# Tree-guided Token Decoding for SQL Generation: Augmenting LLM Decoding for Jointly Reduced Latency and Hallucination

Anonymous ACL submission

## Abstract

Grammar-constrained decoding (GCD) empowers LLMs to generate highly structured output such as programming code (ex: SQL, Python, Go, etc.). Although GCD is effective in generating structurally valid code (Poesia et al., 2022; Ugare et al., 2024) and does not require fine-tuning, this line of work has two notable limitations: (a) these methods demand high inference times as compared to that of the unconstrained autoregressive decoding, making them unfit for real-time code generation applications (AI Assistants and Co-pilots); and (b) these methods are prone to schema hallucination errors like autoregressive decoding.

In this paper, we tackle the above research gap particularly in the context of a SQL generation task. By observing that the standard autoregressive LLM decoding methods are on par with GCD methods at generating structurally valid SQL code, we propose a novel unified approach − we refer to as *Tree-guided Token Decoding (TTD)* − which guides LLMs in decoding both SQL-keywords and database schema items (i.e. names of tables and columns) without focusing on the SQL grammar. Guiding schema items reduces hallucination errors, and such guiding often results in auto-filling certain tokens without explicit LLM calls, which also results in reduced inference times. We conducted extensive experiments using two popular datasets (Spider and BIRD) with three language models to demonstrate the efficacy of our proposed TTD approach using three metrics: execution accuracy, token rate, and number of executable SQLs generated.

## 1 Introduction

Large language models (LLMs) have been extensively used in several complex generation tasks with impressive human-level performance (Chang et al., 2024; Lin et al., 2021). Typically, LLMs decode auto-regressively one token at a time (Vaswani et al., 2017). In each decoding step, the LLMs first calculate a probability distribution on a predefined vocabulary, and then the decoding algorithm takes advantage of this probability distribution to sample the next generated token (Vaswani et al., 2017; Lin et al., 2021). Note that, in general, such LLM decoded output need not follow any rigid syntactic structure. However, for certain highly structured tasks − such as data serialization formats (ex: JSON, YAML, etc.), programming code (ex: SQL, Python, Go, etc.) and regular expressions − the LLM output is expected to satisfy certain syntactic constraints (Willard and Louf, 2023; Bailin Wang, 2023; Ugare et al., 2024).

The well-known techniques such as in-context learning and fine-tuning of LLMs have been shown to be inadequate (to meet the needs of the above sort of highly structured tasks) as their output fails miserably to satisfy the syntactic constraints (Beurer-Kellner et al., 2024; Honghua Zhang, 2023). The reason for this behavior is that these techniques provide certain examples (in the form of say <question, answer> pairs) which only *implicitly contains the syntax/grammar aspects of the underlying programming language*, without any explicit treatment of such structural requirements (Ugare et al., 2024).

Grammar-constrained decoding (GCD) is a technique proposed in the literature (Willard and Louf, 2023; Beurer-Kellner et al., 2024; Saibo Geng, 2024; Park et al., 2024; Zhuoer Wang, 2024) to mitigate the above challenges by restricting the output space conditioned on the previously generated tokens, given a set of globally valid structural constraints. The key advantages of the GCD approach are three-fold: (i) Exhibit satisfactory performance in generating syntactically valid code, (ii) Constrain the generation of keyword tokens of the underlying programming language; and (iii) This approach does not require any training or fine-tuning. However, this strand of work has the following two notable limitations:

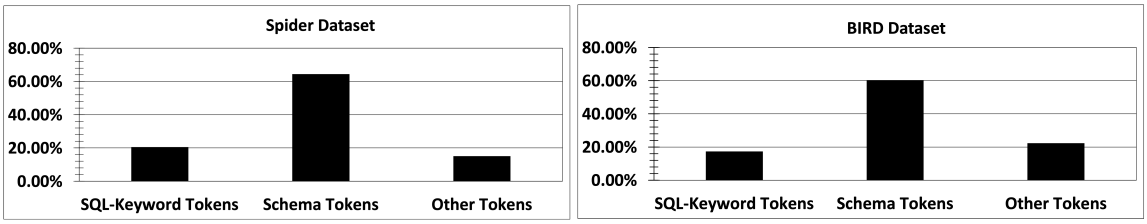


Figure 1: Using ground-truth SQLs from Spider and BIRD-Minidev datasets respectively, we compute distribution of three categories of tokens: SQL-keyword tokens, Database-Schema tokens, Other tokens (includes operators, special symbols, etc.)

**Limitation A**: The existing GCD methods require high inference time as compared to that of the auto-regressive unconstrained generation methods. For example, the work (Ugare et al., 2024) reports an average increase of 22% in inference time. The *high inference time overhead of these GCD methods negatively impact the user experience in real-time code generation applications* such as AI Assistants and Co-Pilots wherein the user's natural language question has to be translated into code on the fly.

**Limitation B**: Note that a syntactically valid SQL as generated by GCD techniques may not be executable (due to hallucination of database schema items, tables, and columns) and, even if the generated SQL is executable, it may result in wrong outcomes (due to semantic errors such as wrong usage of schema items in the SQL). That is, schema hallucination errors adversely impact the final accuracy of syntactically valid SQLs. For example, considering the SQL generation task using the Spider data set (Poesia et al., 2022), even though 76% of the generated SQLs are syntactically valid, the reported execution accuracy using the constrained GPT-3 175B model is only 37%. Similarly, SynCode (Ugare et al., 2024) exhibits very low functional correctness despite its very high syntactical accuracy – refer to Table 2 and Table 4 in (Ugare et al., 2024) for more details.

**Note:** Please refer to Appendix B for more details on the limitations of GCD-based approaches.

In this paper, we attempt to address the above limitations - latency overhead and hallucination of schema items - by developing a novel approach. especially for the SQL code generation task. In what follows, we explain our proposal to tackle the above two limitations in the context of developing our approach for SQL generation. We start with two key observations.

