AtomSQL: Interactive Disambiguation of NL-to-SQL via User-Guided Atom-Level Alignment

Aritra Mazumder
University of Utah
Salt Lake City, Utah
aritra.mazumder@utah.edu

Parth Desai
University of Utah
Salt Lake City, Utah
parth.desai@utah.edu

Fuheng Zhao
University of Utah
Salt Lake City, Utah
fuheng.zhao@utah.edu

Anna Fariha
University of Utah
Salt Lake City, Utah
afariha@cs.utah.edu

ABSTRACT

Natural language to SQL (NL-to-SQL) systems are widely used for querying databases in natural language, achieving high accuracy on benchmark datasets. In practice, however, these systems face a fundamental challenge: a single natural language question can have multiple reasonable SQL interpretations, differing in predicates, aggregation functions, join paths, or set modifiers such as `DISTINCT`. This ambiguity is typically localized to specific parts of a query rather than the query as a whole. Existing approaches either assign a single confidence score to the entire query, which does not reveal where models disagree, or engage users in multi-turn clarification dialogues that resolve ambiguity opaquely, leaving users uninformed about the process.

We demonstrate AtomSQL, a system that helps users resolve ambiguity in NL-to-SQL at the "atom-level" granularity of query components, and learns from their choices to improve NL-to-SQL disambiguation over time. AtomSQL runs multiple NL-to-SQL models on a user's question, decomposes each generated query into *atoms* (clause-level units such as join conditions, predicates, and aggregation functions), and computes a confidence score for each atom based on cross-model agreement. The interface highlights the atoms and their possible NL interpretations, so the user can pinpoint exactly where interpretations differ and select the one that matches their intent. AtomSQL records these user interactions and uses them to personalize future query results toward that user's preferences.

PVLDB Reference Format:
Aritra Mazumder, Parth Desai, Fuheng Zhao, and Anna Fariha. AtomSQL: Interactive Disambiguation of NL-to-SQL via User-Guided Atom-Level Alignment. PVLDB, 19(12): 4514 - 4517, 2026. doi:10.14778/3827998.3828053

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/aritra741/AtomSQL.

1 INTRODUCTION

Natural language to SQL (NL-to-SQL) systems are increasingly used to help users query databases without writing SQL. Recent systems generate executable SQL queries for complex questions and achieve high accuracy on benchmark datasets like Spider [11]. As a result, NL-to-SQL is often treated as a solved or nearly solved problem from an accuracy perspective.

However, in practice, users face a fundamental challenge: a natural language question may have multiple reasonable SQL interpretations, and these generated queries are both valid and executable. Real user questions are often underspecified: the same question can map to several distinct SQL queries that differ in predicates, aggregation functions, join paths, or set modifiers like `DISTINCT`. Recent work [3, 9, 12] has formally characterized this phenomenon, showing that a single question can have multiple valid SQL answers and that such ambiguity is prevalent in real-world queries. This ambiguity is often more pronounced to specific parts of the SQL query rather than the entire query, where multiple NL-to-SQL systems agree on most parts of a query but disagree on a few components.

One approach to handle this ambiguity is to assign a confidence score to each query generated from the NL question. When the system is uncertain, a low score can warn the user of possible misinterpretation of their intent. A more interactive alternative is to detect ambiguity and engage the user in a clarification dialogue, asking targeted questions until the intended SQL is pinned down. Both directions capture something important: confidence signals help users decide when to trust the output, and clarification dialogues can resolve ambiguity before a query is executed. Yet, as shown in Example 1, each also leaves a significant gap when confronted with the localized, component-level nature of SQL ambiguity.

