

Cheaper, Better, Faster, Stronger: Robust Text-to-SQL without Chain-of-Thought or Fine-Tuning

Yusuf Denizay Dönder^{1*}, **Derek Hommel^{1*}**, **Andrea W Wen-Yi²**,

David Mimno², **Unso Eun Seo Jo^{1,2}**

¹Gena Co.; ²Cornell University

denizay@gena.co

Abstract

LLMs are effective at code generation tasks like text-to-SQL, but is it worth the cost? Many SOTA approaches use non-task-specific LLM approaches including Chain-of-Thought (CoT), self-consistency, and fine-tuning models. These methods can be costly for inference, sometimes requiring over a hundred LLM calls with reasoning, incurring average costs of up to \$0.46 per query, while fine-tuning models can cost up to thousands of dollars. We introduce “*N*-rep” consistency, a more cost efficient text-to-SQL approach that achieves similar BIRD benchmark scores as other more expensive methods, only costing \$0.039 per query. *N*-rep leverages multiple representations of the same schema input to mitigate weaknesses in any single representation, making the solution more robust and allowing the use of smaller and cheaper models without any reasoning or fine-tuning. To our knowledge, *N*-rep is the best performing text-to-SQL approach in its cost range.

1 Introduction

While LLMs have boosted SOTA performance on enduring code-generation tasks such as text-to-SQL, these approaches remain well below human-level performance (Li et al., 2024).

Current LLM text-to-SQL approaches rely on two computationally expensive techniques: Chain-of-Thought (CoT) self-consistency (Wang et al., 2023) and fine-tuning (Gao et al., 2025). Chain-of-Thought self-consistency uses CoT prompting to generate a pool of candidate queries, requiring many LLM calls (Talaei et al., 2024). Fine-tuning requires extensive resources for data labeling and model training. Combining both approaches amplifies the costs.

We introduce “*N*-rep” Consistency, a novel, cheaper, SQL-specific approach leveraging n different text representations of the database schema for

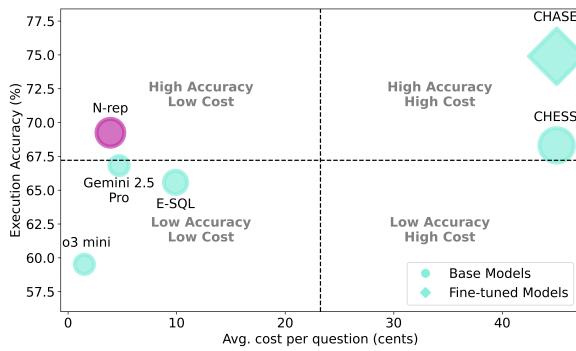


Figure 1: Comparison of Execution Accuracy (EX) and average cost (cents) per query for different models on the BIRD benchmark dev set. Shape sizes reflect the logarithmic scale of the number of LLM calls.

both schema linking and candidate generation. By utilizing various schema representations with different schema linking filtering, our method is robust to input sensitivity and schema linking errors while using fewer candidates than other consistency-based approaches. “*N*-rep” requires fewer tokens per candidate than reasoning methods and performs comparably to top-10 ranking approaches on the BIRD benchmark while costing nearly 10 times less per query, without fine-tuning expenses.¹ Open source package will be available upon acceptance.

2 Related Work

Schema Linking *Schema Linking* is the task of identifying the database tables and columns relevant to the user’s question. Maamari et al. (2024) leverage LLM long context windows to bypass a schema linking step. Qu et al. (2024) utilize SQL generation for schema linking by prompting an LLM to generate an SQL query using the full schema, then using the tables and columns from that prediction as inputs for the final SQL generation. Recent approaches utilize more elaborate multi-step schema linking methods that use

*Equal contribution.

