Current Limitations of LLM Guided Program Synthesis

First Author[1] , **Second Author**[2] , **Third Author**[2,3] , **Fourth Author**[4]
[1]First Affiliation
[2]Second Affiliation
[3]Third Affiliation
[4]Fourth Affiliation
{first, second}@example.com, third@other.example.com, fourth@example.com

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

[*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 errors can propagate throughout the decoding process which ultimately produces invalid programs.

[*Constrained decoding enforces constraints at decoding time, but they are evaluated on final outputs.*] Recent LLM augmented program synthesis techniques have focused on enforcing constraints at decoding time to ensure certain properties (Nagy et al. 2026). These techniques are often evaluated in terms of final outputs. However, constrained decoding operates at the lexeme level. To understand its effectiveness, we must 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.

[*We use SQL synthesis, and existing error taxonomies are based on fully generated queries.*] We conduct our lexeme-level analysis on a subspace of the program synthesis domain, the text-to-SQL task, as the correctness of an SQL query can be accurately assessed. Existing 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.

[*We evaluate lexeme level errors, and by doing so observe the potential benefit of symbolic repair and the limitations of constrained decoding.*] 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 correctable failures.
- We introduce a constrained decoding methodology, *schema constrained decoding* to address schema-based errors.

2 Related Work

Prior SQL synthesis using classic program synthesis approaches

Probablistic-symbolic methods

Current SOTA for SQL synthesis

Constrained Decoding

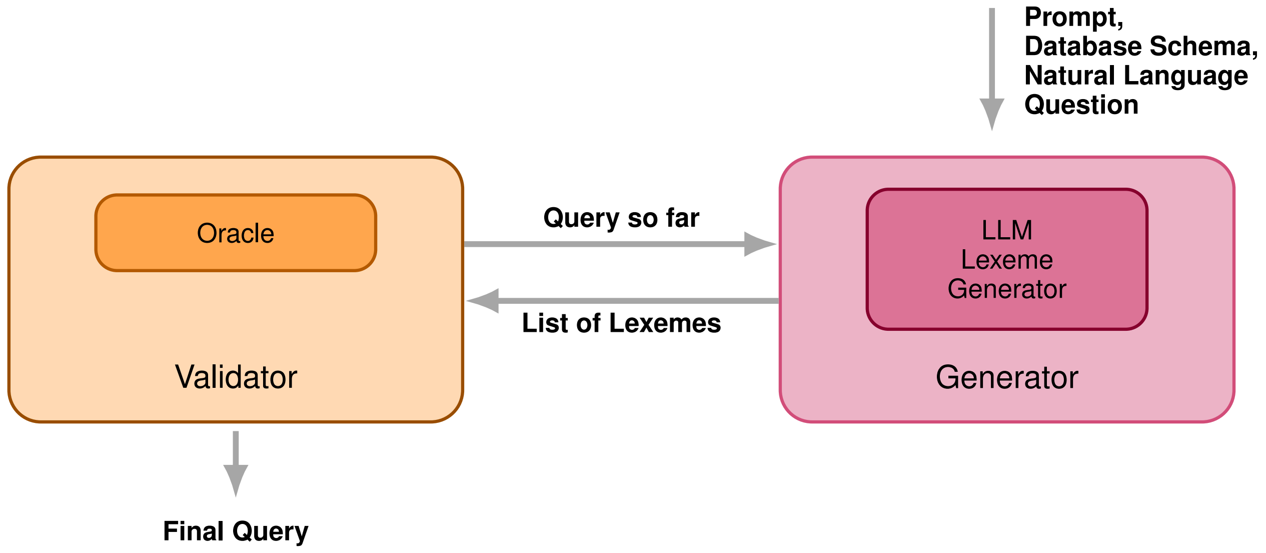

Figure 1: The overall system architecture. The Language Model Generator is prompted to generate a SQL query given a database schema and natural language question. The generator returns a list of lexemes to the Validator which incrementally validates the most probable lexeme against the gold one and corrects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

Prior research has explored SQL synthesis using classic approaches, such as enumerative search, to produce correct queries (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). They often include soundness and bounded completeness guarantees, but struggle to scale to larger, more complex queries.

