Node-Level Uncertainty Estimation in LLM-Generated SQL

Hilaf Hasson*, **Ruocheng Guo**†
*Cohesity, hilaf.hasson@cohesity.com
†Intuit AI Research, ruocheng_guo@intuit.com

Abstract

We present a practical framework for detecting errors in LLM-generated SQL by estimating uncertainty at the level of individual nodes in the query's abstract syntax tree (AST). Our approach proceeds in two stages. First, we introduce a semantically aware labeling algorithm that, given a generated SQL and a gold reference, assigns node-level correctness without over-penalizing structural containers or alias variation. Second, we represent each node with a rich set of schema-aware and lexical features - capturing identifier validity, alias resolution, type compatibility, ambiguity in scope, and typo signals - and train a supervised classifier to predict per-node error probabilities. We interpret these probabilities as calibrated uncertainty, enabling fine-grained diagnostics that pinpoint exactly where a query is likely to be wrong. Across multiple databases and datasets, our method substantially outperforms token log-probabilities: average AUC improves by +27.44% while maintaining robustness under cross-database evaluation. Beyond serving as an accuracy signal, node-level uncertainty supports targeted repair, human-in-the-loop review, and downstream selective execution. Together, these results establish node-centric, semantically grounded uncertainty estimation as a strong and interpretable alternative to aggregate sequence-level confidence measures.

Keywords: Text-to-SQL, Uncertainty Estimation, Abstract Syntax Trees, Error Detection, Calibration

1. Introduction

With the advent of large language models (LLMs) (Achiam et al., 2023; Brown et al., 2020; Hoffmann et al., 2022; Guo et al., 2025), recent advancements in Text-to-SQL have been largely driven by solutions that leverage LLMs to generate SQL queries, which now dominate benchmark leaderboards such as BIRD (Li et al., 2023), Spider 1.0 (Yu et al., 2018), and Spider 2.0 (Lei et al., 2024). (See Deng et al., 2025; Gao et al., 2023; Shkapenyuk et al., 2025.)

Errors in SQL generation—or, more broadly, in structured output generation—can be categorized into two main types: syntactic errors, which involve violations that make the SQL query unable to run; and semantic errors, often referred to as schema-linking errors in the context of SQL, which result in a SQL query that can be executed, but misinterprets the meaning of the question, including selecting the wrong table, column, or value. For syntactic errors, rule-based solutions abound, with Shen et al., 2025 being salient among them. For semantic errors, following the general principle articulated in Kuhn et al., 2023 that "very uncertain generations should be less likely to be correct", many state-of-the-art generative Text-to-SQL approaches incorporate a voting mechanism (e.g., Deng et al., 2025; Shkapenyuk et al., 2025). However, Xiong et al., 2023 observed that LLMs can be overconfident in their responses, indicating that supervised uncertainty estimation could offer advantages when sufficient data is available. Since SQL generation is open-ended, we focus our approach on supervised node-level uncertainty quantification. To that end, we begin by introducing the first semantically-aware labeling algorithm for correctness of nodes in pairs of generated and ground truth SQL—a priori an ambiguous task (see Section 3.1). See Appendix A for how our design choices make the node-level correctness labeling agree with intuition in 13 cases we used while prototyping, while node-level hashing (the only alternative in related works) does not. We then create a training dataset with carefully designed, schema-informed features and labels at the node level (see Section 3.2). Finally, we train a gradient-boosted tree model on this dataset to estimate node-level uncertainty (see Section 4). In particular, we interpret the term "uncertainty" as calibrated predictive probability of error.

Our experiments show conclusively that our proposed approach vastly outperforms the arithmetic mean of the logarithm of token probabilities from the LLM which generated the node as a predictor of error, suggesting the aforementioned principle in Kuhn et al., 2023 can be significantly improved upon in the case of node-level uncertainty quantification.

More precisely, our contributions are as follows:

1. We introduce a semantically aware algorithm for labeling whether an individual node in a generated SQL AST is correct, given a generated query and a gold reference (Algorithm 1). Appendix A shows 13 illustrative cases used during prototyping. (See Section 3.1.)

