.39

Table 3: Ablation study on Spider2-SQLite (in %) evaluating the impact of the Python interpreter, diverse planning, and repair.

Method	Precision	Recall	F1
Ours + oss-120b (Best of 8)	95.46	95.06	95.26
Ours + oss-120b (First plan)	87.26	89.13	88.19
CHES + oss-120b	69.21	73.36	71.22
ReFoRCE + o4 + o3 + o4 mini *	66.83	99.73	80.03
ReFoRCE + DeepSeek-V3 [†]	58.09	62.59	60.26
DSR-SQL + DeepSeek-V3 [†]	75.62	91.13	82.65

Table 4: Table-level schema linking on Spider2-Snow. *Best of 8*: best schema across $K=8$ plans. *First plan*: schema from the first plan only. *From Deng et al. (2025) GitHub release. [†]From Hao et al. (2025).

ReFoRCE across the entire range, confirming that our approach delivers reliable gains as test-time compute scales.

Leaderboard comparison. To quantify the practical impact of this scaling behavior, Table 2 compares FlexSQL against leading open-source entries on the Spider2-Snow leaderboard, including DSR-SQL with DeepSeek-R1 and ReFoRCE with gpt-o3. Our base gpt-oss-120b model with $K=8$ already achieves a competitive 59.74%. Scaling to $K=16$ yields 65.44%, surpassing both DSR-SQL with DeepSeek-R1 (63.80%) and ReFoRCE with gpt-o3 (62.89%). This demonstrates that FlexSQL can match or exceed open-source systems that use stronger reasoning models.

5.2 Analysis

To understand the contribution of each design choice, we ablate bilingual generation, diversity-enforced planning, and flexible repair on the Spider2-SQLite split using both gpt-oss-120b and gpt-oss-20b (Table 3).

Effectiveness of diversity-enforced planning. We ablate the verbalized diversity constraint by letting the agent generate K independent plans without enforcing variation. This reduces Majority@8 from 64.44% to 55.56% for the 120b model and from 54.07% to 49.63% for the 20b model, confirming that explicitly prompting for varied interpretations improves majority-vote quality. By generating plans in batches and enforcing diversity, the agent avoids converging on a single, potentially flawed interpretation, giving majority voting a broader set of candidates to draw consensus from. Interestingly, for the 120b model, removing diverse planning increases Pass@8 from 78.52% to 83.70%, revealing an exploration–exploitation trade-off: enforcing diversity strengthens majority consensus but can occasionally lead the model to overcomplicate its reasoning at the expense of simpler, viable answers.

The role of bilingual generation. As shown in Table 3, removing the Python interpreter (“SQL Only”) causes the largest drop across all ablations: Majority@8 falls from 64.44% to 52.59% for the 120b model and from 54.07% to 42.22% for the 20b model.

Table 5 provides a finer-grained view by categorizing solved questions according to which languages produced correct answers. A large proportion fall into the “Both” category (47.1% for the 120b model and 69.9% for the 20b model on Spider2-SQLite), indicating that Python consistently generates viable solutions alongside SQL. Moreover, Python exclusively solves up to 27.6% of queries on Spider2-SQLite. To-

Dataset	Model	Only Python	Only SQL	Both
Spider2-Snow	120b	7.1%	48.8%	44.1%
	20b	5.5%	61.0%	33.5%
Spider2-SQLite	120b	27.6%	25.3%	47.1%
	20b	23.3%	6.8%	69.9%

Table 5: Breakdown of solved questions by language for Majority@8 groups on Spider2-Snow and Spider2-SQLite. SQL/Python ratio within “Both” is in Appendix B.

gether, these results show that the two languages are complementary: the bilingual toolkit expands coverage by providing multiple pathways to the correct answer.

The impact of repair and backtracking. Disabling the repair mechanism, which removes both code-level repair and plan-level backtracking, reduces Majority@8 from 64.44% to 57.04% (120b) and from 54.07% to 51.85% (20b). Without repair, the system can neither fix implementation errors (e.g., wrong aggregation or column references) nor backtrack to the planning stage when deeper reasoning errors are detected (e.g., wrong table selection or missing joins).