¹bird-bench.github.io as of May 2025

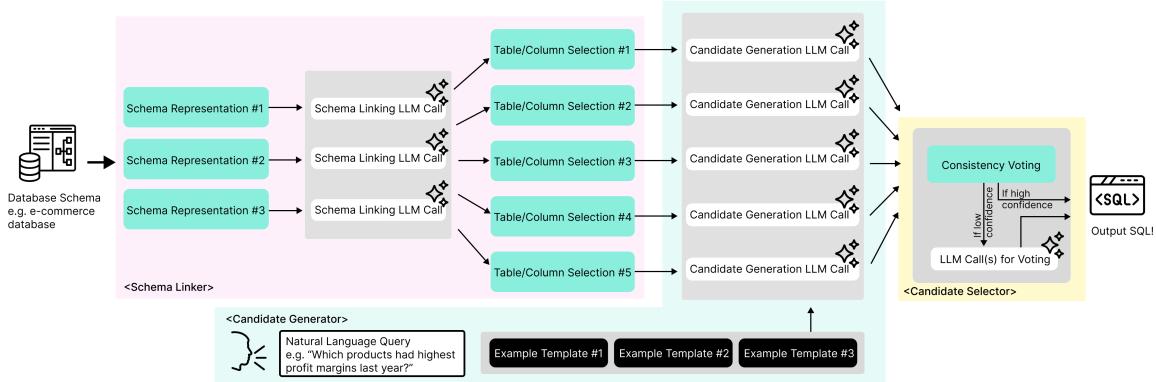


Figure 2: Overview of the N-rep approach for Text-to-SQL generation.

Model	Execution Accuracy	LLM Calls Typical(Avg.)	Tokens (K)	Cost (\$)
Qwen3[‡] 8B 14B 32B	44.98 53.46 56.71	1(1) 1(1) 1(1)	3.2 3.2 3.2	—
N-rep w/ Qwen3[‡] 8B 14B 32B	57.04 61.67 64.02	8(12.6) 8(12.0) 8(10.6)	36.8 33.9 31.7	—
o3-mini	59.52	1(1)	3.8	0.015
N-rep (ours)	69.25	8(10.9)	32.0	0.039
Gemini 2.5 Pro	66.82	1(1)	4.7	0.047
E-SQL	65.58	3(3)	3.7	0.09
CHASE [*]	74.9	— (100+)	—	0.44
CHESS	68.31	— (31.83)	319.9	0.46

Table 1: Execution Accuracy (EX) comparison of models along with average per query LLM calls, token usage (in thousands), and cost on the BIRD dev set. *Typical* shows median and the *Avg.* shows the mean number of calls. Same prompt with 3 few-shot samples used for base models without frameworks. Refer to Appendix A for details on configurations and cost calculations. [‡]Open-source model. ^{*}Model uses fine-tuning.

locality-sensitive hashing (LSH) to check keywords against database values and leverage external column description information when available (Talaei et al., 2024; Pourreza et al., 2025a; Gao et al., 2025).

Schema Representation How to format database schemas for LLMs is an area of active research. DAIL-SQL (Gao et al., 2023), DTS-SQL (Pourreza and Rafie, 2024), CHESS (Talaei et al., 2024) and E-SQL (Caferoglu and Ulusoy, 2024) use data description language (DDL); TA-SQL (Qu et al., 2024) use a dictionary with table.column keys and column description values; Maamari et al. (2024) use bulleted lists of tables, columns, types and sample values; DIN-SQL (Pourreza and Rafie, 2023) lists each table on one line with a list of its columns. MAC-schema (Wang et al., 2025) and

M-Schema (Gao et al., 2025) are semi-structured representations, with each table shown as a list of column tuples. Both MAC-schema and M-schema outperform the DDL format on various LLMs (Gao et al., 2025). See Appendix E for examples.

Self-consistency CoT self-consistency (Wang et al., 2023) combines CoT prompting (Wei et al., 2022) and temperature sampling to generate candidate answers from multiple reasoning paths before selecting the most “consistent” one. CHASE-SQL (Pourreza et al., 2025a) utilizes *multi-path candidate generation* with multiple CoT templates to generate 21 candidates, and uses a fine-tuned LLM to select the final SQL via pairwise comparison. CHESS generates 20 candidates via a structured CoT prompt and uses voting that evaluates each candidate against LLM-generated “unit tests”.

XiYan-SQL (Gao et al., 2025) utilizes multiple fine-tuned LLMs to generate 5 candidates, then another fine-tuned LLM for candidate selection.

3 Motivation

Our approach is motivated by a key observation: LLMs are sensitive to schema representation formats (Gao et al., 2025). Varying schema representations creates meaningful candidate diversity without temperature sampling or CoT reasoning. Generating multiple “schema linking candidates” is also simpler and more robust than other approaches that rely on extensive preprocessing and multiple LLM calls that are prone to error propagation from missing critical tables/columns.

4 Methodology

The N-rep Framework has three stages (Figure 2). **Schema Linker** utilizes multiple schema representations for schema linking. The results get passed to **Candidate Generator** to create multiple candidate SQL queries. Finally, the **Candidate Selector** uses a two-stage confidence-aware selection process to choose the final SQL query.