**Key observation** 1: Since GCD methods impose significantly high inference overhead in the process of generating structurally valid SQL code, let us shed some light on how satisfactory the pre-trained auto-regressive models are to generate valid SQL code. For example, using Spider and BIRD-Minidev datasets (refer to Section 4 for more details on these two datasets), Table 1 shows that the percentage of executable SQL code generated is on par with that of the GCD methods – refers to the percentage of valid SQLs using the base LLM models in Table 2 from (Poesia et al., 2022). In summary, *GCD methods with high inference time demand may not be required to generate executable SQL code, as auto-regressive language models are adequate for this task.*

| LLM Model | % of Executable SQLs | |
|---|---|---|
| | Spider | BIRD |
| SQLCoder-7B | 75% | 51% |
| Code-Llama-7B-Instruct | 93% | 59% |
| Granite-8b-code-instruct-4k | 89% | 53% |
| GPT-3.5-Turbo-175B | 96% | 82% |

Table 1: For SQL generation task using Spider and BIRD-Minidev public datasets, the percentage of executable SQLs generated by using unconstrained auto-regressive LLM models

**Key Observation** 2: Using ground-truth SQLs from the Spider and BIRD-Minidev datasets (refer to Section 4 for more details), respectively, we study the distribution of three categories of tokens: SQL-keyword tokens, database schema (tables/columns) tokens, and other tokens (such as operators, etc.). Figure 1 shows the distribution of these tokens. From this figure, *we observe that the fraction of database schema tokens is significantly higher (about 3 times) than that of SQL keyword tokens.* Since we know the database schema a priori like the SQL grammar, *it is technically possible to*

*restrict the decoding of schema tokens (in addition to SQL-keyword tokens) to reduce schema hallucination errors by LLMs and thereby aim to improve the accuracy of the SQL generation task.*

Inspired by the above two key observations, in this paper, we summarize our proposed approach as well as specific contributions:

**[1]** Since auto-regressive (or generically fine-tuned) LLM decoding methods are very efficient in generating executable SQL code (and thus paying no attention to SQL-grammar constraints), we propose to augment the auto-regressive LLM decoding process with a computationally light-weight and efficient mechanism (unlike GCD approaches) that restricts the token space wherever possible while decoding SQL-keywords, schema tables, and schema columns.

**[2]** To meet the proposed needs, in an offline manner, we construct THREE types of trees aligned with the LLM token space: one for SQL-keywords, one for database schema tables, and one more for database schema columns.

**[3]** During the inference time, we leverage these trees to restrict the token space for LLM decoding or to directly *autofill the next token (without LLM call) if the potential restricted token space contains only the* 1 *token*. Thus, we refer to our proposed approach as *Tree-guided Token Decoding (TTD)* for SQL generation. The auto-fill feature of our TTD approach is triggered more often as the names of schema tables and several columns are unique; thus having unique LLM token sequences, respectively, and thus the restricted token space often contains just the 1 token. This autofill feature of the TTD approach contributes significantly to the reduction of inference time (please refer to the results in Section 4.1 for empirical evidence). This autofill feature also helps to reduce schema hallucination errors, and Figure 2 highlights the same using an example from the Spider data set.

To the best of our knowledge, there is no known effort in the literature to tackle the two limitations of GCD approach and ours is first attempt indeed.

**[4]** The proposed TTD approach can be augmented with language models in the following two different settings. First, in the standard auto-regressive decoding setting (in particular, open language models such as Code-Llama, SQL-Coder, etc.). Second, in the speculative decoding setting (with both draft and target language models) (Yaniv Leviathan, 2023; Charlie Chen, 2023).

**Question:** How many models does each car maker produce? List maker full name, id and the number.

**SQL Generated using SQLCoder-7B Model:**

```
SELECT m.full_name, m.id, COUNT(ml.model_id)
AS model_count
FROM car_makers m JOIN model_list ml ON m.id = ml.maker
GROUP BY m.full_name, m.id
ORDER BY model_count DESC NULLS LAST;
```

**Schema Column Hallucination Error:** Here note that full_name is not a valid column.

**(Corrected) SQL Generated using TTD Model (SQLCoder-7B):**

```
SELECT m.fullname, m.id, COUNT(ml.modelid) AS
model_count
FROM car_makers m JOIN model_list ml ON m.id = ml.maker
GROUP BY m.fullname, m.id
ORDER BY model_count DESC NULLS LAST;
```

Figure 2: An illustrative example from Spider Dataset: Mitigating schema hallucination using our proposed TTD approach

Please refer to Appendix D for more results on this.

**[5]** We demonstrate the joint reduction of latency (through improved token generation rates) and hallucination (through improved accuracy by properly generating schema tokens) using thorough experiments on the SQL generation task using 3 metrics (execution accuracy, token rate, and number of executable SQLs) and 2 public datasets: Spider and BIRD-Minidev.

## 2 Relevant Work

Here we cover the relevant literature on the use of auto-regressive generative models for structured code generation tasks - wherein the output needs to satisfy certain programming language (ex: SQL, Cypher, Bash, etc.) constraints. Constrained decoding is a technique that has been proposed to guide the language model generation process to produce only valid outputs as needed by the syntax rules of underlying structured code task. In what follows, we divide the relevant literature into two broad categories.

*Category 1: General Constrained Decoding (Grammar):* There are largely two subcategories in this line of research. The first subcategory of the work deals with an in-context learning approach for grammar-constrained decoding. In particular, (Bailin Wang, 2023) deals with providing the language grammar as a few shot examples to conduct constrained decoding.

The second subcategory of the work deals with grammar constraints using sketches (suited for black-box LLMs) or finite-state machines (FSM).

In particular, (Saibo Geng, 2024) proposed a novel sketch-guided constrained decoding that is suitable for black-box LLM decoding. Again, (Luca Beurer-Kellner and Vechev, 2023) proposed an iterative constrained decoding approach for black-box LLMs. (Kexun Zhang, 2023; Zhuoer Wang, 2024) dealt with the constrained generation of API calls for tool usage using a FSM. Similarly, the following relevant literature (Beurer-Kellner et al., 2024; Willard and Louf, 2023; Park et al., 2024; Ugare et al., 2024) uses FSMs for grammar-constrained decoding for highly structured domains. We so far provided a few representative references on the relevant literature on grammar-constrained decoding, and it is by no means not an exhaustive coverage. By the way, this highlighted literature does not deal with SQL code generation task and we cover the same in the following.