Schema linking performance. Table 4 reports table-level schema linking on Spider2-Snow. Existing approaches retrieve schema upfront: ReFoRCE and DSR-SQL include all candidate tables and fall back to independent table filtering on overflow, while CHESS (Talaie et al., 2024) applies column-level filtering followed by table identification, incurring the high computational cost of a separate language model call for each column. In contrast, FlexSQL explores the database incrementally and retrieves schema elements on demand, yielding substantially higher precision with comparable recall and the best overall F1. Even the first generated plan attains competitive F1, suggesting that flexible exploration can reduce context overhead while preserving accurate schema grounding.

Python to SQL accuracy. Table 6 reports transpilation accuracy on the 93 questions whose majority-vote answer was produced exclusively by Python across our Spider2-Snow and Spider2-SQLite experiments. The framework achieves 96.77% accuracy. Manual inspection shows these cases rely on Python features like regular expressions and analytical libraries to build

Source Model	Spider2-Snow	Spider2-SQLite
gpt-oss-20b	15/15	17/17
gpt-oss-120b	22/23	23/24
gpt-oss-120b (pass@16 (MV))	13/14	–
Transpiled with gpt-oss-20b	81/93 $\approx$ 87.09%	
Overall accuracy	90/93 $\approx$ 96.77%	

Table 6: Transpilation accuracy of our framework on all experiments with gpt-oss models on Spider2-Snow and Spider2-SQLite.

intermediate solutions before conversion to SQL. Our transpilation pipeline uses a two-tier approach, an initial pass with gpt-oss-20b, then a second pass with gpt-oss-120b on remaining failures. The 20b model alone handles $\approx$ 87% of cases, covering 40/41 instances in Spider2-SQLite, while the larger model is needed mainly for Snowflake-specific SQL syntax. The three unsolved examples appear in Appendix C.

5.3 Augmenting General-Purpose Coding Agents with FlexSQL Paradigm

As general-purpose coding agents such as Claude Code gain wide adoption, we bring FlexSQL’s flexible exploration and execution paradigm to Claude Code to evaluate its effectiveness beyond our controlled setting. We package FlexSQL’s tool suite and exploration instructions as Claude skills, encouraging the agent to explore and execute flexibly. As a baseline, we use the same agent equipped with a single query-execution tool. In this experiment, we use Claude Sonnet 4.6 with medium thinking effort, generating one answer per question.

Spider2-Snow	Pass@1	Average execution time (sec)
Claude Code	57.7	216.5
with FlexSQL harness	66.0	288.5

Table 7: Performance of Claude Code with our custom FlexSQL harness on Spider2-Snow.

Table 7 shows our harness performance on Spider2-Snow. The harness yields an absolute improvement of 8.3% (57.7% $\rightarrow$ 66.0% Pass@1) over the baseline. This indicates that flexible exploration and execution are beneficial beyond our original setup, continuing to provide consistent gains when integrated into a widely used general-purpose agent. Though FlexSQL’s performance comes with the cost of increased average latency (216.5 seconds/question $\rightarrow$ 288.5 seconds/question), which mainly due to systematic database exploration and SQL execution, this is a worthwhile trade-off for performance-demanding applications.

6 Related Work

Text-to-SQL pipelines. Text-to-SQL has progressed from early statistical methods (Zelle & Mooney, 1996; Tang & Mooney, 2000) to language model based systems (Liu et al., 2025; Hong et al., 2025), evaluated on increasingly realistic benchmarks (Yu et al., 2018; Li et al., 2023; Lei et al., 2025). Most modern approaches follow a fixed pipeline of schema linking, SQL generation, and post-hoc repair (Wang et al., 2020; Talaei et al., 2024; Li et al., 2025a; Hao et al., 2025; Deng et al., 2025; Pourreza et al., 2025), with extensions for planning (Sumers et al., 2024; Pourreza & Rafiei, 2023b; Wang et al., 2025; Li et al., 2025b) and candidate diversity (Liu et al., 2026; Lee et al., 2025; Gao et al., 2023). However, database interactions in these systems are confined to predetermined stages: the schema is retrieved once upfront, and execution feedback is used only for final validation. FlexSQL instead provides exploration tools throughout reasoning, enabling flexible backtracking when downstream errors reveal incorrect schema assumptions.