4.1 Schema Linker

This stage identifies tables and columns required to answer the natural language question (NLQ). First, target database schema is converted into n representations. Second, each representation, along with the NLQ and a fixed set of 3 hand-selected few-shot examples, is used to predict relevant tables and columns. Lastly, we produce three schema linking results for each representation — one using the full database schema (*no filtering*), one with only the selected tables and all their columns (*table-only*), and one with only the selected tables and columns (*full filtering*). See Appendix F for illustrations.

4.2 Candidate Generator

This stage generates multiple SQL candidates. An LLM takes a schema representation from previous stage — either fully filtered, table-only, or unfiltered—the NLQ, and 3 few-shot examples to generate a candidate SQL. N-rep retrieves few-shot samples with fully-filtered schema information from the training dataset following XiYan-SQL. See Appendix D.1 for the system prompt.

4.3 Candidate Selector

N-rep adopts a **confidence-aware** two-stage candidate selection strategy that combines regular

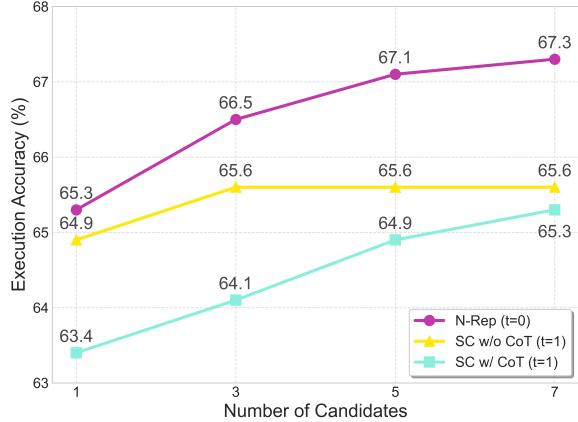


Figure 3: Comparison of N-rep, Self Consistency with CoT and Self Consistency without CoT. $t = 1$ means sampling temperature is 1.

self-consistency voting with CHASE-SQL’s LLM-based pair-wise voting.² Using the number of votes from regular voting as an indicator of confidence, N-rep applies LLM-based selection only for low confidence cases. Appendix B.2 shows the details for confidence threshold and the Appendix C.2 shows the relation between confidence and performance.

5 Experiments

5.1 Datasets and Metrics

We evaluate the N-rep framework on the BIRD (Li et al., 2024) and SPIDER (Yu et al., 2018) benchmarks. SPIDER is a cross-domain benchmark with 200 databases and 10,181 query pairs (7000 train, 1034 dev, 2147 test). BIRD contains 95 databases and 12,751 query pairs (9428 train, 1534 dev, 1789 test) that covers more than 37 real-world domains such as finance and medicine. For evaluation, we use Execution Accuracy (EX), the average number of queries where the predicted and ground truth SQL execution results are identical.

We compare N-rep (Table 1) to other publicly available solutions, and a selection of closed and open-source base models, including OpenAI o3-mini (OpenAI, 2025), Gemini 2.5 pro (Gemini Team, 2025), and Qwen3 (Yang et al., 2025). Base model tests use 3 few-shot samples from train sets following XiYan-SQL.

²In some cases, the LLM based selection is more effective, but it may require up to $2 \cdot \binom{M}{2}$ LLM calls. Each pairwise comparison is done twice by swapping the order to mitigate any order bias.

5.2 Experiment Setup

To achieve our best performing configuration of N-rep, we select 4 best schema representations based on schema linking experiments to optimize for precision-recall tradeoffs. For each candidate size n in Figure 3, we conduct a parameter search across all combinations of our 4 selected representations and 3 table filtering levels on a 10% dev subset, using regular voting to identify the highest-EX combination. All experiments in 5.3 and 5.4 use Gemini 1.5 Flash (Team et al., 2024) with 3 few-shot samples. Baseline and test configurations are described in Appendix A, and final N-rep configuration is shown in Appendix B.1.

5.3 Self-Consistency vs N-rep

We compare the performance of N-rep with greedy decoding against traditional self-consistency using temperature sampling with and without CoT in Figure 3. While EX of self-consistency without CoT plateaus after 3 candidates, EX of CoT self-consistency and N-rep both increase with the number of candidates. Moreover, for all candidate numbers tested, EX of N-rep is 2 percentage points higher than that of CoT self-consistency. Furthermore, Figure 4 shows that N-rep’s upper bound increases more than CoT self-consistency’s as candidates increase, suggesting that with better selection methods, N-rep’s performance over CoT self-consistency could further increase.

