Current Limitations of LLM Guided Program Synthesis

Anonymous Author(s)

Abstract

Large Language Models (LLMs) exhibit strong performance for program synthesis tasks including text-to-SQL, up to 86.6% accuracy on the Spider 1.0 benchmark. The top-performing methods bridge the semantic gap between natural language and SQL by using prompt engineering, fine-tuning, or in-context learning techniques; however, these purely neural techniques remain prone to hallucinations. Existing symbolic approaches to improve accuracy on program synthesis tasks generally rely on post-hoc verification, which fails to constrain unproductive path exploration, or constrained decoding techniques that rely on the underlying distribution of the model. The effectiveness of symbolic guidance under complex queries and the semantic ambiguity inherent in text-to-SQL benchmarks, however, remains unclear. In this work, we study the *"ground truth"* errors made by LLMs in the text-to-SQL task looking at both fine-tuned models and general-purpose models optimized for code generation. We observe that errors can be grouped into five error categories, two of which are amenable to simple symbolic fixes. Importantly, we find that for many queries, correct predictions do not rank near the top of the LLMs likely predictions, implying that *constrained decoding alone is insufficient* to guide synthesis on hard problems. Our findings provide interpretable insight into where LLMs fail and suggest promising, neuro-symbolic directions for improving LLM guided program synthesis.

1 Introduction

[*Problem: We consider studying the kinds of incremental prediction errors LLMs make that take them down the incorrect path in program synthesis tasks.*]

[*Why Important: LLMs are used for program synthesis, but errors can compound if the LLM starts off on the wrong track.*] Large Language Models (LLMs) are increasingly used as program generators to synthesize working code from natural language prompts. In contrast to other program synthesis techniques, LLMs are scalable to larger and more complex programs; however, due to their autoregressive nature, they can get stuck down an incorrect path as *incremental prediction errors* can propagate through the decoding process which ultimately produces invalid programs.

[*In this paper we study the incremental errors LLMs make when synthesizing SQL queries. We categorize these mistakes to evaluate whether constrained decoding or other symbolic reasoning techniques an intervene during decoding and steer LLMs to correct queries.*]

[*LLMs predict queries token by token and constrained decoding enforces constraints at decoding time, so we focus our study on the incremental errors LLMs make.*] Recent work in LLM-augmented program synthesis has explored decoding-time interventions, including SQL grammar-constrained decoding (Arcadinho et al. 2022) and related constrained generation techniques (Nagy et al. 2026). Constrained decoding methods are typically evaluated only on fully generated queries; however, constrained decoding fundamentally operates at the lexeme level. To understand its effectiveness, we analyze the model's learned distribution for lexemes during decoding, since a constraint can only be enforced if the appropriate lexeme appears within the model's top-k candidates.

[*Existing SQL error taxonomies are based on fully generated queries, so we introduce a new taxonomy focusing on the incremental errors LLMs make.*] Existing text-to-SQL taxonomies analyze errors of fully generated queries (Qu et al. 2024). However, SQL synthesis is particularly dependent on lexeme-level correctness. In addition to overall query structure, SQL queries are dependent on correct schema references and correct operator usage. In contrast to existing taxonomies, we propose a taxonomy of incremental lexeme-level errors made during decoding.

[*A taxonomy of incremental SQL prediction errors then allows us to observe the potential benefit of symbolic repair and the limitations of constrained decoding for LLM-driven SQL synthesis.*] We evaluate the ability of LLMs to synthesize correct SQL queries at the lexeme level and find that we can broadly categorize lexeme level errors into six categories. We find that many errors within these categories can be mitigated with simple symbolic fixes and constrained decoding. We also observe limitations of constrained decoding. It cannot correct errors if the correct lexeme is absent from the model's top-k candidates.

Contributions:

- We present a hierarchical taxonomy of error categorization for lexeme level LLM mistakes during the decoding process.
- We identify error categories that are amenable to symbolic repair and quantify an upper bound on automatically cor-