2. We design a rich set of 72 node-level, schema-aware features for detecting semantic SQL errors (e.g., identifier validity, alias resolution,

Work done entirely while Hilaf Hasson was at Intuit AI Research.

Proceedings of the Fifteenth Language Resources and Evaluation Conference (LREC 2026), pages 2145–2153
11–16 May 2026.

scope ambiguity, and type compatibility). (See Section 3.2.)

3. We report three experimental settings demonstrating large AUC gains over token log-probabilities: (1) in-database evaluation on BIRD dev; (2) cross-database evaluation within BIRD dev; and (3) cross-dataset evaluation where the model is trained on SynSQL-2.5M and tested on held-out BIRD dev databases. Across these settings, performance degrades as training data becomes less in-distribution (Experiment 1 > Experiment 2 > Experiment 3). (See Section 4.)

2. Related Work

Most Text-to-SQL literature treats uncertainty quantification as an integral part of an overarching solution, which is not analyzed separately. For example Xie et al., 2025 employs consistency alignment, self-consistency voting, and error correction to mitigate hallucinations and variability, but does not quantify uncertainty outright.

The recent work (Somov and Tutubalina, 2025) introduces entropy-based confidence estimates with selective classifiers to enable query rejection under distribution shift. Complementing this, Liu et al., 2025 propose multivariate Platt scaling leveraging sub-clause frequency statistics across sampled SQL generations.

AST-based comparison has also been explored for SQL equivalence checking. The `sqlgpt-parser` package labels correctness at the *query level*, treating two SQL queries as "equal" only if their ASTs are isomorphic trees with identical node types and attribute values at every position, matching nodes element-by-element in order and ignoring only line and position metadata (eosphoros ai, 2023). This approach does not include alias-to-base-table normalization, treatment of qualified vs. unqualified columns, handling of symmetric or anti-symmetric operators, or reconciliation between contextual and global alignment when structure differs.

Another line of work, PPOCoder (Shojaee et al., 2023) employs AST and data-flow graph (DFG) overlaps as sequence-level reward components in a reinforcement-learning framework for code generation. It does not define SQL-specific node equivalence, produce per-node correctness labels, or estimate calibrated node-level uncertainty.

A more recent work (Anonymous, 2024), is the first to introduce node-level uncertainty estimation in Text-to-SQL by training a transformer model (codellama 7B) that takes the original question, the schema, and the generated SQL, and outputs a confidence score per node. This differs from our work in 2 significant ways: The first is that their solution is a black-box model over serialized input. Namely, the question, schema, and serialized generated SQL are all fed in as a single text, and the model is expected to learn their structure implicitly. This is in contrast to the current work, where nodes are featurized explicitly—allowing for a more explainable model and enabling selection of informative features. Second, their definition of ground truth is a very high level heuristic that is not semantically grounded: for each node in the generated SQL they hash its subtree using a relational algebra tree (RAT); then they attempt to find a node with the same hash in the ground truth query. This causes errors to propagate upwards: when a child node is incorrect, all its ancestors are marked incorrect as well, even when their structure and logic are otherwise valid.

For example, with the ground truth SQL:

```
SELECT department.name, SUM(employee.
    salary)
FROM department
JOIN employee ON department.id =
    employee.department_id
GROUP BY department.name
```

And model-generated SQL is:

```
SELECT department.name, SUM(employee.
    wage)
FROM department
JOIN employee ON department.id =
    employee.department_id
GROUP BY department.name
```

The only error in this instance is the incorrect use of `employee.wage` in place of the correct column `employee.salary`. However, the RAT hashing mechanism compares entire subtrees. Since the subtree rooted at `SUM(employee.wage)` does not match the one rooted at `SUM(employee.salary)`, the `SUM` node receives a different hash and is marked incorrect. There is also no explicit accounting of alias resolution, handling of symmetry and anti-symmetry, and so on; which causes equivalent statements to be labeled as incorrect.