Example 1 (Atom-level ambiguity in NL-to-SQL). *Consider an administrator querying a student-pet database. She wants to find students meeting certain age and pet ownership criteria for a housing allocation report, so she asks:*

Q: *"Show two older students who have two pets."*

When multiple NL-to-SQL models process this question, they generate substantially different queries:

***T5-LM** generates:*

```
SELECT has_pet.StuID
FROM Has_pet
JOIN Student ON has_pet.StuID = student.StuID
GROUP BY has_pet.StuID
HAVING COUNT(*) = 2
```

***Llama 3 (8B parameters)** generates:*

```
SELECT *
FROM Student
WHERE Age > (SELECT AVG(Age) FROM Student)
  AND LName IN (
    SELECT student.LName FROM Student
    JOIN Has_Pet ON student.StuID = has_pet.StuID
    JOIN Pets ON has_pet.PetID = pets.PetID
    GROUP BY student.LName
    HAVING COUNT(pets.PetType) = 2)
```


Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828053

***SQLCoder** generates:*

```
SELECT s.fname, s.lname, COUNT(h.petid) AS pet_count
FROM student s
JOIN has_pet h ON s.stuid = h.stuid
GROUP BY s.fname, s.lname
HAVING COUNT(h.petid) >= 2
ORDER BY s.age DESC
LIMIT 2;
```

All three queries are schema-valid and executable, but they encode different interpretations. Some parts are stable across models (all agree on joining `Student` *and* `Has_Pet`*), while others diverge: whether "two pets" means exactly two or at least two, whether "older" requires an age predicate, and whether to apply* `ORDER BY` *and* `LIMIT`*.*

Related work. The example above reflects a recurring pattern in practice. Most NL-to-SQL systems [5, 6, 8] produce a single SQL query without modeling ambiguity or exposing uncertainty to the user, and existing efforts to address this fall into two categories, each with a shortcoming.

Query-level confidence methods [1, 10] assign a single confidence score to each generated SQL query, treating the entire query as an atomic unit. In our example, a confidence score might flag the output as uncertain, but it cannot tell whether the disagreement is about the `HAVING` threshold, the age predicate, or both. Ambiguity in NL-to-SQL is typically localized to specific components. A whole-query score does not reveal where the models disagree, so the user cannot tell which parts of the query to trust and which to question.

Interactive clarification systems [2, 4, 7, 12] use multi-turn conversations to produce a single SQL query. These systems ask the user targeted questions to resolve ambiguity, but the interaction is one-sided: the user answers questions without ever seeing the candidate queries that motivated them. In our example, such a system might ask "Do you mean exactly two pets or at least two?" without revealing that there are also open questions about the age predicate and whether to apply an `ORDER BY` clause. The user resolves one ambiguity while remaining unaware of the others, and the final query may still not match their intent.

Recent datasets and benchmarks [3, 9] demonstrate that ambiguity is prevalent in real-world NL-to-SQL queries. AMBROSIA [9] and PRACTIQ [3] formally characterize how a single question can yield multiple valid SQL answers.

AtomSQL. We address these limitations by treating each SQL query as a composition of *atoms*: indivisible semantic units such as tables, join conditions, predicates, aggregation functions, and modifiers like `DISTINCT`. AtomSQL runs multiple NL-to-SQL models, decomposes each output into atoms, and compares atoms across models. Cross-model agreement yields per-atom confidence scores that show where interpretations diverge, providing the atom-level granularity that whole-query confidence scores lack. The interface shows the generated query with uncertain atoms highlighted and the available alternatives for each atom shown alongside. When a user selects a preferred interpretation, AtomSQL learns this preference and applies it to future queries.

Demonstration. In our demonstration, participants will interact with AtomSQL and observe how it exposes ambiguity in NL-to-SQL queries. Using real-world datasets, we will showcase how AtomSQL identifies which query atoms are stable versus uncertain, presents alternative interpretations for ambiguous atoms, and adapts to user preferences over time. Participants will experience how the interface enables informed decision-making without requiring multi-turn conversations.

2 SYSTEM OVERVIEW

We model NL-to-SQL ambiguity as a probabilistic alignment between natural-language (NL) atoms and SQL atoms. AtomSQL combines cross-model agreement and user preference through five components: (1) a **model execution layer**, (2) an **atom decomposer**, (3) an **agreement and confidence engine**, (4) a **user preference store**, and (5) a **query assembler**. We describe components (2)–(5) using the query of Example 1.