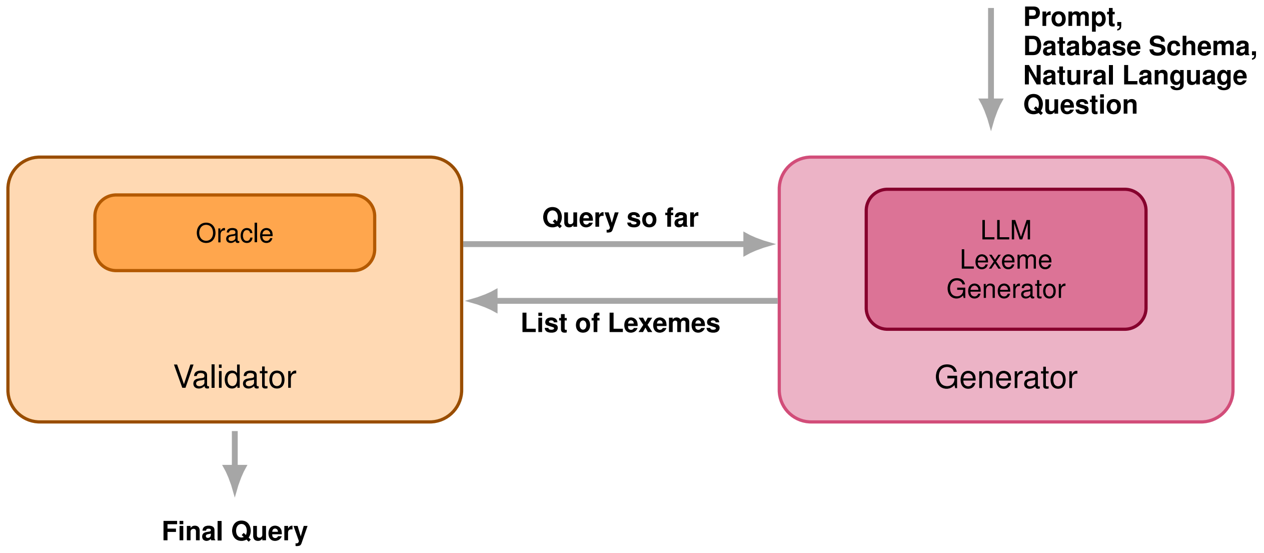

Figure 1: The overall system architecture. The Language Model Generator is prompted to generate a SQL query given a database schema and natural language question. The generator returns a list of lexemes to the Validator which incrementally validates the most probable lexeme against the gold one and corrects the generator if there is a mismatch.

rectable failures.

2 Related Work

[*Text-to-SQL is a task in which SQL queries are generated from NL questions, and many high performing methods use techniques such as prompt engineering and in-context learning.*] Text-to-SQL is a program synthesis task concerned with SQL generation for natural language questions. Recent research has focused on leveraging LLMs for this task and have performed well empirically, reaching test suite accuracies of up to 86.6% (Gao et al. 2024) against the Spider 1.0 dataset (Yu et al. 2018). Many of the methods that top the Spider 1.0 leaderboard involve prompt engineering and in-context learning strategies. These methods often treat LLMs as black box synthesizers but are able to scale to complex queries.

[*Traditional program synthesis techniques provide guarantees, but don't scale.*] Before the advent of LLMs, symbolic approaches to SQL synthesis explored enumerative search and program sketching to guarantee correct query generation (Wang, Cheung, and Bodik 2017; Takenouchi et al. 2021). These works focus on pruning the search space either through additional information provided by the user, such as the constants that are allowed to appear in a query (Wang, Cheung, and Bodik 2017; Takenouchi et al. 2021), or through pruning unproductive paths (Wang, Cheung, and Bodik 2017; Takenouchi et al. 2021). These traditional methods often include soundness and bounded completeness guarantees, but struggle to scale to larger, more complex queries.

[*Recent work to aid LLM synthesis focuses on enforcing constraints at decoding time.*] Recent work has focused on neuro-symbolic approaches to bridge the scalability of LLM aided program generation and more symbolic approaches. Constrained decoding methods aim to enforce certain properties during the decoding process. In SQL synthesis, techniques such as PICARD (Scholak, Schucher, and Bahdanau 2021) incrementally validate partial queries and prune token continuations that violate SQL grammar constraints, while other methods enforce context-free grammar constraints explicitly (Arcadinho et al. 2022). Other methods enforce semantic constraints (Nagy et al. 2026) or type safety (Mündler et al. 2025) more generally in program synthesis. Constrained decoding approaches improve syntactic and semantic correctness but depend on the correct token appearing within the model's top-k predictions based on its learned distribution.

[*There are error taxonomies for text-to-SQL, but they consider fully generated queries.*] Prior work has been done to analyze errors for text-to-SQL, but they are concerned with categorizing fully generated queries (Qu et al. 2024; Shen et al. 2025). These taxonomies do not capture the lexeme-level mistakes that arise during the decoding process.