3. Proposed Method

Our framework identifies SQL errors by reframing query validation as a fine-grained classification problem at the level of individual query components, i.e., nodes in the AST of each SQL query. Any given SQL query is first parsed into an AST, a standard hierarchical representation of its structure. We then use a Gradient Boosted Decision Tree (GBDT) model to predict the probability that each node in the AST is erroneous. This node-level probability serves as a direct measure of uncertainty, pinpointing the specific parts of the query that are likely incorrect.

SQL:

```
SELECT a.name
FROM artist AS a
WHERE a.id = 7
```

Simplified AST (nodes):

```
Select
|-- Column(name) [table=a]
|-- From
|   `-- Table(artist)
|       `-- TableAlias(a)
`-- Where
    `-- EQ
        |-- Column(id) [table=a]
        `-- Literal(7)
```

Figure 1: Example of an SQL query and a simplified AST view (exact node naming varies by parser). Each line item corresponds to an AST node that our method labels and featurizes.

3.1. Ground Truth Generation

To train a robust supervised learning model for SQL error detection, we construct a dataset of machine-generated SQL queries annotated with fine-grained, node-level error labels. Each generated query is paired with a gold-standard, human-verified reference query. Labels are assigned through recursive comparison of their abstract syntax trees (ASTs) with ground truth, using a context-preserving, semantic alignment strategy.

Node Alignment and Labeling The labeling process performs a top-down traversal over the generated AST. For each pair (n_{gen}, n_{gold}), we first test semantic recursive equivalence; which will be defined clearly below. If true, we mark all nodes in the subtree rooted at n_{gen} as correct and stop recursing under this pair. Otherwise, we mark n_{gen} as error and recurse on every child pair (c_{gen}, c_{gold}) formed by the children of n_{gen} and the children of n_{gold} (order-invariant, cartesian). After this traversal, we suppress blame for structural containers (**SELECT**, **FROM**, **WHERE**, **GROUP BY**, **HAVING**, $=$, $\neq$) when only their children differ.

This traversal continues recursively through the structure of the generated AST, with alignment decisions based only on local subtree comparisons; the gold AST is not searched globally during this phase. As a result, some semantically correct nodes may remain labeled as errors due to greedy or structurally mismatched alignment decisions. To further improve the labeling heuristic, we do a final, context-agnostic pass that globally re-evaluates nodes still labeled as errors—trading structural precision for increased recall. The precise algorithm is spelled out in Algorithm 1; Appendix A illustrates, across 13 cases, how these design choices align labels with intuitive judgments.

Semantic Equivalence Criteria

Two nodes n_{gen} and n_{gold} are considered semantically equivalent if the following conditions hold:

1. Type Errors:
   (a) The node types are identical, *or* they are a known operator pair (e.g., `>` vs. `<`) that permits equivalence under operand swapping.
   (b) If the nodes are binary operators (`=`, `>`, etc.), they are equivalent if:
      - both children match in order, or
      - the operator is symmetric or anti-symmetric and the children match in reverse order.
2. Content Errors:
   (a) If the nodes are **Column** expressions, their unqualified names must match exactly; qualifiers (aliases or table names) must either match directly, or resolve to the same base table via an alias-to-table map extracted from each AST.
   (b) If the nodes are literals, identifiers, or base tables, their content (e.g., string or numeric value) must match exactly.

Remark 1. To reduce over-labeling, the following heuristics are applied:

- Nodes like **SELECT**, **FROM**, **WHERE**, **GROUP BY** , and comparison operators (=, =!) are not blamed if only their children differ.
- Structural nodes like **JOIN** and **LIMIT** *are* blamed if they are incorrect or inserted.

Illustrative Example

- **Generated SQL:** `SELECT name FROM artists ORDER BY name`
- **Gold SQL:** `SELECT name FROM artist WHERE name = 'A'`

Here, the labeler would output:

- An error for the table name `artists` (vs. `artist`),
- Errors for the additional **ORDER BY** clause and its child nodes,
- Since the goal of the algorithm is only to label nodes in the generated SQL as either correct or incorrect, the missing **WHERE** does not arise in the labeling,

- The `name` in `name = 'A'` will be labeled as incorrect in the first-pass traversal across generated and ground truth ASTs, but will be relabeled as correct in the final context-agnostic pass.