*Category 2: Constrained Decoding for SQL Generation:* There is only a limited relevant literature available on this topic. Perhaps (Poesia et al., 2022) is the first paper that dealt with SQL generation and demonstrated results using Spider public data set. Like we already pointed out in the Introduction section. This paper (Poesia et al., 2022) reports 22% additional overhead on the inference time as compared to that of the standard auto-regressive setting.

Another interesting line of work (Torsten Scholak, 2021; Samuel Arcadinho, 2022) dealt with an incremental parsing-based approach (PICARD) for constrained beam-search for decoding SQL and this approach is compatible with any auto-regressive LLM. In particular, at each decoding step, this approach operates on the output of the language model to determine valid output sequences by rejecting inadmissible tokens from the beam at the earliest possible time. Clearly, this work does not deal with error mitigation during inference time as in our approach in this paper. Furthermore, since PICARD (Torsten Scholak, 2021; Samuel Arcadinho, 2022) performs a constrained beam search and post-decode error correction, its overall SQL inference time would be extremely high and the authors (Torsten Scholak, 2021; Samuel Arcadinho, 2022) did not report the running times.

**Note:** Please refer to Appendix C for more references on Text-to-SQL problem.

## 3 Our Proposed Approach

Let $D$ be the set of vocabulary corresponding to SQL programming language and we define it to be the collection of SQL-keywords, names of database schema tables and columns, defined as follows:

$$D = \{a_1, a_2, \ldots a_m\} \tag{1}$$

where, $a_i$ refer to the $i^{th}$ element of $D$. Let $V$ be vocabulary/tokens of the underlying language model (LLM). Let $T$ be the set of tokens that correspond to the elements of $D$. That is,

$$T = \{(i, j, k) \mid a_i[j : j + k - 1] \in V\} \tag{2}$$

where $i$ refers to the $i^{th}$ element of $D$ (i.e., $a_i$), $j$ refers to the starting index of substring of $a_i$, $k$ refers to the length of that substring that corresponds to a token in $V$. That is, $T \subseteq V$. Then, $t \in T$ can be defined as $t = (i, j, k)$. Note that each token $t \in T$ also corresponds to a token id of the underlying tokenizer of the LLM. Let the *End of Sequence* token be denoted as $t_{EOS}$. Also, let us define a function $Next(.) : T \rightarrow T \cup \{t_{EOS}\}$ for each $t = (i, j, k)$ as:

$$Next(t) = \begin{cases} b = (i, j + k, k'), & b \in T \\ t_{EOS}, & Otherwise \end{cases} \tag{3}$$

Using this, we define a tree with tokens being nodes and the set of edges – call it $E_{tokens}$ – is defined as:

$$E_{tokens} = \{(t, Next(t)) \mid t \in T\} \tag{4}$$

Note that $E_{tokens}$ is defined in the LLM token space. Now, we define the set of outbound tokens (or sub-strings) given a token (or sub-string) $t \in T$ using a function $P : T \rightarrow T$ as follows:

$$P(t) = \{Next(t) \mid (t, Next(t)) \in E_{tokens}\}. \tag{5}$$

We make use of this formulation to restrict the decoding space (i.e. possible set of tokens) of the underlying LLM as follows. Let us define another function $g : T \rightarrow V$ for each $t = (i, j, k)$ as:

$$g(t) = \begin{cases} P(t), & if \;\; \exists\, (t, Next(t)) \in E_{tokens} \\ & for\ some\ Next(t) \in T \\ V, & Otherwise \end{cases} \tag{6}$$

Note that $g(.)$ performs token restriction wherever possible using $E_{tokens}$ (i.e. tree).

Using the above framework, we are in a position to define our proposed *tree-guided token decoding (TTD)* approach. Towards this end, let $S = \{t_1, t_2, \ldots, t_i\}$ be the sequence of already decoded tokens with $S[-1] = t_i$ being the latest decoded token. Given that $S$ is already decoded token sequence, let $f_{LLM}(S, V)$ be any arbitrary LLM decoding function that auto-regressively decodes a new token from the token space $V$. Also, define that $V^* = g(S[-1])$ denotes (possibly restricted) token space given the latest decoded token is $S[-1]$.

Now, we define another function, $h(.)$ that performs tree-guided token decoding for SQL generation as follows:

$$h(S, V^*) = \begin{cases} t, & if\ V^* = \{t\} \\ f_{LLM}(S, V^*), & Otherwise. \end{cases} \quad (7)$$

Note that, if $V^* = \{t\}$, then we directly auto-fill the next token to be decoded with $h(S, V^*) = t$. This is a powerful feature of our proposed TTD approach that contributes to reduce the inference time. Algorithm 1 formally defines the key steps involved in our proposed TTD approach. In this algorithm, we need to define one more function, $\psi(.)$, which takes the current state of KV cache of the LLM, and updates it with the calculations made to decode the new token.

**Algorithm 1** TTD: Tree-guided Token Decoding for SQL Generation

**Require:** $S = \{t_1, t_2, \ldots, t_i\}$: Initial sequence of already decoded tokens; $K$: Initial KV-cache; $V$: Vocabulary/tokens of the underlying LLM; $N$: Maximum length for the tokens to be generated.

$t^* = \phi$
**while** $t^* \neq t_{EOS}$ or len$(S) <$ N **do**
  $V^* \leftarrow g(S[-1])$
  $t^* \leftarrow h(S, V^*)$
  $S \leftarrow S \cup t^*$
  $K \leftarrow \psi(t^*, K)$
**end while**

Note that the existence of the tree and its corresponding token edges $E_{tokens}$ play a critical role to efficiently infer the subsequent tokens as shown in Algorithm 1. Hence, it is important to understand how we construct the tree as well as the set $E_{tokens}$. In fact, we construct this tree in offline fashion as described below.