[*Our work looks at the lexeme level efficacy of constrained decoding and symbolic fixes.*] Our work addresses the gap of understanding *incremental errors* by analyzing lexeme-level errors made by LLMs for text-to-SQL. By categorizing incremental mistakes and identifying those amenable to symbolic repair, we provide insight into where constrained decoding can succeed and where simple symbolic fixes can aid LLM guided program synthesis.

3 Our Approach

Our overall system is shown in Figure 1.

3.1 Language Model Generator

[We generate lexemes token by token using a heap.] Consider $\mathcal{L}$ to be an (autoregressive) Language Model created for a sequence generation task. At each timestep $i \in [1, n]$, the model generates a sequence of these tokens, $(t_1, t_2, \ldots, t_n)$ based on previous tokens. At each generation step, the model predicts the top-k next tokens along with their probabilities. Candidate sequences are tracked using cumulative negative log-probabilities and stored in a min-heap, effectively approximating beam search. Lexeme generation continues iteratively until a space is produced, the heap exceeds a predefined maximum size, or a timeout is reached. This process allows $\mathcal{L}$ to generate SQL queries lexeme-by-lexeme while maintaining probabilistic ranking and supporting top-k exploration of candidates.

3.2 Language Model Validator

. *[We validate each lexeme by comparing it to the gold lexeme and continue generating.]* For each generated lexeme, we perform incremental validation by comparing it to the corresponding lexeme in the gold query. If the top-predicted lexeme does not match the gold lexeme, it is considered a mistake. To handle aliasing and schema prefixes, we compare only the column names, so that T1.BehaviorMonitoring and BehaviorMonitoring are treated as equivalent. When a mismatch occurs, we guide the language model by injecting the gold lexeme into the generated sequence, ensuring that subsequent predictions are conditioned on the correct context. This lexeme-level validation allows the model to follow more productive generation paths, rather than waiting to evaluate correctness only after the entire query is generated.

4 Experimental Setup

4.1 Implementation

Our system is implemented in Python and leverages open-source LLMs hosted on HuggingFace for text-to-SQL generation, including XiYanSQL-QwenCoder-7B-2504, deepseek-coder-6.7b-instruct, and XiYanSQL-QwenCoder-14B-2504. Models are loaded in 4-bit quantized mode when possible to reduce GPU memory usage, and generation is performed on A100 NVIDIA GPUs with 40GB VRAM for XiYanSQL-QwenCoder-7B-2504, deepseek-coder-6.7b-instruct and 80GB for XiYanSQL-QwenCoder-14B-2504 with automatic device mapping and offloading for larger models.

4.2 Model Choice

Finetuned models XiYanSQL-QwenCoder-7B-2504 and XiYanSQL-QwenCoder-14B-2504 were chosen due to their high performance on the Spider dataset (Liu et al. 2025). We also chose a general purpose model deepseek-coder-6.7b-instruct.

4.3 Dataset

From the test split of the Spider 1.0 dataset (Yu et al. 2018), we evaluate 369 queries that are categorized as hard and extra hard as we are most interested in errors that arise from complex queries.

4.4 Schema Selection

For each database, we extract the corresponding schema definition from the provided .sql file and include its contents verbatim in the model prompt. No additional preprocessing the of schema elements is performed. This allows us to isolate lexeme-level generation errors without introducing confounding effects from schema filtering mechanisms.

5 Taxonomy of Lexeme-Level SQL Errors

[Overview of taxonomy where we categorize incremental errors and identify easy, medium and hard fixes] We categorize incremental prediction errors into six primary types. We categorize errors that can be mitigated with simple symbolic fixes as **easy**. Errors where we can infer information that can be potentially useful for error correction are categorized as **medium**. Errors categorized as **Hard** are inherently more challenging due to dependencies, complex clause interaction, or the need for deep semantic understanding of the natural language query and database schema. Each category below includes a definition and the expected difficulty of applying symbolic fixes.

5.1 Schema-based Errors

Errors that occur when the model fails to correctly reference the database schema, such as selecting the wrong table or column, or hallucinating values.

Value Errors Errors where the model generates a lexeme that does not match the intended literal value.

Symbolic Fix: Moderate. These errors require align natural language references with database values using string similarity or enumeration over valid entries.

Table and Column Errors Errors involving incorrect table or column selection.