Remark 2. Omissions are not penalized because they cannot be assigned node-level probabilities. Query-level completeness metrics (e.g., clause coverage or overlap with the gold query) are out of scope for this paper, but could be combined in future work.

We now proceed to spell out some details of the subtleties in the definition of Content Errors in the Semantic Equivalence Criteria.

Content Error Details Identifiers, literals, and function names must match exactly. Otherwise, it is a content error.

- **Identifiers:** `artists` vs. `artist`
- **Literals:** `1` vs. `2`
- **Functions:** **`MAX`**`()` vs. **`SUM`**`()`

This includes mismatches in base table names and literal constants.

Table nodes are considered correct iff their corresponding identifiers are correct after an alias-to-table mapping. Column nodes may include explicit qualifiers such as table names or aliases (e.g., `a.name` vs. `name`); these are treated as semantically equivalent when:

- The column name is identical, and
- The qualifiers (if any) either match, or resolve to the same base table via alias mapping.

Alias names themselves are not required to match, and will not trigger content errors. A mapping from alias $\rightarrow$ base table is computed for both ASTs and used to normalize qualified comparisons.

Post-Processing A final pass reclassifies any node that matches an equivalent node elsewhere in the gold tree, to correct for greedy alignment mismatches. (Note that the first, contextual, pass is not redundant because being contextually aware leads to more "correct" labels.)

To show why the final pass is required consider the ground truth **`SELECT`** `name` **`FROM`** `artists` and generated SQL **`SELECT`** `name` **`FROM`** `artist`. Because the tree diverges at the level of the table, the first, contextual, pass of Algorithm 1 would have labeled `name` as incorrect, which does not agree with intuition. In Algorithm 1 the function `AreRecursivelyEquivalent` refers to verifying that the nodes are semantically equivalent in the sense of Section 3.1, that their children have a 1-1 correspondence of semantically equivalent nodes, and so on recursively.

See Appendix A to see how 5 out of the 13 unit tests we include there for node-level correctness labeling to agree with intuition would have failed if not for the post-processing.

Algorithm 1: Labeling Generated AST via Alignment with Gold AST

Input: Generated AST T_{gen}, Gold AST T_{gold}
Output: Label map $\mathcal{L}$ assigning 0 (correct) or 1 (error) to each node in T_{gen}

Initialize $\mathcal{L}$ by labeling all nodes in T_{gen} as 1;
Build alias maps from both ASTs to normalize qualifiers;
function `AlignAndLabel`(n_{gen}, n_{gold}) **is**
  if `AreRecursivelyEquivalent`(n_{gen}, n_{gold}) **then**
    Mark all nodes in the subtree rooted at n_{gen} as correct in $\mathcal{L}$; **return**;
  Mark n_{gen} as error in $\mathcal{L}$;
  foreach *child* c_{gen} *of* n_{gen} **do**
    foreach c_{gold} *in children of* n_{gold} **do**
      `AlignAndLabel`(c_{gen}, c_{gold})
end function
`AlignAndLabel`(*root of* T_{gen}, *root of* T_{gold});
Suppress blame for structural nodes if only their children are labeled as errors;
foreach *node* n_{gen} *in* T_{gen} *still labeled as error* **do**
  if n_{gen} *matches any* n_{gold} *in* T_{gold} *under* `AreRecursivelyEquivalent()` **then**
    Relabel n_{gen} as correct;
return $\mathcal{L}$;

3.2. Featurization

The node classification model does not see the gold query; its ability to predict the ground truth label for a given node is derived entirely from the feature vector associated with a given node. Our featurization process provides a rich, evidence-based description of each node, capturing its structure, semantic validity, and context. Features are grouped into several logical categories.

1. Base Structural Features This foundational group describes the node's identity and position within the AST. It includes the node's syntactic type (e.g., **`Column`**, **`Join`**), its depth relative to the root, and the type of its immediate parent.