Other research instead foregoes guarantees for empirical results. Many recent works focus on leveraging Large Language Models (LLMs) to produce SQL queries (Gao et al. 2024). They have performed well empirically, reaching test suite accuracies of up to 86.6% (Gao et al. 2024) against the Spider SQL dataset (Yu et al. 2018). FIXME MiniSeek 91.2 but no reference

Some prior works build off the strengths of both approaches, using probabilistic models to guide search while verifying the correctness of produced candidates. These prior works either require white-box access to probabilistic models (Lee et al. 2018; Menon et al. 2013), or treat models as program candidate oracles (Jha et al. 2023), or apply a hybrid approach that combines both methods (Li, Parsert, and Polgreen 2024). However, all these works use probabilistic models to generate full candidate programs before evaluating them. This leaves a large search space for the probabilistic model to categorize.

In contrast, we observe that LLMs are inherently next token predictors, rather than full sequence predictors. We leverage this insight to guide LLM search using lexeme level guidance which provides early guidance for the LLM to avoid unproductive search paths.

3 Our Approach

Our overall system is shown in Figure 1.

3.1 Language Model (LM) Generator

Consider $\mathcal{L}$ to be an (autoregressive) Language Model created for a sequence generation task that has a predefined vocabulary Σ, consisting of some k tokens which are used as part of the tokenization. At each timestep $i \in [1, n]$, the model generates a sequence of these tokens, $(t_1, t_2, \ldots, t_n)$ based on an autoregressive decoding process where the token with the highest probability is selected. This is usually part of a heuristic or sampling based search procedure on the set of k tokens to predict the next most probable token. Now, consider the LM to be finetuned on a new DSL and the tokens in the vocabulary are not changed. This means that the LM is now capable of producing the probability distribution over the same tokens but conforming to the changed ordering of our DSL *i.e.* if an original ordering of tokens $w \in \Sigma$ exists, then a new ordering of tokens $w^* \in \Sigma$ should also be produced by the LM. A $\mathcal{L}$ generates tokens and concatenates them by taking the token with the highest probability and generates a new token and continues until a whitespace token is generated or a max heap size is reached.

Example 3.1. *Consider the original token sequence w = SELECT, *, FROM, books, drawn from SQL queries present in the model's training data. Using our DSL, this sequence is transformed into w^* = PROJECT, *, FROM, books. In this transformation, the only substituted token is PROJECT, and we therefore expect the model's vocabulary to assign a non-zero probability p to this token.*

For commonly occurring words, we can fully expect the new word to also exist in the vocabulary, however, if the word does not exist we know that some n-gram combination of the word does exist. When for example the token, *REGENRATE $\notin \Sigma$ then we can form the same word by some combination like $\{$RE, GEN, E, RATE$\} \in \Sigma$. In our approach, we use QWEN-7B (Bai et al. 2023), which uses an open-source fast Byte Pair Encoding (BPE) for tokenization and len$(\Sigma) \in 151642$.*

Since we use the language model as an autoregressive next token predictor, we would also like to harness the power of the probabilistic nature in which the tokens are ranked. Usually, since the top-1 token is taken as the next prediction, there is an uncertainty that whether this next token is part of the valid query that satisfies the question that is requested by the user. Hence, we would also like to store, some top-m tokens at each time step and perform a beam search when on these token sets when there is a mismatch in the token requested and the query validated.

Language Model Validator. With each generated lexeme, we validate the generated lexeme by evaluating it against the expected lexeme of the gold query. If the most probable lexeme does not match the gold lexeme, we consider it to be a mistake. We compare the generated lexeme with the gold lexeme while considering issues that arise with aliasing for column names by only considering the column name. For example $T1.BehaviorMonitoring$ and $BehaviorMonitoring$ would both be considered correct lexemes compared to the gold lexeme $BehaviorMonitoring$. If the top most probable lexeme does not match the gold lexeme, we guide the Language Model by correcting it with the gold lexeme and continue generating. We incrementally validate each generated lexeme to guide the Language Model towards more productive generation paths rather than allow it to fully generate a query before evaluating its correctness.