Symbolic Fix: Easy For many of these errors constrained decoding can help mitigate these errors by selecting the top predicted valid schema element from the top-k.

5.2 Structural / Clause Ordering Errors

Violations of expected SQL structure, such as missing or misordered clauses.

LIMIT Errors The model omits a `LIMIT` clause at the end of the query.

Symbolic Fix: Easy Post-process the query to append a LIMIT.

Other Structural Errors General clause misordering or missing clauses.

Symbolic Fix: Hard These errors require reasoning over query tree and matching the natural language question intent. These errors are hard because they often involve long-range dependencies and interactions between multiple clauses, which cannot be corrected by a simple fix.

5.3 Operator Errors

Errors in operator selection or placement.

Error Category	deepseek-coder-6.7b-instruct	XiYanSQL-7B	XiYanSQL-14B
Gold is highest valid schema element	187	396	423
Total Table/ Column Schema-based Errors	379	509	534

Table 1: This table presents the table and column based schema errors where the gold lexeme is the highest valid lexeme in the top-k.

Error Category	deepseek-coder-6.7b-instruct	XiYanSQL-7B	XiYanSQL-14B
Total Structural / Clause Ordering	710	323	254
Limit Errors	128	124	128
Other Structural Errors	582	199	126
Total Schema Grounding Errors	387	527	571
Value Errors	8	18	37
Table/Column Errors	379	509	534
Total Operator Errors	676	165	126
Where and Having Errors	290	140	102
Other Operator Errors	386	25	24
Aggregate Function Errors	105	103	94
Alias Errors	21	27	415
Other / Unclassified	195	190	96
Total Errors	2094	1335	1556

Table 2: Lexeme-level error distribution across models.

WHERE / HAVING Errors Missing or incorrect operators in WHERE or HAVING clauses at the end of a query.

Symbolic Fix: Medium/ Hard For HAVING or WHERE clauses at the end of a query, the correct operator may be infered from the schema and natural language context.

Other Operator Errors Other operator misuse.

Symbolic Fix: Hard Determining the correct operator often requires deeply understanding clause relations and subquery intricacies and understanding the full query context and schema, which may not be inferable from top-k candidates alone.

5.4 Aggregate Errors

Incorrect use of aggregation functions such as COUNT, SUM, or MAX.

Symbolic Fix: Medium/ Hard Some aggregations may be able to be infered from the natural language context. In general, these may be hard to infer

5.5 Alias Errors

Incorrect usage of `AS` clauses or missing alias definitions.

Symbolic Fix: Easy Can consult top-k candidates for valid alias names.

5.6 Other / Unclassified Errors

Syntactically valid but semantically incorrect or uncategorized errors.

Symbolic Fix: Hard These errors are difficult to detect and correct automatically, often involving multiple lexemes or hallucinated elements.

6 Results

We evaluate lexeme-level errors across three LLMs for text-to-SQL synthesis. Errors are categorized according to our proposed taxonomy. We also monitor the frequency with which the correct lexeme appears in the top-k candidates, indicating the potential effectiveness of constrained decoding.

Results for the gold lexeme occuring in the top-k candidates for table and column schema based errors are presented in 1. These results highlight the cases where the gold lexeme corresponds to the highest-ranked valid schema element. Across models, a large fraction (68% – 78%) of schema-based errors are "recoverable" via *schema based constrained decoding*, in which the highest ranked valid schema element is chosen.

Table 2 summarizes the lexeme-level error distribution across models and reveals distinct incremental failure patterns. Across models, schema grounding remains a large percentage of errors made. LIMIT errors persist across models as well comprising of between 18-50% of Structural based errors. A notable disparity emerges between XiYanSQL-7B and XiYanSQL-14B. While the 14B model reduces structural and operator errors relative to its 7B counterpart (254 vs. 323 structural errors; 126 vs. 165 operator errors), it exhibits a substantial increase in alias errors (415 vs. 27). This suggests that improvements in structural coherence do not uniformly translate to consistent symbolic reference generation. One possible explanation is that the larger model introduces greater variability in alias usage, which is surfaced under lexeme-level incremental validation. Because our evaluation compares each generated