### 3.1 Offline Construction of Trees

We construct 3 type of trees offline for decoding during inference: (1) one for SQL-keywords (2) one for database schema - tables (3) and the other for database schema - columns. Below, we describe our offline construction approach for these trees.

Towards this end, we have to establish a mapping from each of SQL-keywords list, schema tables list, and schema columns list respectively to the list of LLM tokens (to tackle the token-misalignment problem (Beurer-Kellner et al., 2024; Ugare et al., 2024; Poesia et al., 2022)). For each of the above three lists (i.e., SQL-keywords, tables, columns) respectively, we carry out the following processing: (a) We prepare a comprehensive list to account for various allowed versions of the individual elements (such as lower and upper cases); (b) We next tokenize each element in this list resulting in a list of substrings aligned with LLM tokens – for instance using Spider dataset, *'singer_in_concert'* is a table name (i.e. element) in the list of schema tables and tokenizing this element results in the following list of substrings aligned with the SQLCoder tokens: ['singer','_','in','con','cert']. Repeating this process for each SQL based element (keywords, tables, columns) results in a collection of list of substrings aligned with LLM-tokens; (c) We then construct a tree leveraging the substrings (or LLM tokens) from this list as vertices and each edge in the tree denotes the sequence in which they appear in the list so that when we traverse the resulting tree, we get the tokenized version of the entity under consideration. For example, Figure 3 refers to one such tree of substrings (aligned with SQLCoder tokens) using a list of schema tables (which contains 3 tables: 'stadium', 'singer_in_concert', 'concert') of one database from Spider dataset.

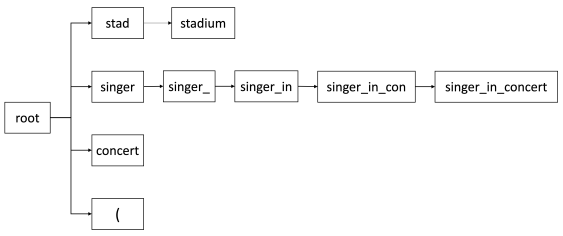


Figure 3: For a database schema from Spider dataset that contains 3 tables ('stadium', 'singer_in_concert', 'concert'), we construct a corresponding tree aligned with LLM tokens space

*The above constructed trees are extremely useful during inference by LLMs (via $E_{tokens}$ set) in the*

*sense that, starting from a given node (token), if there is only a single child in the tree, then we can directly auto-fill that token, to invoking the LLM decoding step (thus reducing inference time!).* As these trees are constructed by keeping the order of the tokens present in the tokenized version of the entity under consideration, auto-filling in this fashion would not result in any incorrect (hallucinated) tokens being decoded. In case there is more than one child in the tree corresponding to the already decoded token, then LLM decoding is triggered by restricting the possible token space to the set of all children of that decoded token in the tree. In this manner, we ensure that the LLMs always decode the names of SQL-keywords, schema tables, and schema columns without any hallucination (refer to Section 4 for empirical evidence on the hallucination error reduction).

Thus, the above trees built from the tokens of the SQL keywords, schema tables, schema columns primarily serve the dual purpose of speeding up inference wherever possible (via auto-fill which takes orderwise lesser time than an LLM invocation) and ensuring token restriction resulting in reduced LLM hallucination.

### 3.2 Invoking Trees during LLM Decoding

To make appropriate use of the proposed token trees during LLM decoding process, it's important to ensure that our TTD (*Tree-guided Token Decoding*) approach by restricting the decoding space of the LLM is invoked only at the relevant places (as our approach do not constrain SQL grammar). Accordingly, we come up with a simple set of carefully crafted rules, which when applied ensures that invoking our TTD approach results in improvement in the quality of SQL being generated. We devise the following rules corresponding to each type of the tree:

*Tree - SQL-keywords:* The main objective of the keyword tree is to provide us speedup in token decoding rate. This tree is invoked via TTD only when the tokens corresponding to the current word being decoded by the model are present in the tree.

*Tree - Schema Tables:* The main objective of this tree is to ensure that LLM decodes the schema table names accurately (without any hallucination). Note that pre-trained LLMs are not aware of the schema tables as they are user-defined or proprietary, unless they are fine-tuned over the schema under consideration. We augment this tree when the last decoded words are "FROM" or "JOIN" - SQL structure implies that the subsequent set of tokens must correspond to a table name. However, they can also be followed by parenthesis, "(", to accommodate for nested queries, and the alias used for them (ex: refer to Figure 3).

*Tree - Schema Columns:* Similar to the tree for schema tables as above, this tree also has the main objective of reducing the hallucination errors corresponding to schema columns during the SQL generation task. We invoke this tree via TTD whenever the last decoded literal is ".", and the second last literal is not a numeral - in these case, the next set of tokens must correspond to a column.

| Dataset | Type | #Test Questions | #DBs |
|---|---|---|---|
| Spider | Public | 992 | 19 |
| BIRD (MiniDev) | Public | 500 | 11 |

Table 2: Description of two public datasets used in our Experiments. Here *#DBs* refers to *Number of Databases*

## 4 Experiments

In this section, we first describe the datasets, experimental setup and then our experimental results in detail.

**Description of Datasets:** We work with 2 public datasets for our experiments.

*Spider - Public Dataset:* It is a popular cross-domain database for Text-to-SQL parsing (Tao Yu, 2018). There is no overlap between questions or databases among the respective training, development and test sets. This dataset contains 200 databases that cover 138 domains, such as colleges, government, etc. There are 10181 questions with 5693 unique SQL queries. Among them, Spider randomly selects 7000 annotated instances as the training set and rest form the development set. From the development set, we used 19 databases, totaling 992 questions.