2.1 Atoms and Setup

A user query is decomposed into n NL atoms $x_1, \ldots, x_n$, and the outputs of all active models are parsed into m SQL atoms $a_1, \ldots, a_m$. NL atoms are short meaningful spans from the question, while SQL atoms are clause-level units. To compare outputs from different models, we represent equivalent SQL atoms in the same form before matching them to NL atoms. Each NL atom maps to one or more SQL atoms; the goal is to find the most likely mapping.

For the query *"Show two older students who have two pets."*, we identify the following NL atoms:

x_1 = students, x_2 = older, x_3 = two pets, x_4 = show two

Among these, x_1 is unambiguous in our running example: all active models agree on the corresponding student-related SQL atoms. We therefore focus on x_2–x_4, which capture the main ambiguities.

The focused SQL atoms are:

a_1 = WHERE Age > AVG(Age), a_2 = ORDER BY Age DESC,
a_3 = HAVING COUNT(*)=2, a_4 = HAVING COUNT(petid)>=2,
a_5 = LIMIT 2

These focused atoms correspond to the ambiguous NL atoms:

$$x_2 \to \{a_1, a_2\}, \quad x_3 \to \{a_3, a_4\}, \quad x_4 \to \{a_5\}$$

Here, x_2 ("older") can map to more than one SQL atom, such as a filter (`WHERE Age > AVG(Age)`) or a ranking signal (`ORDER BY Age DESC`). Similarly, x_3 ("two pets") can correspond to either exactly two pets or at least two pets. We treat predicates such as `COUNT(*) = 2` and `COUNT(petid) >= 2` as different atoms, while atoms that differ only in aliases or minor syntactic variation are represented in the same form. The atom x_4 ("show two") maps to `LIMIT 2`.

2.2 Model Agreement

The agreement matrix $M \in \mathbb{R}^{n \times m}$ shows how much the models agree on mappings between NL atoms and SQL atoms. Each row corresponds to an NL atom x_i, and each column corresponds to a SQL atom a_j. We build M by matching each SQL atom to the NL atom it best corresponds to based on clause type and referenced schema items. The entry M_{ij} is the fraction of active models that generate SQL atom a_j for NL atom x_i: $M_{ij} = 1$ means full agreement, and smaller values mean less agreement.

The full matrix is sparse, so we show only the ambiguous rows for x_2 ("older"), x_3 ("two pets"), and x_4 ("show two"), and the focused atoms $a_1, \ldots, a_5$ above.

With three models in our running example, the focused agreement matrix is:

$$M = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & \frac{2}{3} & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{3} \end{bmatrix}$$

for the atom "older," the models split between a filtering interpretation and an ordering interpretation; for "two pets," two models prefer the exact-count interpretation while one prefers the at-least-two interpretation; and for "show two," only one model explicitly produces a `LIMIT 2` atom, while the remaining $\frac{2}{3}$ corresponds conceptually to null (no-atom) outcomes that are not shown in the matrix.

2.3 User Preference and Alignment

For each user, AtomSQL stores a preference matrix U over NL-atom/SQL-atom pairs within the current schema, together with model-level preference counts. With no prior user history, we initialize $U = M$:

$$U = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & \frac{2}{3} & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{3} \end{bmatrix}$$

We combine model agreement and user preference with

$$P_i = \text{softmax}\big(\log M_i + \beta \log U_i\big)$$

where β controls the influence of user preference.

At cold start, $\beta = 0$, so $P_i = \text{softmax}(\log M_i) = M_i$ and the initial alignment follows the cross-model agreement:

$$P = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & \frac{2}{3} & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{3} \end{bmatrix}$$

As user feedback becomes available ($U \neq M$), we increase β, allowing user-specific preferences to influence the alignment.