Node Type	Prop.	Prop. False	AUC (ours)	AUC (Logprobs)
All	100%	13.07%	**76.51%**	49.07%
Identifier	40.06%	9.74%	**63.91%**	51.42%
Column	15.25%	8.32%	**56.92%**	49.29%
Literal	5.99%	10.35%	**69.59%**	42.44%
Table	5.73%	4.66%	48.25%	**49.63%**
TableAlias	4.90%	6.61%	**61.48%**	54.18%

Table 1: Results for the first experiment, where each of the databases in BIRD dev is split into training and test sets. The model is trained on all of the training sets and tested on the test sets across the databases. "Prop." is how often this node occurs in the LLM-generated data; and "Prop. False" is the ratio of nodes that are deemed as False compared to ground truth (see Section 3.1). Here, AUC refers to area under the ROC curve.

2. Schema Consistency Features This is a critical feature set for detecting content errors. It validates a node's content against the target database schema, using a scope resolution module that correctly interprets table aliases and nested queries. Key features include:

- **Identifier Validity:** A binary feature checking if a table or column name exists in the schema. For an erroneous node like the `Table artists`, this feature provides a powerful error signal.
- **Qualifier Scope Validity:** For a qualified column like `T.name`, this checks if the qualifier `T` is a valid table or alias within the current scope of the query.
- **Column Ambiguity:** A feature that detects if an unqualified column name could refer to columns in multiple tables that are currently in scope, a common source of error in queries with multiple joins.

3. Lexical and Typo-Detection Features This group helps the model identify errors in identifiers that are syntactically well-formed but semantically incorrect (i.e., typos). These features are primarily for `Identifier` and **`Column`** nodes.

- **Levenshtein Distance:** If an identifier is not found in the schema, we compute its Levenshtein (edit) distance to the closest valid identifier (table or column) in the entire schema. A small distance is a strong indicator of a typo.
- **Name Patterns:** A series of binary features capturing lexical patterns in an identifier's name, such as whether it contains numbers, underscores, or is written in 'ALL_CAPS' or 'mixedCase'.

4. Contextual and Heuristic Features This broad category captures common SQL error patterns that depend on a node's relationship with other nodes, helping the model to infer structural and type-based errors.

- **Aggregate Context:** A feature specifically designed to detect one of the most common SQL errors: the presence of a non-aggregated column in a **`SELECT`** list that also contains an aggregate function (e.g., **`SUM`**`()`) but lacks a corresponding **`GROUP BY`** clause.
- **Data Type Compatibility:** For binary operations (`=`, `>`, etc.), we resolve the data types of the left and right operands and include a feature representing their compatibility (e.g., **`NUMERIC`** vs. `STRING`).
- **Operator-Specific Features:** For certain node types, we add highly specialized features. For a **`Like`** node, we extract features about its pattern (e.g., presence of wildcards, pattern length). For an **`In`** node, we capture the number of elements in its list.

5. Feature Vector Consistency A key aspect of our implementation is ensuring that every node, regardless of its type, produces a feature vector of the same length. Features that are not applicable to a given node are assigned a neutral default value. For example, the 'Levenshtein Distance' feature is only computed for `Identifier` nodes; for all other node types (e.g., **`Select`**, **`Join`**), it defaults to a high constant value representing "no close match." Likewise, boolean features default to `0`. This ensures that the GBDT model receives a consistently shaped input for any node in any given SQL query.

4. Experiments

We use two sources of model-generated SQL paired with gold references: (i) the dev slice of the BIRD dataset (Li et al., 2023) and (ii) SynSQL-2.5M (Li et al., 2025), a very large synthetic dataset created for training the Text-to-SQL model OmniSQL. In both cases, we generate SQL queries with OmniSQL-7B, which as of the writing of this paper ranks #29 on the BIRD-SQL leaderboard. We chose OmniSQL-7B because it is open-source, it is

Node Type	Trained on BIRD	Trained on SynSQL-2.5M	Logprobs AUC
All	**69.46%**	66.47%	48.45%
Identifier	50.43%	**56.14%**	48.63%
Column	45.48%	**51.72%**	46.79%
Literal	**58.91%**	57.93%	51.16%
Table	52.61%	**52.83%**	46.91%
TableAlias	51.43%	**67.67%**	42.53%

Table 2: Results for the second and third experiments. In this table, the evaluation is done on the queries associated with the databases california schools, card_games, and toxicology in the BIRD dev dataset. The column "Trained on BIRD" refers to a model trained on all of the queries in the BIRD dev dataset not associated with these 3 databases. The "Trained on SynSQL-2.5M" column refers to a model trained *only* on queries from 453 databases from SynSQL-2.5M. AUC in this table refers to area under the ROC curve.

close to SoTA performance, and it is small enough to run experiments efficiently.