*BIRD (Minidev) - Public Dataset*: This (development) dataset aims to streamline development cycles, specifically for testing and improving SQL query generation models in a cost-effective manner. Compiled from community feedback, it consists of 500 high-quality text-to-SQL pairs sourced from 11 different databases from the Bird development environment (Li et al., 2024).

| Metric | Spider | | | BIRD MiniDev | | |
|---|---|---|---|---|---|---|
| | *AR* | *SynCode* | *TTD* | *AR* | *SynCode* | *TTD* |
| **Execution Accuracy (EX)** | 50.9% | **52.2%** (↑2.55%) | 51.7% (↑1.57%) | 21.2% | 21.2% | **22%** (↑3.77%) |
| **Token Rate (TR)** | 15.74 | 14.77 (↓6.16%) | **17.33** (↑10.1%) | 15.17 | 13.64 (↓10.09%) | **15.59** (↑2.77%) |
| **#Executable SQLs** | 744 | 754 (+10) | **759** (+15) | 255 | 255 | **288** (+ 33) |

Table 3: Results for the SQLCoder-7B model. All the increments/decrements shown are with respect to the AR.

**LLM Models Used in Our Experiments**: We use 4 LLM models in our experiments: (i) *SQL-Ccoder* [1] 7B model, (ii) *CodeLlama* [2] 7B-instruct model, and (iii) *Granite-8b-code-instruct-4k* [3]

**Experimental Settings:** We conduct our experiments using the following 3 experimental configurations: (i) Standard autoregressive decoding (which we refer to as *AR*); (ii) Syncode (GCD); (iii) Our proposed approach (*TTD*). Further, given a Natural Language question, we use the setup provided in SQLCoder documentation (prompt as well as post-processing module) to obtain the SQL query. However, for Codellama and Granite, as they are not specifically fine-tuned for the NL2SQL task, they are prone to generate text other than an SQL query. To limit the generation only to SQL, we used "\n\n" and "[/" (based on the SQLCoder prompt) as additional stopping criteria.

**Evaluation Approach:** We used the evaluation framework presented in (Zhong et al., 2020) [4] for evaluating results over the Spider dataset, and the setup provided at the official Github repository of BIRD (Minidev)[5], to evaluate generated SQL queries from the variety of approaches/models.

**Three Evaluation Metrics:** We use the following three metrics to benchmark the efficacy of our approach compared to the baseline methods:

*Execution accuracy (EX)*: We use this metric (Tao Yu, 2018) to measure the overall correctness of the generated SQL. Following this metric, we check whether or not the database results extracted using the generated SQL are equivalent to the database results extracted using the ground-truth SQL.

*Token Rate (TR):* We use this metric to measure the latency (or inference time) of any given LLM model for the SQL generation task. Following this metric, we calculate the average number of LLM tokens generated per second. Note that for a higher token rate, the inference time of the underlying LLM model becomes shorter (and the better it is).

*Number of Executable SQLs Generated:* Since a structurally valid SQL may not be executable (due to hallucination of schema items), we focus on the *number of executable SQLs generated* by the LLM as a metric. Note that every executable SQL is a structurally valid SQL, but not vice versa.

### 4.1 Our Results [TTD Approach]

Here we present our results by considering the three configurations mentioned above for each of the three language models we used in our experiments for both Spider and Bird Minidev data sets.

Table 3 presents evaluation metrics for the SQL-Coder 7b model. For the Bird minidev dataset, *TTD* outperforms other approaches in all metrics. For the Spider dataset, *TTD* surpasses *AR* and *Syncode* in token rate and executable SQLs, while *Syncode* achieves marginally higher execution accuracy than *TTD*.

Table 4 presents evaluation metrics for the Code-Llama-7B-Instruct model. For the Bird minidev dataset, *TTD* leads in all metrics. In the Spider dataset, *TTD* outperforms *AR* and *Syncode* in token rate and number of executable SQLs, while *AR* is best for execution accuracy. Note that *TTD* performs almost similar to *AR* and achieves about 3% better accuracy than *Syncode*. For results using Granite model, please refer to Appendix A.

As observed, the *TTD* approach achieves higher token rates on average. This improvement is due to reduced inference times in SQL generation, as *TTD* performs token auto-filling when the restricted token space is just 1, eliminating the need for an

[1] https://github.com/defog-ai/sqlcoder/tree/main

[2] https://huggingface.co/collections/meta-llama/code-llama-family-661da32d0a9d678b6f55b933

[3] https://huggingface.co/ibm-granite/granite-8b-code-instruct-4k

[4] https://github.com/taoyds/spider

[5] https://github.com/bird-bench/mini_dev

| Metric | Spider | | | BIRD MiniDev | | |
|---|---|---|---|---|---|---|
| | *AR* | *SynCode* | *TTD* | *AR* | *SynCode* | *TTD* |
| **Execution Accuracy (EX)** | **60.4%** | 57.7% (↓4.47%) | 60.1% (↓0.5%) | 24.6% | 24.6% | **25%** (↑1.63%) |
| **Token Rate (TR)** | 16.47 | 14.19 (↓13.84%) | **17.12** (↑3.95%) | 15.44 | 13.68 (↓11.4%) | **15.47** |
| **#Executable SQLs** | 924 | 906 (-18) | **929** (+5) | 295 | 292 (-3) | **330** (+ 35) |

Table 4: Results for Code-Llama-7B-Instruct. All the increments/decrements shown are with respect to the AR.

explicit LLM decoding call. Tables 5 and 6 show that a significant fraction of tokens are auto-filled using *TTD*: on average (across Spider and BIRD minidev datasets), 15.52% of tokens are auto-filled with SQLCoder, 11.7% with Code-Llama, and 7% with Granite.

| Model | Fraction of Auto-filled Tokens by *TTD* |
|---|---|
| SQLCoder-7B | $\frac{7798}{43758} = 17.82\%$ |
| Code-Llama-7B | $\frac{7312}{51564} = 14.18\%$ |
| granite-8b-code | $\frac{3358}{33980} = 9.88\%$ |

Table 5: The fraction of auto-filled tokens (without explicit LLM decoding) by our proposed *TTD* approach using different LLM models on the Spider dataset

| Model | Fraction of Auto-filled Tokens by *TTD* |
|---|---|
| SQLCoder-7B | $\frac{4742}{35890} = 13.21\%$ |
| Code-Llama-7B | $\frac{3916}{42531} = 9.21\%$ |
| granite-8b-code | $\frac{1474}{29411} = 5.01\%$ |

Table 6: The fraction of auto-filled tokens (without explicit LLM decoding) by our proposed *TTD* approach using different LLM models on the Bird minidev dataset

### 4.2 Comparison with GCD (Syncode) Baseline

We compare the performance of our *TTD* approach with one *GCD* baseline – *Syncode* (Ugare et al., 2024) – using three metrics (execution accuracy, token rate, and number of executable SQLs) and two datasets (Spider and BIRD MiniDev). From the results in Table 3, Table 4 and Table 7, *TTD* outperforms *Syncode* in 17 of 18 comparisons (i.e., 6 comparisons coming from each table). In particular, we make the following two key empirical observations:

- *TTD* generates more executable SQLs than *Syncode*. This is because *Syncode* only restricts SQL grammar tokens, not schema items, leading to hallucinations in schema items that may prevent SQL execution. In contrast, *TTD* guides both SQL keywords and schema item generation.