Agentic reasoning. Our work builds on the broader tradition of interleaving reasoning with environment interaction. ReAct (Yao et al., 2022) showed that alternating reasoning traces with actions improves grounding, and subsequent agent frameworks have applied this to code repositories (Yang et al., 2024), operating systems (Wu et al., 2024), and interactive coding environments (Yang et al., 2023). Revising or backtracking on earlier decisions based on execution feedback is a well-established strategy in agentic systems (Huang et al., 2022; Shinn et al., 2023; Madaan et al., 2023; Shi et al., 2026; Song et al., 2026); FlexSQL instantiates this principle for database exploration, where backtracking targets not just code-level errors but also upstream schema and plan assumptions.

Intermediate representations. Intermediate representations between natural language and SQL have been shown to improve text-to-SQL generation (Wang et al., 2017; Yaghmazadeh et al., 2017; Guo et al., 2019; Gan et al., 2021) and error correction (Chen et al., 2023). Python has emerged as an effective intermediate language for code generation owing to its expressiveness and strong representation in language model training data (Bhatia et al., 2024; Chi et al., 2025). FlexSQL adopts this idea, allowing the agent to implement solutions in Python and translate them back to SQL.

7 Conclusion

We presented FlexSQL, a text-to-SQL agent that replaces fixed pipeline stages with tools for on-demand database exploration throughout reasoning. This flexibility enables diverse planning, plan-level backtracking, and bilingual generation to each operate over up-to-date schema context, and our ablations confirm that their gains compound when combined. On Spider2.0, FlexSQL outperforms strong baselines across model scales using open-source models with 20B and 120B parameters, and exhibits effective test-time scaling with increased sampling budgets. Furthermore, integrating our paradigm into general-purpose coding agents like Claude Code yields consistent performance gains, demonstrating the FlexSQL’s architecture generalization across diverse agent platforms.

Acknowledgments

This work is supported in part by a Canada CIFAR AI Chair award to XY and a Canada CIFAR AI Chair award to JQC. This research has also been enabled in part by support provided by the Digital Research Alliance of Canada.

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

- Anthropic. Claude code, 2026. URL <https://claude.com/product/claude-code>. Agentic coding tool.
- Sahil Bhatia, Jie Qiu, Niranjana Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Verified code transpilation with llms. In *Proceedings of the 38th International Conference on Neural Information Processing Systems, NIPS '24*, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.
- Ziru Chen, Shijie Chen, Michael White, Raymond Mooney, Ali Payani, Jayanth Srinivasa, Yu Su, and Huan Sun. Text-to-SQL error correction with language models of code. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 1359–1372, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-short.117. URL <https://aclanthology.org/2023.acl-short.117/>.
- Yongdong Chi, Hanqing Wang, Yun Chen, Yan Yang, Jian Yang, Zonghan Yang, Xiao Yan, and Guanhua Chen. Pi-SQL: Enhancing text-to-SQL with fine-grained guidance from pivot programming languages. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, Suzhou, China, 2025.
- Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. ReForCE: A text-to-SQL agent with self-refinement, format restriction, and column exploration. In *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*, 2025. URL <https://openreview.net/forum?id=OuFfDBwQd>.
- Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. Natural SQL: Making SQL easier to infer from natural language specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 2030–2042, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.174. URL <https://aclanthology.org/2021.findings-emnlp.174/>.
- Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *CoRR*, abs/2308.15363, 2023.
- Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards complex text-to-SQL in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4524–4535, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1444. URL <https://aclanthology.org/P19-1444/>.
- Zhifeng Hao, Qibin Song, Ruichu Cai, and Boyan Xu. Text-to-sql as dual-state reasoning: Integrating adaptive context and progressive generation. *arXiv preprint arXiv:2511.21402*, 2025.
- Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. Next-generation database interfaces: A survey of llm-based text-to-sql. *IEEE Transactions on Knowledge and Data Engineering*, 2025.
- Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. Inner monologue: Embodied reasoning through planning with language models. In *arXiv preprint arXiv:2207.05608*, 2022.
- Nan Huo, Xiaohan Xu, Jinyang Li, Per Jacobsson, Shipeng Lin, Bowen Qin, Binyuan Hui, Xiaolong Li, Ge Qu, Shuzheng Si, Linheng Han, Edward Alexander, Xintong Zhu, Rui Qin, Ruihan Yu, Yiyao Jin, Feige Zhou, Weihao Zhong, Yun Chen, Hongyu Liu, Chenhao Ma, Fatma Ozcan, Yannis Papakonstantinou, and Reynold Cheng. BIRD-INTERACT: Re-imagining text-to-SQL evaluation via lens of dynamic interactions. In *The Fourteenth International Conference on Learning Representations*, 2026.