Our system architecture is outlined in Algorithm 1.

Algorithm 1 System Algorithm

```
 1: Require: Language Model $\mathcal{L}$, beam width $m$, gold query $g$ and some string prompt $\mathcal{P}$, number of mistakes $n$
 2: procedure LANGUAGEMODELVALIDATOR($\mathcal{P}, w, g, n$)
 3:     if $g == w$ returns TRUE then
 4:         $\mathcal{P} \leftarrow \mathcal{P} \,||\, w$
 5:         return $\mathcal{P}, n$
 6:     else
 7:         $\mathcal{P} \leftarrow \mathcal{P} \,||\, g$
 8:         $n \leftarrow n + 1$
 9:         return $\mathcal{P}, n$
10:     end if
11: end procedure
12: procedure ORCHESTRATOR()
13:     $n \leftarrow 0$
14:     for $i = 1, 2, \ldots, \text{len}(\text{gold\_query})$ do
15:         $w_i \leftarrow \mathcal{L}(\mathcal{P})$
16:         $g_i \leftarrow$ GOLDLEXEME($g, i$)
17:         $\mathcal{P}, n \leftarrow$ LANGUAGEMODELVALIDATOR($\mathcal{P}, w_i, g_i, n$)
18:     end for
19:     return $n$
20: end procedure
```

4 Experimental Setup

4.1 Implementation

The system is implemented in Python using open source models FIXME and FIXME. We consider a lexeme to be complete if the next generated token is whitespace.

4.2 Results

Our empirical evaluation results are presented in Table FIXME.

5 Taxonomy

In this section, we outline our error taxonomy.

5.1 Schema-based Errors

Schema-based errors are defined as errors in which the model fails to correctly reference the database schema. These errors typically involve either selecting an incorrect table or column name, hallucinating schema attributes, or choosing an inappropriate value from the database for a condition.

Incorrect Naming These errors are defined as ones where the model generates a lexeme that does not correspond to the correct schema element. For example, the model might predict FIXME instead of the correct column FIXME.

Symbolic Fix: **Easy** To address the generation of hallucinated schema values, we propose a symbolic fix *schema-based constrained decoding* in which the first valid schema element that appears in the top-k is chosen. If no valid element is initially predicted, one can force the model to select from the set of valid schema elements, often yielding the correct token. Based on empirical results presented in FIXME, the highest valid schema element is FIXME percentage of the time the correct one. Such a technique would fix FIXME number of errors.

Incorrect Values These errors typically occur in `WHERE` conditions, where the model selects an incorrect literal value, even though the query structure and schema reference are correct. For instance, given the question:

"What are the addresses, towns, and county information for all customers who live in the United States?"

the gold query is:

```
SELECT address_line_1, town_city, county
FROM Customers
WHERE Country = 'USA';
```

A model might predict `'United'` as the value, which reflects a plausible misunderstanding of the question's phrasing. Correctly mapping natural language values to the database representation often requires explicitly grounding the question in the schema or the database contents.

Symbolic Fix: **Moderate** One approach is to align natural language mentions with schema values, using either string

	XiYanSQL-QwenCoder-7B-2504	deepseek-coder-6.7b-instruct	XiYanSQL-QwenCoder-14B-2504
Schema	1330	1107	1356
Structural	583	1385	472
Operator	305	1440	198
Aggregate	204	212	189
Alias	45	34	864
Other	272	259	142
Total	2739	4437	3221

similarity or enumeration over the database domain. This allows the model to select or rank candidates that correspond to actual entries in the database, mitigating errors in value selection.