5.4 Candidate Selector

We compare the regular voting system, LLM-based voting system and our *confidence-aware* method. Table 2 shows confidence-aware final SQL selection can reduce the number of LLM calls by 60% while increasing the EX.

Selection Method	EX	LLM Calls
Regular Voting	67.2	0
LLM Based Voting	67.9	6178
Confidence-aware Voting	68.8	2436

Table 2: Execution Accuracy (EX) and number of LLM calls over the whole BIRD development set for different selection methods.

5.5 Cost Analysis

Compared to Gemini 2.5 Pro baseline, N-rep achieves 2.43 percentage points higher EX at a 17% cheaper price per query (\$0.039 vs \$0.047),

Model	Execution Accuracy
MiniSeek	91.2
CHESS	87.2
N-Rep (Ours)	87.0
DAIL-SQL	86.6

Table 3: Execution Accuracy (EX) comparison of models on the SPIDER test set.

Against CHESS, a CoT self-consistency method, N-Rep scores 0.94 higher EX while using only a third of LLM calls and 8.5% of the inference cost; against CHASE, a leading CoT self-consistency method using a fine-tuned candidate selection LLM, N-rep comes within 5.65 percentage points EX at only 8.7% of the inference cost (\$0.039 vs \$0.44) per query (Table 1).

6 Discussion

Our findings reveal that LLMs remain sensitive to schema representation for text-to-SQL, suggesting the models rely more on surface understanding than deep reasoning about database structures. The N-rep approach leverages this insight by combining multiple representations.

Our experiments with the Qwen3 models suggest that *N-rep reduces reliance on model scale*. Specifically, smaller models (e.g., Qwen3-8B) exhibit larger relative gains in EX when augmented with N-rep — improving from 44.98% to 57.04% EX, a jump of over 12 points — compared to a larger model like Qwen3-32B, which improved by 7.3 percentage points (Table 1).

To evaluate generalizability, we also tested N-rep on the SPIDER benchmark (Table 3), where it achieves 87.0 execution accuracy indicating that our solution is robust across benchmarks.

These results show how domain expertise — specifically, understanding the impact of schema representation variability — can outperform generic approaches like CoT self consistency that do not address this limitation (Figure 3).

7 Conclusion

We propose N-rep, a robust and cheaper framework that utilizes multiple text representations of database schemas. N-rep serves as a case study of how, for specific well-defined tasks with fixed input and output, using domain specific knowledge can minimize cost of LLM-based solutions.

Limitations

The requirement for using manually-selected schema representations limits the number of total candidates with unique representations, and our analysis is not exhaustive. Future work could examine a broader range of schema representations, and explore generating alternate forms with LLMs.

Also, our confidence-aware voting policy is currently hand-tailored for each specific number of candidates. Future work could optimize and automate this voting policy to scale to an arbitrary number of candidates.

Acknowledgments

Left blank for anonymous submission.

References

- Hasan Alp Caferoglu and Özgür Ulusoy. 2024. E-sql: Direct schema linking via question enrichment in text-to-sql. *arXiv preprint arXiv:2409.16751*.
- Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. [Text-to-sql empowered by large language models: A benchmark evaluation](#). *Preprint*, arXiv:2308.15363.
- Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yun-tao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. 2025. [A preview of xiyan-sql: A multi-generator ensemble framework for text-to-sql](#). *Preprint*, arXiv:2411.08599.
- Google Gemini Team. 2025. [Gemini 2.5 pro preview model card](#).
- Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.
- Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36.
- Karime Maamari, Fadhl Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The death of schema linking? text-to-sql in the age of well-reasoned language models. *arXiv preprint arXiv:2408.07702*.
- OpenAI. 2025. [Openai o3-mini system card](#).
- Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2025a. [CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL](#). In *The Thirteenth International Conference on Learning Representations*.
- Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: decomposed in-context learning of text-to-sql with self-correction. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.
- Mohammadreza Pourreza and Davood Rafiei. 2024. [DTS-SQL: Decomposed text-to-SQL with small large language models](#). In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 8212–8220, Miami, Florida, USA. Association for Computational Linguistics.
- Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wan, Hailong Li, Azalia Mirhoseini, Amin Saberi, Sercan Arik, and 1 others. 2025b. Reasoning-sql: Reinforcement learning with sql tailored partial rewards for reasoning-enhanced text-to-sql. *arXiv preprint arXiv:2503.23157*.
- Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. [Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-SQL generation](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 5456–5471, Bangkok, Thailand. Association for Computational Linguistics.
- Nils Reimers, Elliott Choi, Amr Kayid, Alekhya Nandula, Manoj Govindassamy, and Abdullah Elkady. 2023. [Introducing embed v3](#). Accessed: 2025-05-17.
- Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. [Chess: Contextual harnessing for efficient sql synthesis](#). *Preprint*, arXiv:2405.16755.
- Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, and 1 others. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.
- Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. [MAC-SQL: A multi-agent collaborative framework for text-to-SQL](#). In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 540–557, Abu Dhabi, UAE. Association for Computational Linguistics.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. [Self-consistency improves chain of thought reasoning in language models](#). In *ICLR 2023*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS ’22, Red Hook, NY, USA. Curran Associates Inc.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. *Qwen3 technical report*. Preprint, arXiv:2505.09388.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 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*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