- Rocky Klopfenstein, Yang He, Andrew Tremante, Yuepeng Wang, Nina Narodytska, and Haoze Wu. Spotit: Evaluating text-to-SQL evaluation with formal verification. In *The Fourteenth International Conference on Learning Representations*, 2026.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
- Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. MCS-SQL: Leveraging multiple prompts and multiple-choice selection for text-to-SQL generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 337–353, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics. URL <https://aclanthology.org/2025.coling-main.24/>.
- Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows. In *The Thirteenth International Conference on Learning Representations*, 2025.
- Boyan Li, Chong Chen, Zhujun Xue, Yinan Mei, and Yuyu Luo. Deepeye-sql: A software-engineering-inspired text-to-sql framework. *arXiv preprint arXiv:2510.17586*, 2025a.
- Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. Alpha-SQL: Zero-shot text-to-SQL using monte carlo tree search. In *Forty-second International Conference on Machine Learning*, 2025b. URL <https://openreview.net/forum?id=kGg1ndttmI>.
- Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, 8(1):73–84, September 2014. ISSN 2150-8097. doi: 10.14778/2735461.2735468. URL <https://doi.org/10.14778/2735461.2735468>.
- Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. Codes: Towards building open-source language models for text-to-sql. *Proc. ACM Manag. Data*, 2(3), May 2024. doi: 10.1145/3654930. URL <https://doi.org/10.1145/3654930>.
- Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiayi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, et al. Can llm already serve as a database interface. *A big bench for large-scale database grounded text-to-sqls*. *CoRR*, abs/2305.03111, 2023.
- Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. A survey of text-to-sql in the era of llms: Where are we, and where are we going? *IEEE Transactions on Knowledge and Data Engineering*, 2025.
- Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, et al. Xiyan-sql: A novel multi-generator framework for text-to-sql. *IEEE Transactions on Knowledge and Data Engineering*, 2026.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.
- Mohammadreza Pourreza and Davood Rafiei. Evaluating cross-domain text-to-SQL models and benchmarks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023a.
- Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *arXiv preprint arXiv:2304.11015*, 2023b.

- Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Serkan O Arik. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. In *The Thirteenth International Conference on Learning Representations*, 2025.
- Taiwei Shi, Sihao Chen, Bowen Jiang, Linxin Song, Longqi Yang, and Jieyu Zhao. Experiential reinforcement learning. *arXiv preprint arXiv:2602.13949*, 2026.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.
- Yuda Song, Lili Chen, Fahim Tajwar, Remi Munos, Deepak Pathak, J Andrew Bagnell, Aarti Singh, and Andrea Zanette. Expanding the capabilities of reinforcement learning via text feedback. *arXiv preprint arXiv:2602.02482*, 2026.
- Theodore Sumers, Shunyu Yao, Karthik R Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL <https://openreview.net/forum?id=1i6ZCvf1QJ>. Survey Certification, Featured Certification.
- Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*, 2024.
- Lappoon R. Tang and Raymond J. Mooney. Automated construction of database interfaces: Intergrating statistical and relational learning for semantic parsing. In *2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pp. 133–141, Hong Kong, China, October 2000. Association for Computational Linguistics. doi: 10.3115/1117794.1117811. URL <https://aclanthology.org/W00-1317/>.
- Lukas Twist, Jie M Zhang, Mark Harman, Don Syme, Joost Noppen, Helen Yannakoudakis, and Detlef Nauck. A study of llms’ preferences for libraries and programming languages. *arXiv preprint arXiv:2503.17181*, 2025.
- Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7567–7578, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.677. URL <https://aclanthology.org/2020.acl-main.677/>.
- Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. MAC-SQL: A multi-agent collaborative framework for text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 540–557, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics. URL <https://aclanthology.org/2025.coling-main.36/>.
- Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. *SIGPLAN Not.*, 52(6):452–466, June 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062365. URL <https://doi.org/10.1145/3140587.3062365>.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2023.
- Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhounianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456*, 2024.
- Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi: 10.1145/3133887. URL <https://doi.org/10.1145/3133887>.