We prompt the model using a standard Text-to-SQL format that includes the natural-language question and a serialization of the target database schema.

Both BIRD dev and SynSQL-2.5M span multiple databases, each containing multiple tables. BIRD dev has 11 databases (1534 queries). For SynSQL-2.5M, we generated SQL for 453 databases (67,901 queries) rather than the full corpus. We evaluate both in-database and cross-database generalization.

In Experiment 1 (in-database, BIRD dev), we split queries within each database into Train (80%) and Test (20%). We train a single model over the union of all Train queries across all databases and evaluate on the union of all Test queries across all databases. We use LightGBM (Ke et al., 2017) with parameters `n_estimators=100, learning_rate=0.05`. The results are in Table 1. We can see that our approach is highly effective in detecting whether generated nodes are correct, with the following results on common node types: 69.59% AUC on `Literal`, 63.91% AUC on `Identifier`, and 61.48% on `TableAlias`. We can also see that this approach is far superior to log probabilities, whose AUC is similar to random chance, suggesting that the principle "very uncertain generations should be less likely to be correct" articulated in Kuhn et al., 2023 can be significantly improved upon in the case of structured output.

Experiment 2 is cross-database (within BIRD dev). We split the BIRD dev databases into Train Databases and Test Databases (california schools, card_games, toxicology). We train on all queries from the train databases and test on all queries from the test databases.

Experiment 3 is cross-dataset: we train on SynSQL-2.5M (the 453 databases for which we generated SQL) and test on the same BIRD dev test databases as in Experiment 2 for a fair comparison.

As shown by results in Table 2, both are superior to log probability and are reasonably good on `Literal`. However, we observe that the more that the training data is in-distribution the better the model performs: Experiment 1 trained on the same databases it tested on; Experiment 2 trained on different databases from test, but from the same dataset of queries; and Experiment 3 was across datasets and performed the worst. This suggests that while in principle it might be possible to make a foundation model for node-level uncertainty, it is currently still best to train in-distribution.

Conclusions

We introduced a semantically grounded framework for classifying correctness of LLM-generated SQL at the node level. Our approach contributes the first semantically aware labeling algorithm, a rich set of schema-aware features for node classification, and demonstrates substantial gains over log-probability baselines across multiple datasets. Together, these advances establish a practical and interpretable foundation for fine-grained uncertainty estimation in Text-to-SQL generation.

Limitations

The ground truth labeling procedure in Section 3.1 was designed to reflect intuitive assessments of uncertainty and has shown strong alignment in practice. While the method offers a solid foundation for meaningful analysis, occasional divergence from intuition may arise due to the inherent ambiguity in defining node-level correctness. Finally, we view the integration of uncertainty estimation into agentic mechanisms that dynamically revise or guide SQL generation as a promising direction for future work.

5. Bibliographical References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anonymous. 2024. Coarse and fine-grained confidence calibration of LLM-based text-to-SQL generation. In *Submitted to ACL Rolling Review - June 2024*. Under review.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. 2025. Reforce: A text-to-sql agent with self-refinement, format restriction, and column exploration. *arXiv preprint arXiv:2502.00675*.

eosphoros ai. 2023. sqlgpt-parser. https://github.com/eosphoros-ai/sqlgpt-parser.

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

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30.

Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation. *arXiv preprint arXiv:2302.09664*.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, et al. 2025. Omnisql: Synthesizing high-quality text-to-sql data at scale. *arXiv preprint arXiv:2503.02240*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2023. 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*, 36:42330–42357.