Error Category	**deepseek-coder-6.7b-instruct**	**XiYanSQL-7B**	**XiYanSQL-14B**
Total Structural / Clause Ordering Errors	542/710	247/323	148/254
Limit Errors	17/128	69/124	47/128
Other Structural Errors	525/582	178/199	101/126
Total Schema Grounding Errors	330/ 387	463/527	473/571
Value Errors	8/8	6/18	16/37
Table/Column Errors	322/379	457/509	457/534
Total Operator Errors	46/676	80/165	67/126
Where and Having Errors	38/290	72/140	59/102
Other Operator Errors	8/386	8/25	8/24
Aggregate Function Errors	80/105	91/103	86/94
Alias Errors	9/21	27/27	411/415
Other / Unclassified	151/195	142/190	39/96
Total Errors	1158/2094	1050/1335	1224/1556

Table 3: This table presents the percentage of the gold lexeme found in the top-k by category.

lexeme against the canonical gold query, early divergences in alias selection may propagate and increase downstream mismatches. We emphasize that this analysis does not imply semantic incorrectness, but rather highlights representational divergence that becomes visible under incremental evaluation.

7 Conclusion

We present a lexeme-level analysis and taxonomy of **incremental errors** in LLM text-to-SQL synthesis. We categorize errors based on common pitfalls and symbolic repairability. Our analysis reveals that 68–78% of column and table schema grounding errors could be resolvable via schema-constrained decoding. This work establishes clear limits for constrained decoding: success requires correct lexemes in the model's top-k distribution. Easy fixes justify hybrid neuro-symbolic approaches, while Hard errors motivate multi-step refinement frameworks. Our taxonomy provides a structured framework for future text-to-SQL techniques, quantifying where symbolic methods complement neural generation.

References

Arcadinho, S. D.; Aparicio, D.; Veiga, H.; and Alegria, A. 2022. T5QL: Taming language models for SQL generation. In Bosselut, A.; Chandu, K.; Dhole, K.; Gangal, V.; Gehrmann, S.; Jernite, Y.; Novikova, J.; and Perez-Beltrachini, L., eds., *Proceedings of the Second Workshop on Natural Language Generation, Evaluation, and Metrics (GEM)*, 276–286. Abu Dhabi, United Arab Emirates (Hybrid): Association for Computational Linguistics.

Gao, D.; Wang, H.; Li, Y.; Sun, X.; Qian, Y.; Ding, B.; and Zhou, J. 2024. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.* 17(5):1132–1145.

Liu, Y.; Zhu, Y.; Gao, Y.; Luo, Z.; Li, X.; Shi, X.; Hong, Y.; Gao, J.; Li, Y.; Ding, B.; and Zhou, J. 2025. Xiyan-sql: A novel multi-generator framework for text-to-sql.

Mündler, N.; He, J.; Wang, H.; Sen, K.; Song, D.; and Vechev, M. 2025. Type-constrained code generation with language models. *Proc. ACM Program. Lang.* 9(PLDI).

Nagy, S.; Zhou, T.; Polikarpova, N.; and D'Antoni, L. 2026. Chopchop: A programmable framework for semantically constraining the output of language models. *Proc. ACM Program. Lang.* 10(POPL).

Qu, G.; Li, J.; Li, B.; Qin, B.; Huo, N.; Ma, C.; and Cheng, R. 2024. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-sql generation.

Scholak, T.; Schucher, N.; and Bahdanau, D. 2021. PICARD: Parsing incrementally for constrained autoregressive decoding from language models. In Moens, M.-F.; Huang, X.; Specia, L.; and Yih, S. W.-t., eds., *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 9895–9901. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics.

Shen, J.; Wan, C.; Qiao, R.; Zou, J.; Xu, H.; Shao, Y.; Zhang, Y.; Miao, W.; and Pu, G. 2025. A study of in-context-learning-based text-to-sql errors.

Takenouchi, K.; Ishio, T.; Okada, J.; and Sakata, Y. 2021. Patsql: efficient synthesis of sql queries from example tables with quick inference of projected columns. *Proc. VLDB Endow.* 14(11):1937–1949.

Wang, C.; Cheung, A.; and Bodik, R. 2017. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, 452–466. New York, NY, USA: Association for Computing Machinery.

Yu, T.; Zhang, R.; Yang, K.; Yasunaga, M.; Wang, D.; Li, Z.; Ma, J.; Li, I.; Yao, Q.; Roman, S.; Zhang, Z.; and Radev, D. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In Riloff, E.; Chiang, D.; Hockenmaier, J.; and Tsujii, J., eds., *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 3911–3921. Brussels, Belgium: Association for Computational Linguistics.