- The token rate of *TTD* is higher than that of *Syncode*. This is mainly due to the auto-fill feature of our proposed *TTD* approach.

**Comparison between Syncode and AR:** *Syncode* has a lower token rate than AR in all scenarios in these tables, indicating higher inference times for *GCD*. Regarding executable SQLs, *Syncode* generates more executable SQLs than *AR* in 2 instances, but in 3 instances, it generates fewer than *AR*.

## 5 Conclusions and Future Work

Although constrained decoding generates syntactically valid code, it requires higher inference times than autoregressive models and is prone to schema hallucinations, as shown. To address these issues, we propose the tree-guided token decoding (*TTD*) approach, which avoids SQL grammar constraints and only guides the LLM to generate SQL keywords, table names, and column names. We empirically demonstrated the effectiveness of *TTD* compared to the autoregressive approach and GCD approaches (Syncode) using execution accuracy and token rate metrics.

An interesting future research direction would be to extend our approach to other low-code generation tasks such as Cypher code or Bash code.

## 6 Limitations

Since we do not leverage the SQL grammar explicitly due to its high inference times (which does not

suit to real-time code generation settings), we use a rule-based approach to decide when to trigger our proposed tree-guided token decoding approach.

Also note that we do not leverage the tree guidance in every LLM token decoding. Below are a few such instances:

(a) While our approach guides the LLMs in generating the names of database schema items (tables and columns), it does not verify the membership of columns to the tables. This would further improve the accuracy of our proposed approach.

(b) If a table alias is used in both the outer and inner subqueries, such table aliases are not leveraged by our approach while generating the respective column names.

## References


Ahmed Hassan Awadallah Ahmed Elgohary, Saghar Hosseini. 2020. Speak to your parser: Interactive text-to-sql with natural language feedback. In *Proceedings of ACL*, pages 2065–2077.

Lijie Wen Aiwei Liu, Xuming Hu and Philip S. Yu. 2023. A comprehensive evaluation of chatgpt's zero-shot text-to-sql capability. In *CoRR abs/2303.13547*.

Alexandros Zervakis Alkis Simitsis Georgia Koutrika Yannis E. Ioannidis Andreas Kokkalis, Panagiotis Vagenas. 2012. Logos: a system for translating queries into narratives. In *Proceedings of SIGMOD Conference*, pages 673–676.

Xuezhi Wang Yuan Cao Rif A. Saurous Yoon Kim Bailin Wang, Zi Wang. 2023. Grammar prompting for domain-specific language generation with large language models. In *NeurIPS*.

L. Beurer-Kellner, M. Fischer, and M.T. Vechev. 2024. Guiding llms the right way: Fast, non-invasive constrained generation. In *Proceedings of 41st International Conference on Machine Learning (ICML)*.

Ursin Brunner and Kurt Stockinger. 2021. Valuenet: A natural language-tosql system that learns from database information. In *Proceedings of 37th IEEE International Conference on Data Engineering (ICDE)*, page 2177–2182.

Shuaichen Chang and Eric Fosler-Lussier. 2023. How to prompt llms for textto-sql: A study in zero-shot, single-domain, and cross-domain settings. In *CoRR abs/2305.11853*.

Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P.S. Yu, Q. Yang, and X. Xie. 2024. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol (ACM TIST)*, 15(3):39:1–39:45.

Geoffrey Irving Jean-Baptiste Lespiau Laurent Sifre John Jumper Charlie Chen, Sebastian Borgeaud. 2023. Accelerating large language model decoding with speculative sampling. In *CoRR abs/2302.01318*.

Michael J. Cafarella Christopher Baik, Zhongjun Jin and H. V. Jagadish. 2020. Duoquest: A dual-specification system for expressive sql queries. In *Proceedings of the 2020 International Conference on Management of Data*, page 2319–2329.

Yaliang Li Xiuyu Sun Yichen Qian Bolin Ding Dawei Gao, Haibin Wang and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. In *CoRR abs/2308.15363*.

N. Deng, Y Chen, and Y. Zhang. 2022. Recent advances in text-to-sql: Asurvey of what we have and what weexpect. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 2166–2187.

Nanyun Peng Guy Van den Broeck Honghua Zhang, Meihua Dang. 2023. Tractable control for autoregressive language generation. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 40932–40945.

Gautier Izacard Xavier Martinet Marie-Anne Lachaux Timothée Lacroix Baptiste Rozière Naman Goyal Eric Hambro Faisal Azhar Aurélien Rodriguez Armand Joulin Edouard Grave Hugo Touvron, Thibaut Lavril and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. In *CoRR abs/2302.13971*.

Kenton Lee Jacob Devlin, Ming-Wei Chang and Kristina Toutanova. 2019. Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, page 4171–4186.

Abdul Quamar Fatma Özcan Vasilis Efthymiou Ayushi Dalmia Greg Stager Ashish R. Mittal Diptikalyan Saha Jaydeep Sen, Chuan Lei and Karthik Sankaranarayanan. 2020. Athena++: Natural language querying for complex nested sql queries. In *Proceedings of VLDB Endowment 13, 11*, page 2747–2759.

Lei Li William Yang Wang Kexun Zhang, Hongqiao Chen. 2023. Syntax error-free and generalizable tool use for llms via finite-state decoding. In *CoRR abs/2310.07075*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 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.