A Methodology for performance and cost analysis

A.1 Models Chosen

For our comparison, along with our main Nrep solution, we included results for: three frameworks from the BIRD leaderboard, two closed-source models, and three versions of the Qwen3 family of models with and without our Nrep framework.

We selected CHESS and E-SQL because their authors shared code, enabling us to replicate results and measure token usage. For these, we used our own LLM call and token measurements, reporting EX values as published.

For CHASE, although the code was not available we referenced a graph that includes cost information in another paper Pourreza et al. (2025b) to estimate the cost.

We included OpenAI o3-mini and Gemini 2.5 Pro to evaluate out-of-the-box capabilities of closed-source reasoning models. The Qwen3 series was selected specifically because multiple model sizes are available, allowing us to analyze scaling in performance with our method.

A.2 Configurations

For Gemini 2.5 Pro and o3-mini, we conducted experiments using default settings with reasoning enabled. For the Qwen3 models, we disabled reasoning capabilities to ensure fair comparison between the base implementation and our Nrep approach.

Both the closed-source models and the Qwen3 models without Nrep used identical prompts with three few-shot samples for consistency, and the temperature was set to 0.

We use Cohere Multilingual v3 with "clustering" input type (Reimers et al., 2023) for all embeddings for few-shot retrieval.

All experimental results reported are based on a single run.

A.3 Cost Calculation

We calculated costs using current pricing models as of May 16, 2025. For the o3-mini experiments, we used Azure OpenAI Service pricing of \$1.10 per 1M input tokens and \$4.40 per 1M output tokens. For E-SQL, which used GPT-4o (Hurst et al., 2024), \$2.50 per 1M input tokens and \$10.00 per 1M output tokens.

For Gemini 2.5 Pro and CHESS, which uses Gemini 1.5 Pro, we applied Gemini Developer API pricing: \$1.25 per 1M input tokens and \$10.00 per 1M output tokens.

All calculations reflect full pricing without accounting for potential discounts from cache hits, providing a consistent basis for comparison across all models.

B Implementation details

B.1 Final configuration

Format	Filtering level	Model
MAC Schema	No Filtering	—
MAC Schema	Col. Filtering	GPT-4o
M-Schema	Table Filtering	GPT-4o
M-Schema	Full Filtering	GPT-4o
DDL	Full Filtering	Gemini 1.5 Pro

Table 4: Schema representations used in the final implementation. Note that the rows for *M-Schema with Table Filtering* and *M-Schema with Full Filtering* use the same prediction, corresponding to a single LLM call.

For our final configuration of Nrep, we used 5 candidates. Gemini 1.5 Flash was used for all candidate generations and LLM-based candidate selections for low confidence answers.

For the schema linking phase we used three formats: DDL format, MAC-Schema format, and M-Schema format, making 3 calls in total for the whole phase. We also used 2 different models

Format	Table			Column		
	Precision	Recall	F1	Precision	Recall	F1
MAC-Schema	92.9	93.1	93.0	78.1	87.2	82.4
SQL Alchemy	91.7	93.2	92.5	70.7	87.2	78.1
JSON Raw	91.8	92.9	92.4	74.9	87.2	80.6
DIN-SQL	90.8	93.2	92.0	74.9	86.9	80.4
M-Schema	91.8	93.1	92.5	75.5	87.1	80.9
DDL	92.2	93.6	92.8	75.5	86.9	82.5