- John Yang, Akshara Prabhakar, Karthik R Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL <https://openreview.net/forum?id=fvKaLF1ns8>.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL <https://openreview.net/forum?id=mXpq6ut8J3>.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.
- Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 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*, 2018.
- John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2, AAAI’96*, pp. 1050–1055. AAAI Press, 1996. ISBN 026251091X.
- Jiayi Zhang, Simon Yu, Derek Chong, Anthony Sicilia, Michael R Tomz, Christopher D Manning, and Weiyan Shi. Verbalized sampling: How to mitigate mode collapse and unlock llm diversity. *arXiv preprint arXiv:2510.01171*, 2025.

Dataset	Model	Only Python	Only SQL	Both		
				Total	Gen: Py	Gen: SQL
Spider2-Snow	120b	7.1%	48.8%	44.1%	37.2%	62.8%
	20b	5.5%	61.0%	33.5%	31.8%	68.2%
Spider2-SQLite	120b	27.6%	25.3%	47.1%	43.7%	56.3%
	20b	23.3%	6.8%	69.9%	47.7%	52.3%

Table 8: Questions solvability breakdown considering the Pass@1 major vote groups on Spider2-Snow and Spider2-SQLite datasets. *Gen: Py* and *Gen: SQL* respectively denotes the portion of Python and SQL programs among the groups.

A Building Context

Before generating execution plans, FlexSQL performs a preprocessing step to provide the agent with a compact yet informative representation of the database. Following prior work (Deng et al., 2025; Hao et al., 2025), we apply three metadata compression techniques:

- **Null Column Pruning:** We remove columns that contain only NULL values across the entire dataset.
- **Structural Table Grouping:** Tables that share identical schemas but different time-suffixes (e.g., GA_SESSION_20240101, GA_SESSION_20240201) are grouped together.
- **External Knowledge Extraction:** If external documents are provided, a summarization agent extracts domain knowledge relevant to the query.

B Generation ratio of Python and SQL

Table 8 shows the breakdown of programming languages (Python and SQL) used within the majority groups of solved questions.

C Error Analysis on Python-to-SQL Transpilation

We manually inspect three failure cases in Table 6 to understand reasons for unsuccessful Python-to-SQL transpilation. These cases come from two benchmark questions: question sf_bq194 in Spider2-Snow and question local300 in Spider2-SQLite.

Case 1: sf_bq194 (Imperative parsing and language-specific containers). This question requires extracting the second most frequently imported library or module across Python (.py), R (.r, .Rmd), and IPython notebook (.ipynb) files. The Python implementation handles each format through separate helper functions, as outlined in the following pseudocode:

```

1: for each file in dataset do
2:   if extension is .py then
3:     parse import statements line by line
4:   else if extension is .ipynb then
5:     deserialize JSON → extract code cells → parse imports
6:   else if extension is .r or .Rmd then
7:     match library()/require() and :: via regex
8:   end if
9:   accumulate module names into a frequency counter
10: end for
11: return second most frequent entry from counter

```

This logic challenges transpilation for two reasons. First, each file format requires a fundamentally different parsing strategy: .py files need line-by-line tokenization of import

statements, `.ipynb` files must first be deserialized from JSON before their code cells can be scanned, and `.r/.Rmd` files rely on regex matching for `library()` calls and `::` operators. Encoding all three strategies in SQL would require deeply nested CASE expressions interleaved with dialect-specific string functions, a translation that none of the LLMs attempted correctly. Second, the program identifies the second most frequent module using `collections.Counter.most_common()`, a Python-specific API with no direct SQL counterpart. While counting itself maps naturally to GROUP BY with COUNT, selecting specifically the second-ranked entry requires an additional ranking step (e.g., `ROW_NUMBER()` or `LIMIT/OFFSET`). The LLMs failed to compose this ranking correctly on top of the already complex parsing logic.