Terrance Liu, Shuyi Wang, Daniel Preotiuc-Pietro, Yash Chandarana, and Chirag Gupta. 2025. Calibrating llms for text-to-sql parsing by leveraging sub-clause frequencies. *arXiv preprint arXiv:2505.23804*.

Jiawei Shen, Chengcheng Wan, Ruoyi Qiao, Jiazhen Zou, Hang Xu, Yuchen Shao, Yueling Zhang, Weikai Miao, and Geguang Pu. 2025. A study of in-context-learning-based text-to-sql errors. *arXiv preprint arXiv:2501.09310*.

Vladislav Shkapenyuk, Divesh Srivastava, Theodore Johnson, and Parisa Ghane. 2025. Automatic metadata extraction for text-to-sql. *arXiv preprint arXiv:2505.19988*.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*.

Oleg Somov and Elena Tutubalina. 2025. Confidence estimation for error detection in text-to-sql systems. *arXiv preprint arXiv:2501.09527*.

Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment. *arXiv preprint arXiv:2502.14913*.

Miao Xiong, Zhiyuan Hu, Xinyang Lu, Yifei Li, Jie Fu, Junxian He, and Bryan Hooi. 2023. Can llms express their uncertainty. *An Empirical Evaluation of Confidence Elicitation in LLMs*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

A. Illustrative Node-Level Labeling Examples

This appendix presents 13 paired SQL comparisons (generated vs. gold) and the node-level labels our algorithm produces. Each example highlights a specific design choice: (i) contextual subtree alignment with early stop on recursive equivalence; (ii) blame suppression for structural containers; (iii) semantic equivalence (alias/qualification normalization; operator symmetry/anti-symmetry); and (iv) a final global, acontextual rescue pass that corrects greedy misalignments.

Ex. 1: Perfect Match

Generated: `SELECT name FROM people`
Gold: `SELECT name FROM people`
Nodes Labeled as Incorrect: none.
Why it works: Early stop on full-subtree equivalence; no recursion needed.

Ex. 2: Content Error on Table Only

Generated: `SELECT name FROM artists`
Gold: `SELECT name FROM artist`
Nodes Labeled as Incorrect: `Table(artists)`, `Identifier(artists)`.
Why it works: Content mismatch on base table identifier; column sibling `name` remains correct (qualification semantics prevent cascaded blame). Without the global pass, the sibling `Column(name)` can be provisionally overblamed due to local divergence; the global pass reclassifies it as correct.

Ex. 3: Content Error on Literal Only

Generated: `SELECT * FROM t WHERE a = 1`
Gold: `SELECT * FROM t WHERE a = 2`
Nodes Labeled as Incorrect: `Literal(1)` only.
Why it works: Containers (`SELECT`, `FROM`, `WHERE`, and the comparison operator `=`) are suppressed; the global pass rescues `Star`, `Table(t)`, and `Column(a)`, leaving only `Literal(1)` incorrect. Without the global pass, this case fails (siblings remain incorrectly blamed).

Ex. 4: Type Error on Operator Only

Generated: `SELECT * FROM t WHERE a > 1`
Gold: `SELECT * FROM t WHERE a = 1`
Nodes Labeled as Incorrect: `GT(a > 1)` only.
Why it works: Operator type differs; containers are suppressed; children are equivalent. Without the global pass, `Star`, `Table(t)`, and `Column(a)` can be overblamed; the global pass rescues them, leaving only `GT`.

Ex. 5: Structural Insertion Blamed (Clause + Children)

Generated: `SELECT * FROM t ORDER BY a`
Gold: `SELECT * FROM t`
Nodes Labeled as Incorrect: `Order(ORDER BY a)`, `Ordered(a)`, `Column(a)`, `Identifier(a)`.
Why it works: Extra clause and its subtree are structural insertions and are blamed precisely; containers elsewhere are suppressed. Without the global pass, `Star(*)` may be overblamed; the global pass rescues it, leaving only the `ORDER BY` subtree.