T. Lin, Y. Wang, X. Liu, and X. Qiu. 2021. A survey of transformers. *https://arxiv.org/abs/2106.04554*.

Weijin Zou Narutatsu Ri Jaesung Tae Ellen Zhang Arman Cohan Linyong Nan, Yilun Zhao and Dragomir Radev. 2023. Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies. In *CoRR abs/2305.12586*.

Marc Fischer Luca Beurer-Kellner and Martin Vechev. 2023. Prompting is programming: A query language for large language models. In *Proceedings of the ACM on Programming Languages, (PLDI)*, pages 1946–1969.

Ngoc Phuoc An Vo Hangu Yeo Elahe Khorashani Octavian Popescu, Irene Manotas and Vadim Sheinin. 2022. Addressing limitations of encoder-decoder based approach to text-to-sql. In *Proceedings of the 29th International Conference on Computational Linguistics*, page 1593–1603.

Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-aligned decoding. In *CoRR abs/2405.21047*.

Wen-tau Yih Pengcheng Yin, Graham Neubig and Sebastian Riedel. 2020. Tabert: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, page 8413–8426.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *In Proceedings of 10th International Conference on Learning Representations (ICLR)*.

Damai Dai Ce Zheng Zhiyong Wu Baobao Chang Xu Sun Jingjing Xu Lei Li Qingxiu Dong, Lei Li and Zhifang Sui. 2023. A survey for in-context learning. In *CoRR abs/2301.00234*.

Zhenjie Zhang Xiaoyan Yang Zijian Li Ruichu Cai, Boyan Xu and Zhihao Liang. 2018. An encoder-decoder framework translating natural language to database queries. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, page 3977–3983.

Hootan Nakhost Hanjun Dai Rajarishi Sinha Pengcheng Yin Ruoxi Sun, Sercan Ö. Arik and Tomas Pfister. 2023. Sql-palm: Improved large language model adaptation for text-to-sql. In *CoRR abs/2306.00739*.

Chris Wendler Martin Josifoski Robert West Saibo Geng, Berkay Döner. 2024. Sketch-guided constrained decoding for boosting blackbox large language models without logit access. In *Proceedings of ACL (Short Papers)*, pages 234–245.

Hugo Veiga António Alegria Samuel Arcadinho, David Aparício. 2022. T5ql: Taming language models for sql generation. In *CoRR abs/2209.10254*.

Alkis Simitsis and Yannis E. Ioannidis. 2009. Dbmss should talk back too. In *Proceedings of CIDR*.

Kai Yang Michihiro Yasunaga Dongxu Wang Zifan Li James Ma Irene Li Qingning Yao Shanelle Roman Zilin Zhang Dragomir R. Radev Tao Yu, Rui Zhang. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of EMNLP*, pages 3911–3921.

Dzmitry Bahdanau Torsten Scholak, Nathan Schucher. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9895–9901.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Improving llm code generation with grammar augmentation. In *CoRR abs/2403.01632*.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, and I. Polosukhin. 2017. Attention is all you need. In *Proceedings of NIPS*, pages 5998–6008.

Brandon T. Willard and Rémi Louf. 2023. Efficient guided generation for large language models. In *CoRR abs/2307.09702*.

Yossi Matias Yaniv Leviathan, Matan Kalman. 2023. Fast inference from transformers via speculative decoding. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 19274–19286.

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic evaluation for text-to-sql with distilled test suites. *arXiv preprint arXiv:2010.02840*.

Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. 2024. Large language model enhanced text-to-sql generation: A survey. In *arXiv:2410.06011v1*.

Alexandros Papangelis Rohan Mukherjee Tzu-Yen Wang Xinyan Zhao Arijit Biswas James Caverlee Angeliki Metallinou Zhuoer Wang, Leonardo F. R. Ribeiro. 2024. Fantastic sequences and where to find them: Faithful and efficient api call generation through state-tracked constrained decoding and reranking. In *CoRR abs/2407.13945*.

## A Granite-8b-code-instruct-4k: Results

Table 7 shows the evaluation metrics for the Granite-8b-code-instruct-4k model. In this case, the same trend is observed for both data sets, that is, *TTD* outperforms the other approaches in terms of execution accuracy and number of executable SQLs, and *AR* is the best in terms of token rate. However, even in this scenario, *TTD* is better than *Syncode*.

| Metric | Spider | | | BIRD MiniDev | | |
|---|---|---|---|---|---|---|
| | *AR* | *SynCode* | *TTD* | *AR* | *SynCode* | *TTD* |
| **Execution Accuracy (EX)** | 68.2% | 67.7% (↓4.47%) | **68.6%** | 22.6% | 22.6% | **23.2%** (↑2.65%) |
| **Token Rate (TR)** | **13.97** | 13.24 (↓5.23%) | 13.43 (↓3.86%) | **13.12** | 12.27 (↓6.48%) | 12.62 (↓3.81%) |
| **#Executable SQLs** | 885 | 891 (-18) | **895** (+5) | 267 | 268 (+1) | **271** (+4) |

Table 7: Results for Granite-8b-code-instruct-4k. All the increments/decrements shown are with respect to the AR.

# B Clarification of GCD Limitations and Efficiency of TTD Approach

In this section, we provide a more detailed discussion on the limitations of grammar-constrained decoding approaches.

## B.1 Limitation A - Inference Time

The existing GCD methods require a long inference time compared to non-constrained auto-regressive generation methods. The primary reason for these high inference times of the GCD methods is the processing of finite-state machines to identify the feasible LLM tokens from which the next token needs to be generated.

Since token trees in our proposed approach are constructed offline, the time taken to build these trees is not included in the reported latency improvements. This is analogous to how mask generation in GCD (Syncode) is excluded from the calculation of token rates. However, it is worth noting that while mask store generation in GCD typically takes several minutes, the tree generation in our case is completed in a matter of seconds.