Table 5: Micro precision and recall scores (in %) for different formats with GPT-4o

Format	Table			Column		
	Precision	Recall	F1	Precision	Recall	F1
MAC-Schema	71.5	96.1	82.0	37.6	94.0	53.7
M-Schema	68.1	95.4	79.5	35.6	92.4	51.4
DDL	69.2	95.7	80.3	39.2	92.7	55.1

Table 6: Micro precision and recall scores (in %) for different formats with Gemini 1.5 Pro

to further increase the variety of representations: Gemini 1.5 Pro and GPT-4o. We used different levels of filtering, again, to increase the range of representations. Table 4 shows the specific representations used for our configuration. The temperature is set to 0 for all models.

B.2 Candidate selection

In cases where vote distribution indicates low confidence we apply LLM-based comparison. In our five-candidate setup, we invoke the LLM selector when: all candidates receive only one vote, two candidates each receive two votes (indicating a tie among plausible options), the top has three votes, but another has two (indicating a strong second contender).

C Detailed Results

C.1 Schema Linking

Table 4 shows table and column performance on schema linking task for our six candidate schema representations. After selecting the best three formats by column F1, we did further tests with Gemini 1.5 Pro (Table 6).

C.2 Execution Accuracy Upper & Lower Bounds

Figure 4 shows upper and lower bounds for N-rep vs self-consistency for 1, 3, 5 and 7 candidates. The *upper bound* reflects the best-case scenario where a perfect candidate selector chooses the correct SQL query from the set, i.e. at least one candidate is

correct. The *lower bound* represents the worst-case selection, i.e. at least one candidate is wrong. The chance of at least one being wrong goes up as the number of candidates goes up and the chance of at least once being right goes up as the number of candidates goes up.

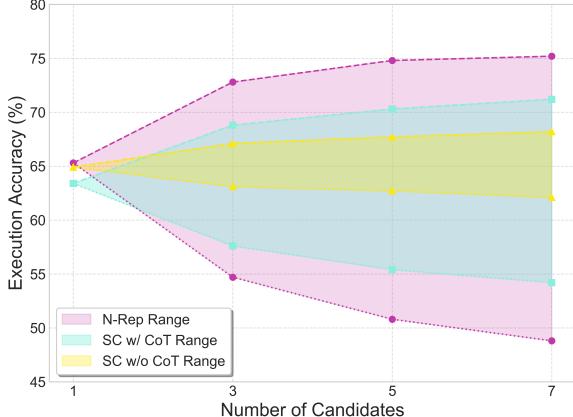


Figure 4: Upper and lower bounds of N-rep, Self Consistency with CoT and Self Consistency without CoT.

C.3 Execution Accuracy by Vote Count

As seen in Figure 5, the upper bound values increase with vote count, demonstrating that the number of votes assigned to the winning group corresponds to the likelihood that at least one candidate is correct, thus, higher confidence. The upper bound shows the maximum EX assuming perfect candidate selection. Regular selection and our confidence-aware selection are equivalent at count = 4; all three are all equal in the case of count = 5