Ex. 6: Structural Omission Not Blamed

Generated: `SELECT * FROM t`
Gold: `SELECT * FROM t ORDER BY a`
Nodes Labeled as Incorrect: none (on the generated query).
Why it works: Omissions in the generated SQL are not penalized because node-level probabilities must attach to existing generated nodes. Without the global pass, `Star`, `Table(t)`, and `Identifier(t)` can be incorrectly blamed; the global pass rescues them so omissions are not penalized.

Ex. 7: Symmetric Operator Equivalence

Generated: `SELECT * FROM t WHERE a = b`
Gold: `SELECT * FROM t WHERE b = a`
Nodes Labeled as Incorrect: none.
Why it works: `=` is symmetric; children can match in reverse order.

Ex. 8: Anti-Symmetric Operator Equivalence

Generated: `SELECT * FROM t WHERE a > b`
Gold: `SELECT * FROM t WHERE b < a`
Nodes Labeled as Incorrect: none.
Why it works: `>` and `<` are anti-symmetric pairs; swapping children and flipping the operator preserves equivalence.

Ex. 9: Alias Names May Differ (Still Correct)

Generated: `SELECT x.name FROM artist AS x`
Gold: `SELECT a.name FROM artist AS a`
Nodes Labeled as Incorrect: none.
Why it works: Alias wrappers are ignored and normalized via alias→table mapping; qualified vs. qualified with different alias names is equivalent.

Ex. 10: Unused Alias Declaration Not an Error

Generated: `SELECT name FROM artist AS a`
Gold: `SELECT name FROM artist`

Nodes Labeled as Incorrect: none.
Why it works: Alias presence/absence is normalized away when it does not change base-table resolution.

Ex. 11: Qualified vs. Unqualified Column (Unambiguous)

Generated: `SELECT a.name FROM artist AS a`
Gold: `SELECT name FROM artist`
Nodes Labeled as Incorrect: none.
Why it works: Column identifiers match; either side unqualified and alias→table normalization yield equivalence (no ambiguity in scope).

Ex. 12: Wrong Base Table is an Error

Generated: `SELECT name FROM albums AS a`
Gold: `SELECT name FROM artist AS a`
Nodes Labeled as Incorrect: `Table(albums)` and/or `Identifier(albums)`.
Why it works: Base table differs; alias normalization cannot reconcile distinct base tables.

Ex. 13: Wrong Alias Used in Column is an Error

Generated: `SELECT b.name FROM artist AS a`
Gold: `SELECT a.name FROM artist AS a`
Nodes Labeled as Incorrect: `Column(b.name)` and/or `Identifier(b)`.
Why it works: Qualifier `b` does not resolve in scope to the correct base table; column reference is invalid under alias normalization.

What could go wrong (and why it does not here).

- **Greedy alignment cascades:** A local mismatch (e.g., a literal or operator) can transiently overblame siblings under contextual alignment. Blame suppression for structural containers, followed by the global rescue pass, reclassifies semantically equivalent nodes back to correct (empirically required in Ex. 2–6).
- **Strict structural equality brittleness:** Reordering children, operator symmetry, or qualified vs. unqualified columns would look "unequal." Semantic equivalence rules (operator symmetry/antisymmetry, alias→table normalization, unqualifiedvsqualified tolerance) prevent false errors (Ex. 7–11).
- **Alias noise:** Different alias names or unused aliases can trigger spurious errors. Ignoring alias wrappers and normalizing to base tables prevents this (Ex. 9–11).
- **Structural insertions vs. omissions:** Extra clauses are blamed precisely at the inserted subtree (Ex. 5), while omissions in the generated SQL are not blamed (Ex. 6), reflecting our nodelevel probability objective.

Ablation Evidence. Disabling the global (acontextual) pass causes 5 of the 13 examples (Ex. 2–6) to fail—precisely those relying on rescuing transiently overblamed siblings. This validates the necessity of the global pass for precise nodelevel labels.

These examples mirror the unit tests in our repository and demonstrate how contextual alignment, semantic equivalence, container suppression, and global rescue work together: the first pass localizes and structures blame; the suppression step avoids noisy container blame; and the global pass restores recall by reclassifying semantically equivalent nodes that contextual alignment alone cannot certify.