Case 2: local300 (Stateful recurrence and multi-stage aggregation). This question asks to compute daily balances over each customer’s full transaction period, clamping negative balances to zero, and then report, for each month, the sum of every customer’s highest daily balance in that month. The central difficulty is that the daily balance is defined by a recurrence:

$$\text{balance}[t] = \max(\text{balance}[t-1] + \text{amount}[t], 0) \quad (1)$$

The $\max(\cdot, 0)$ clamp makes this fundamentally different from a cumulative sum. To illustrate, consider a customer with daily transaction amounts $[+5, -8, +3]$. A cumulative sum would yield running balances of $[5, -3, 0]$, whereas the recurrence in Eq. (1) produces $[5, 0, 3]$: the balance resets to zero on day 2, so day 3 starts from zero rather than from -3 . Because each day’s output depends on the *clamped* result of the previous day, this cannot be expressed with a `SUM() OVER (ORDER BY date) window function`; instead, it requires a **recursive CTE** that materializes one row per customer per day, carrying forward the clamped balance.

In all three transpilation attempts, the LLMs used a window-function-based cumulative sum, producing the $[5, -3, 0]$ trajectory rather than the correct $[5, 0, 3]$. The task is further complicated by multi-stage aggregation that follows the recurrence: daily balances must be grouped by customer and month to extract each customer’s monthly maximum, and those maxima must then be summed across all customers. Each stage introduces a different grouping key and aggregation target, and the LLMs struggled to keep these consistent throughout the translation.

D Examples of Python’s advantages on Spider2-Snow analytical questions

While Snowflake SQL can express complex analytics through specialized extensions, LLMs can generate more reliable solutions in Python for certain classes of analytical queries. This advantage stems from two factors. First, Python’s procedural paradigm aligns naturally with LLMs’ autoregressive generation: complex reasoning decomposes into sequential steps, whereas SQL demands that the same logic be expressed declaratively in a single, often heavily nested statement. Second, Python’s rich data processing libraries (e.g., `pandas`, `numpy`) is prevalent in LLM training data (Bhatia et al., 2024; Twist et al., 2025), yielding strong out-of-the-box proficiency. The following examples illustrates where these advantages are particularly clear.

sf.local1279 requires simulating a monthly inventory system throughout 2019, where each month’s ending stock depends on the previous month’s result and a conditional restocking rule: if inventory falls below a product-specific minimum, a fixed purchase quantity is added. The task then asks for the month per product where ending inventory is closest to the minimum threshold. In Python, this is a straightforward loop with an accumulator variable and an `if/else` branch, naturally expressing the sequential state dependency. In SQL, the same logic demands a recursive CTE that propagates inventory forward month by month, embedding the conditional restock decision within a CASE WHEN expression and carrying mutable state as a column across recursive iterations. Crucially, the recursion cannot be replaced by a window function, because each month’s restock decision alters all subsequent inventory levels, making the running state path-dependent rather than purely cumulative.

Tool	20b-snow	120b-snow	20b-lite	120b-lite
GetSchema	280	251	–	–
GetColumns	18,538	27,355	7,295	8,649
GetColValues	3,083	6,553	565	1,131
FindRows	4,880	9,400	292	677
SQLExecutor	37,261	44,297	4,838	5,138
PythonExecutor	21,996	27,051	24,303	9,194
Total	86,038	114,907	37,293	24,789

Table 9: Number of tool calls for gpt-oss-120b and gpt-oss-20b in our experiments with Pass@8 on Spider2-Snow and Spider2-SQLite.

sf_local002 asks the system to forecast toy sales for four dates in December 2018 by fitting a simple linear regression to historical daily sales and then computing five day symmetric moving averages over the predicted values. In Python, pandas supports date range generation and aggregation, while `numpy.polyfit` fits the regression in a single call, and the rolling window can be applied directly using `pandas.rolling`. Snowflake SQL can express this using built in `REGR_SLOPE` and `REGR_INTERCEPT` aggregate functions together with a recursive CTE for calendar generation. However, functions like `REGR_SLOPE` are platform specific extensions, and their availability and syntax vary across database systems. Allowing a language model to flexibly generate in Python gives it a broader set of tools to draw from, which could lead to better outcomes on tasks involving non standard analytical operations.