Wrong Naming these are errors like wrong column or table name (the gold is present in the schema but the predicted candidate is not)

Symbolic fix: Easy- if valid schema element in top-k then you use a schema based constrained decoding to select (usually the gold is the top most). If there is no valid schema element, have the LLM pick from or rank valid schema elements

Wrong Values For where conditions - the wrong value is selected for example - ”What are the addresses, towns, and county information for all customers who live in the United States?” but the gold query is

```
SELECT $address_line_1$ ,  $town_city$ ,
county FROM Customers WHERE Country  =  'USA'
```

The LLM predicts **'United** which makes sense given the phrasing of the question but involves parsing out all the values of the schema to ”infer” that ”United States” maps to ”USA”

Symbolic Fix: Medium - We may be able to infer the correct condition but we need a way to map that question to the actual values in the database

5.2 Structural based errors

These are elements that are a violation of the expected SQL structure (ie clause based errors).

LIMIT errors The LLM doesn’t predict a LIMIT at the end of a full query.

Symbolic Fix: Easy - we basically just look at the output and if it matches except the number of rows we just add a LIMIT

Everything else

5.3 Where Conditions

Wrong operator wrong operator (sometimes the LLM will correctly predict bigger/ greater but not >

Symbolic fix: easy for wrong operator - if LLM predicts bigger then just replace it with >

upon further inspection there aren’t many of these that can be easily fixable (only about 2 were bigger/greater instead of >) but I’ll leave this category in for now

5.4 Aggregate Errors

Wrong aggregate functions like COUNT, MAX, etc

Symbolic fix: hard

5.5 Other

catch all

includes syntactically/ semantically ”similar” ie they look right....but aren’t

Symbolic fix: hard

6 Conclusion

LLMs are being increasingly used for program synthesis but solely relying on their probabilistic token ranking can lead to incorrect programs. In this work, we investigate these limitations and propose a lexeme level validation methodology to guide the LLM towards semantically valid queries. We find that many errors can be corrected by using a constraint decoding and beam-search guided reasoning technique to rerank the generated lexemes. For errors that cannot be corrected using these methods, purely symbolic techniques can be used to aid program synthesis.

References

Bai, J.; Bai, S.; Chu, Y.; Cui, Z.; Dang, K.; Deng, X.; Fan, Y.; Ge, W.; Han, Y.; Huang, F.; Hui, B.; Ji, L.; Li, M.; Lin, J.; Lin, R.; Liu, D.; Liu, G.; Lu, C.; Lu, K.; Ma, J.; Men, R.; Ren, X.; Ren, X.; Tan, C.; Tan, S.; Tu, J.; Wang, P.; Wang, S.; Wang, W.; Wu, S.; Xu, B.; Xu, J.; Yang, A.; Yang, H.; Yang, J.; Yang, S.; Yao, Y.; Yu, B.; Yuan, H.; Yuan, Z.; Zhang, J.; Zhang, X.; Zhang, Y.; Zhang, Z.; Zhou, C.; Zhou, J.; Zhou, X.; and Zhu, T. 2023. Qwen technical report.

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.

Jha, S. K.; Jha, S.; Lincoln, P.; Bastian, N. D.; Velasquez, A.; Ewetz, R.; and Neema, S. 2023. Counterexample guided inductive synthesis using large language models and satisfiability solving. In *MILCOM 2023 - 2023 IEEE Military Communications Conference (MILCOM)*, 944–949.

Lee, W.; Heo, K.; Alur, R.; and Naik, M. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, 436–449. New York, NY, USA: Association for Computing Machinery.

Li, Y.; Parsert, J.; and Polgreen, E. 2024. Guiding enumerative program synthesis with large language models. In Gurfinkel, A., and Ganesh, V., eds., *Computer Aided Verification*, 280–301. Cham: Springer Nature Switzerland.

Menon, A. K.; Tamuz, O.; Gulwani, S.; Lampson, B.; and Kalai, A. T. 2013. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, I–187–I–195. JMLR.org.

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.

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.