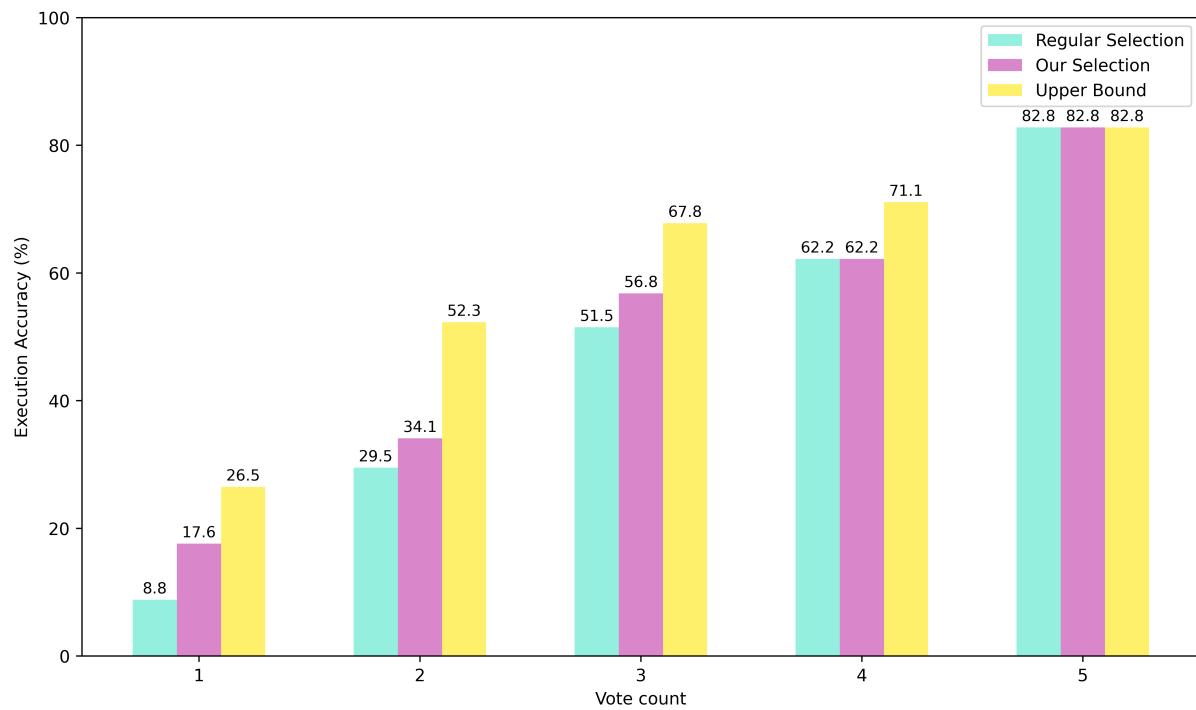


Figure 5: EX by vote count for the selected candidate

because all candidates have equivalent output.

D Prompt Details

D.1 Candidate Generator System Prompt

Figure 6 shows the system prompt for candidate generation.

INSTRUCTIONS:

You write SQL queries for a sqlite database. Users are querying their company database, and your task is to assist by generating valid SQL queries strictly adhering to the database schema provided. The user will provide you with a query intent, an SQL template, and optionally an hint to help create the correct SQL. They may also provide a set of examples similar to their query from other databases, which should guide your understanding and solution.

Translate the user's request into one valid SQLite query. SQL should be written as a markdown code block:

For example:

```
“‘sql
SELECT * FROM table WHERE condition;
“‘
```

Guidelines:

1. Schema Adherence:

- Use only tables, columns, and relationships explicitly listed in the provided schema.
- Do not make assumptions about missing or inferred columns/tables.

2. SQLite-Specific Syntax:

- Use only SQLite syntax. Be aware that SQLite has limited built-in date/time functions compared to other sql dialects.

3. Conditions:

- Always include default conditions for filtering invalid data, e.g., deleted_at IS NULL and status != 'cancelled' if relevant.
- Ensure these conditions match the query's intent unless explicitly omitted in the user request.

4. Output Consistency:

- The output fields must match the query's intent exactly. Do not add extra columns or omit requested fields.

5. Reserved Keywords and Case Sensitivity:

- Escape reserved keywords or case-sensitive identifiers using double quotes (" "), e.g., "order".

If the user's question is ambiguous or unclear, you must make your best reasonable guess based on the schema.

Translate the user's intent into a **single valid SQLite query** based on the schema provided.

Pay special attention to the examples given by the user.

Ensure the query is optimized, precise, and error-free.

You must ONLY output ONE SINGLE valid SQL query as markdown codeblock; do NOT output any other text.

Figure 6: candidate generator system prompt

E Schema Formats

We test six schema representations, M-Schema (Figure 7, MAC-Schema (Figure 8, DDL (Figure 9, DIN-SQL (Figure 10, a raw JSON representation (Figure 11, and a python representation (Figure 12.

```
[DB_ID] financial
[Schema]
# Table: account
[
  (account_id:INTEGER, Primary Key, Examples: [1, 2, 3]),
  (district_id:INTEGER, Examples: [18, 1, 5])
]
# Table: district
[
  (district_id:INTEGER, Primary Key, Examples: [1, 2, 3]),
  (A11:INTEGER, Examples: [12541, 8507, 8980])
]
# Table: loan
[
  (amount:INTEGER, Examples: [80952, 30276, 318480]),
  (status:TEXT, Examples: [A, B, D])
]
[Foreign keys]
account.district_id=district.district_id
```

Figure 7: XiYan-SQL M-SCHEMA format

```
# Table: account
[
  (account_id, account id.),
  (district_id, location of branch.)
]
# Table: district
[
  (district_id, location of branch.),
  (A11, average salary.)
]
# Table: loan
[
  (amount, amount.),
  (status, status. Value examples: ['C', 'A', 'D', 'B'].)
]
```