E Statistics of tool calls

Table 9 details the number of tool calls during our Pass@8 experiments. Generally, the language models prefer the two code execution tools due to their inherent flexibility. Among the tools, `GetSchema` is exclusive to data warehouse systems like Snowflake, where tables are organized into schemas (Section 3), and only 144 questions in Spider2-Snow require accessing multiple schemas, accounting for its low count relative to other tools.

F Heuristic rules for transpilation

Our heuristic rules for transpiling Python to SQL are summarized as follows:

1. **Runtime Type Preservation.** If the Python program’s behavior depends on runtime types (e.g., `isinstance(x, str)`, `dict`, `list`, `tuple`), the SQL translation must preserve that behavior. For instance, do not use string operations unless the runtime value is a string.
2. **VARIANT Column Unnesting.** The SQL translation can handle (semi-)structured data types (e.g., Snowflake’s `VARIANT` and nested objects/arrays). When a Python code snippet defines a helper function that:
 - takes a single column value as input,
 - handles null / missing values by returning an empty list,
 - parses a JSON string or `VARIANT`-like value using `json.loads`,
 - converts a JSON array or dictionary into a Python list, and
 - is applied row-wise via `Series.apply`, `map`, or equivalent,

this snippet is unnesting a Snowflake `VARIANT` column and should be translated into:

```
(LATERAL) FLATTEN(input => COLUMN | PARSE_JSON(COLUMN))
```

3. **Identifier vs. String Distinction.** The SQL translation must clearly distinguish quoted identifiers from strings and regular expressions.

4. **Regex Semantics Alignment.** Python and Snowflake use different regex semantics. When translating `str.contains(REGEX, case=bool)` to SQL, use:

```
REGEXP_LIKE(str, '.*' || REGEX || '.*', flags)
```

You *must* prepend and append `'.*'` to REGEX to ensure semantic equivalence. If `case=False`, use the flag `'is'` (not `'i'` alone) in `REGEXP_LIKE` to ensure consistency with Python’s regex behavior.

5. **VARIANT Data Access Paths.** When writing SQL queries that access VARIANT columns in Snowflake (i.e., columns where each value is a JSON object), you must specify the full access path, including nested keys and array indices:
- use single quotes for case-sensitive key access, e.g., `"col": 'KeyName'`,
 - chain keys for nested object access, e.g., `"col": 'OuterKey': 'InnerKey'`,
 - use zero-based indexing for array element access, e.g., `"col"[0]`,
 - use `GET_PATH("col", "dynamic.key.string")` for dynamic key access.
- Always include the *full* path to the target value (e.g., `"col": 'key': 'sub'[0]`) rather than just the base column name.
6. **JSON String Columns.** For columns whose values are JSON-formatted strings, first apply `PARSE_JSON()` to convert the string into a JSON object, then use the access rules in Rule 5.

G Inference Configurations

In all of our experiments, we set up local vLLM instances (Kwon et al., 2023) using at most 4 Nvidia L40s GPUs to serve gpt-oss models (Agarwal et al., 2025) with `"reasoning_effort" = "high"` and `"temperature" = 1.0`.

H Additional discussions

Evaluation scope. Our main evaluation focuses on Spider2.0 due to its structural complexity and strong grounding in enterprise-level databases. While foundational benchmarks such as Spider (Yu et al., 2018) remain valuable for evaluating standard text-to-SQL generation, their smaller-scale environments abstract away many real-world challenges, making them less indicative of how agentic methods perform in practice. By contrast, Spider2.0 introduces massive schemas and dialect-specific features that more effectively stress-test advanced reasoning and planning.

Comparison with closed-source pipelines. To ensure reproducibility and facilitate controlled comparisons under a shared model setting, we restrict our baselines to open-source approaches with publicly available code and literature. We do not benchmark against several top entries on the Spider2.0 leaderboard that rely on proprietary models or undisclosed system designs. The limited transparency regarding their underlying architectures, prompting strategies, and compute budgets makes it difficult to establish fair baseline settings for comparison.