Following our TTD approach, three types of token trees are constructed: SQL Keyword Trees, Table Trees, and Column Trees. For SQL keywords and tables, a single tree is constructed for each. In contrast, multiple trees are created for columns. Specifically, a tree is built that includes all columns across all tables, and additional individual column trees are constructed for each table based solely on the columns present within that specific table.

The tree containing all columns is used when aliases are involved in the query to connect column names, but the alias has not yet been defined. On the other hand, the individual column trees are instrumental in further restricting the decoding space of the large language model (LLM) when complete table names are used to link column names, or when an alias has already been defined. This ensures that the LLM only selects columns that belong to the corresponding table, thereby improving the latency and accuracy of the decoding process.

## B.2 Limitation B - Schema Hallucination

Note that a syntactically valid SQL as generated by GCD techniques may not be executable (due to hallucination of database schema items, tables, and columns) and, even if the generated SQL is executable, it may result in wrong outcomes (due to semantic errors such as wrong usage of schema items in the SQL). Below two examples from our datasets highlight that GCD (syncode) does not reduce the hallucination related to schema items.

**Example 1:** What is the full name of each car maker, along with its id and how many models it produces?

Auto-Regressively Generated SQL: SELECT m.id, m.full_name, COUNT(DISTINCT ml.model) AS model_count FROM car_makers m JOIN model_list ml ON m.id = ml.maker GROUP BY m.id, m.full_name

Syncode Generated SQL: SELECT m.id, m.full_name, COUNT(DISTINCT ml.model) AS model_count FROM car_makers m JOIN model_list ml ON m.id = ml.maker GROUP BY m.id, m.full$_{n}ame$

**Both autoregressive and GCD (Syncode) approaches hallucinated the column name *full_name*, which is not the schema. The correct column name is *fullname*.**

TTD Approach Generated Correct SQL: SELECT m.id, m.fullname, COUNT(DISTINCT ml.model) AS model_count FROM car_makers m JOIN model_list ml ON m.id = ml.maker GROUP BY m.id, m.fullname ORDER BY m.fullname NULLS LAST;

**Example 2:** Give me Brazil's population and

life expectancies.
Auto-Regressively Generated SQL: SELECT c.population, c.life_expectancy FROM country c WHERE c.code = 'BRA';

Syncode Generated SQL: SELECT c.population, c.life_expectancy FROM country c WHERE c.code = 'BRA';

**Both autoregressive and GCD (Syncode) approaches hallucinated the column name *life_expectancy*, which is not the schema. The correct column name is *lifeexpectancy*.**

TTD Generated Correct SQL: SELECT c.population, c.lifeexpectancy FROM country c WHERE c.code = 'BRA';

The above provides an intuition why the number of Executable SQLs is less than that of our proposed approach.

## C Relevant Work - More References

There exists substantial amount of relevant research on the Text-To-SQL from databases, machine learning, and natural language communities. Early research efforts are largely driven by either rule based models (Christopher Baik and Jagadish, 2020; Jaydeep Sen and Sankaranarayanan, 2020) or sequence-to-sequence based models with encoder-decoder architectures (Octavian Popescu and Sheinin, 2022; Ruichu Cai and Liang, 2018). The advancement of deep learning architectures, such as BERT (Jacob Devlin and Toutanova, 2019), have contributed significantly to the design of SoTA models for Text-To-SQL problem (Brunner and Stockinger, 2021; Pengcheng Yin and Riedel, 2020).

With the emergence of large language models such as GPT 4[6] and LLaMA (Hugo Touvron and Lample, 2023), one of the prominent research efforts to address Text-To-SQL problem is prompt engineering (Linyong Nan and Radev, 2023; Aiwei Liu and Yu, 2023). In-context learning (Qingxiu Dong and Sui, 2023) is an approach that allows LLMs to identify the inherent patterns from contextual information, such as relevant examples, thereby leading to few-shot learning scenarios (Chang and Fosler-Lussier, 2023; Aiwei Liu and Yu, 2023; Linyong Nan and Radev, 2023; Dawei Gao and Zhou, 2023).

The efficiency of LLMs to address the Text-to-SQL problem can be further enhanced by fine-tuning LLMs using a collection of high-quality labeled data comprising of <example, SQL> tuples (Ruoxi Sun and Pfister, 2023; Dawei Gao and Zhou, 2023). Thus, in-context learning and fine-tuning form the two pillars of adopting LLMs to Text-To-SQL problem.

Generating explanations for SQL queries has been an active research area (Simitsis and Ioannidis, 2009) for a long time. (Andreas Kokkalis, 2012) proposed an approach in which the SQL query is represented using a graph and traversing this graph results in explanations. (Ahmed Elgohary, 2020) and (Ahmed Elgohary, 2020) proposed a template approach (or rule-based) to generate explanations. These approaches cannot handle any arbitrary SQL queries.

Finally, we refer the readers to the following surveys for more details on Text-to-SQL problem. First, (Deng et al., 2022) provides a detailed review of techniques and advances in the text-to-SQL problem by highlighting major challenges such as encoding the meaning of natural utterances, decoding SQL queries, and translating the semantics between these two forms. Second, (Zhu et al., 2024) surveys the text-to-SQL generations enhanced by the large language model, classifying them into prompt engineering, fine-tuning, pre-trained and agent groups according to training strategies.

## D Speculative Decoding with TTD Approach

Here we present our results by considering the following two configurations: (a) the standard speculative decoding (*SD*) approach; and (b) the speculative decoding augmented with our proposed TTD approach (*SD + TTD*). In particular, to setup the speculative decoding framework, we consider SQLCoder-7B as the draft model and SQLCoder-70B as the target model. Using vanilla speculative decoding *SD* model, the token rate is $5.29 \pm 0.06$; whereas the token rate using our proposed *SD+TTD* approach is $5.66 \pm 0.09$. Clearly, there is a speedup of $7\%$ in the token rate using our *SD+TTD* approach as compared to that of vanilla *SD* approach. Further, note that both auto-regressive and speculative decoding approaches generate the same token sequences and thus the execution accuracy of both these approaches remain the same. Accordingly, we don't report the execution accuracy (EX) numbers in this speculative decoding setting.

[6] https://openai.com/