Figure 8: MAC-SCHEMA format

```
financial CREATE messages:

CREATE TABLE account (
    account_id INTEGER
    district_id INTEGER
    ,
    PRIMARY KEY (account_id)
    FOREIGN KEY (district_id) REFERENCES district (district_id)
);

CREATE TABLE district (
    district_id INTEGER
    A11 INTEGER
    ,
    PRIMARY KEY (district_id)
);

CREATE TABLE loan (
    amount INTEGER
    status TEXT
    ,
    PRIMARY KEY (loan_id)
);
```

Figure 9: DDL ("SQL CREATE") format

```
table 'account' with columns: account_id (INTEGER), district_id (INTEGER)
table 'district' with columns: district_id (INTEGER), A11 (INTEGER)
table 'loan' with columns: amount (INTEGER), status (TEXT)

Relations:
account.district_id -> district.district_id
```

Figure 10: DIN-SQL style format

```
{  
    "tables": {  
        "account": {  
            "columns": {  
                "account_id": "INTEGER",  
                "district_id": "INTEGER"  
            },  
            "keys": {  
                "primary_key": [  
                    "account_id"  
                ]  
            },  
            "foreign_keys": {  
                "district_id": {  
                    "referenced_table": "district",  
                    "referenced_column": "district_id"  
                }  
            }  
        },  
        "district": {  
            "columns": {  
                "district_id": "INTEGER",  
                "A11": "INTEGER"  
            },  
            "keys": {  
                "primary_key": [  
                    "district_id"  
                ]  
            },  
            "foreign_keys": {}  
        },  
        "loan": {  
            "columns": {  
                "amount": "INTEGER",  
                "status": "TEXT"  
            },  
            "keys": {  
                "primary_key": [  
                    "loan_id"  
                ]  
            },  
            "foreign_keys": {}  
        }  
    }  
}
```

Figure 11: raw JSON format

```
from sqlalchemy import Column, ForeignKey, Integer, MetaData, Table, Text, text

metadata = MetaData()

t_district = Table(
    'district', metadata,
    Column('district_id', Integer, primary_key=True, server_default=text('0')),
    Column('A11', Integer, nullable=False),
)

t_account = Table(
    'account', metadata,
    Column('account_id', Integer, primary_key=True, server_default=text('0')),
    Column('district_id', ForeignKey('district.district_id'), nullable=False, server_default=text('0')),
)

t_loan = Table(
    'loan', metadata,
    Column('amount', Integer, nullable=False),
    Column('status', Text, nullable=False) )
```

Figure 12: python SQLAlchemy format

F Filtering Levels

This shows examples of different schema filtering levels on a toy example dataset in the compact "DIN-SQL" format. The toy dataset has three tables in total, "users", "orders" and "products" (Figure 14). The example schema linking prediction output is a JSON-like dictionary; here it predicted we need to keep two tables, "users" and "orders", with two columns each (Figure 13). *No filtering* preserves all schema entities (Figure 14); *table-only filtering* drops the "products" table, but all columns from "users" and "orders" remain (Figure 15); *full filtering* removes all entities that are not in the prediction (Figure 16).

```
{
    "users": ["user_id", "name"],
    "orders": ["user_id", "order_date"]
}
```

Figure 13: Schema linker output

```
table 'users' with columns: user_id (INTEGER), name (TEXT), email (TEXT),
created_at (DATE)
table 'products' with columns: product_id (INTEGER), name (TEXT), price (DECIMAL),
stock (INTEGER)
table 'orders' with columns: order_id (INTEGER), user_id (INTEGER), product_id
(INTEGER), quantity (INTEGER), order_date (DATE)

Relations:
orders.user_id -> users.user_id
orders.product_id -> products.product_id
```

Figure 14: No filtering example (full schema)

```
table 'users' with columns: user_id (INTEGER), name (TEXT), email (TEXT),
created_at (DATE)
table 'orders' with columns: order_id (INTEGER), user_id (INTEGER), product_id
(INTEGER), quantity (INTEGER), order_date (DATE)

Relations:
orders.user_id -> users.user_id
```

Figure 15: Table-only filtering example

```
table 'users' with columns: user_id (INTEGER), name (TEXT)
table 'orders' with columns: user_id (INTEGER), order_date (DATE)

Relations:
orders.user_id -> users.user_id
```

Figure 16: Full filtering example