2.4 Preference Update

When a user selects atom a_j for NL atom x_i, we increment U_{ij} by a step size $\alpha > 0$ and renormalize row i, shifting P toward the user-selected atom in future queries for the same user and schema. Suppose the user selects a_3 (exactly two pets) over a_4 for x_3 (two pets), with step size $\alpha = 0.3$. The corresponding row of U is updated as:

$$[0,\ 0,\ \tfrac{2}{3},\ \tfrac{1}{3},\ 0] \ \rightarrow\ [0,\ 0,\ \tfrac{2}{3} + 0.3,\ \tfrac{1}{3},\ 0] = [0,\ 0,\ 0.967,\ 0.333,\ 0]$$

We then renormalize the row to obtain a probability distribution:

$$[0,\ 0,\ 0.744,\ 0.256,\ 0]$$

The updated U shifts the alignment P toward the user-selected atom in subsequent queries.

In addition to atom-level feedback, we maintain a model preference score $\text{Pref}(m)$ for each model m, which reflects how often a user selects atoms originating from that model. This captures user trust in different models over time.

2.5 Confidence and Query Scoring

For each atom, we define a confidence score $C_{ij} = M^{raw}_{ij} \cdot P_{ij}$, where M^{raw}_{ij} is the unnormalized agreement count (i.e., the fraction of models that produced SQL atom a_j for NL atom x_i before row normalization), and P_{ij} is the final alignment probability. After the user interaction above:

$$C = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0.496 & 0.085 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{3} \end{bmatrix}$$

To select the best SQL output, we compute scores at the query level. While confidence is estimated at the atom level, the final decision is made at the query level by aggregating these signals. The atom-level confidence C_{ij} captures the reliability of individual NL-to-SQL mappings, and these are combined to score each candidate query.

The atom score $S_{\text{atom}}(q)$ measures how well a SQL output q aligns with the user's NL query by averaging alignment probabilities P_{ij} over all atoms in q, effectively summarizing per-atom confidence into a single query-level score. The final score $S_{\text{final}}(q)$ further incorporates model-level preference:

$$S_{\text{atom}}(q) = \frac{1}{|A_q|} \sum_{j \in A_q} \sum_{i=1}^{n} P_{ij}, \quad S_{\text{final}}(q) = S_{\text{atom}}(q) \cdot (1 + \lambda \cdot \text{Pref}(m))$$

where λ controls the influence of model preference and $\text{Pref}(m)$ is the cumulative preference score for model m. The SQL output with the highest S_{final} is shown to the user.

2.6 End-to-End Flow

On each query, the atom decomposer extracts x_i and a_j; the agreement engine builds M from model outputs; the preference store contributes U from past user choices; the two are combined into the final alignment P; and the query assembler picks the SQL output with the highest S_{final}. With no prior history, AtomSQL relies entirely on model agreement. As the user provides feedback, U shifts and future results increasingly reflect their preferences.

Remark. AtomSQL's disambiguation capabilities are limited to the mappings derived from the corresponding SQL queries generated by the NL-to-SQL models under use. If none matches the user's intended interpretation, the user must reformulate the query.

3 DEMONSTRATION SCENARIO

We will demonstrate AtomSQL using the `pets` database from the Spider benchmark, containing three relations (`Student`, `Has_Pet`, and `Pets`). We guide the users through nine steps (annotated in Figure 1) impersonating Nicole, a university housing coordinator preparing a report on pet-friendly housing assignments. She needs to identify older students with exactly two pets to prioritize them for a pet-friendly dorm wing, and asks *"Show two older students who have two pets,"* demonstrating how atom-level confidence exposes ambiguity and how user feedback adapts future results.

Steps Ⓐ & Ⓑ (Selecting models and reviewing schema). In Ⓐ, Nicole selects NL-to-SQL models from the "Admin" panel. From the options of commercial (GPT-4o, Claude 3.5 Sonnet, Gemini 1.5 Pro) and open-source models (Llama 3 8B, DeepSeek-Coder 7B), Nicole chooses SQLCoder 34B (hereafter referred to as SQLCoder), T5SQL, and Llama 3 8B (hereafter referred to as Llama). The panel

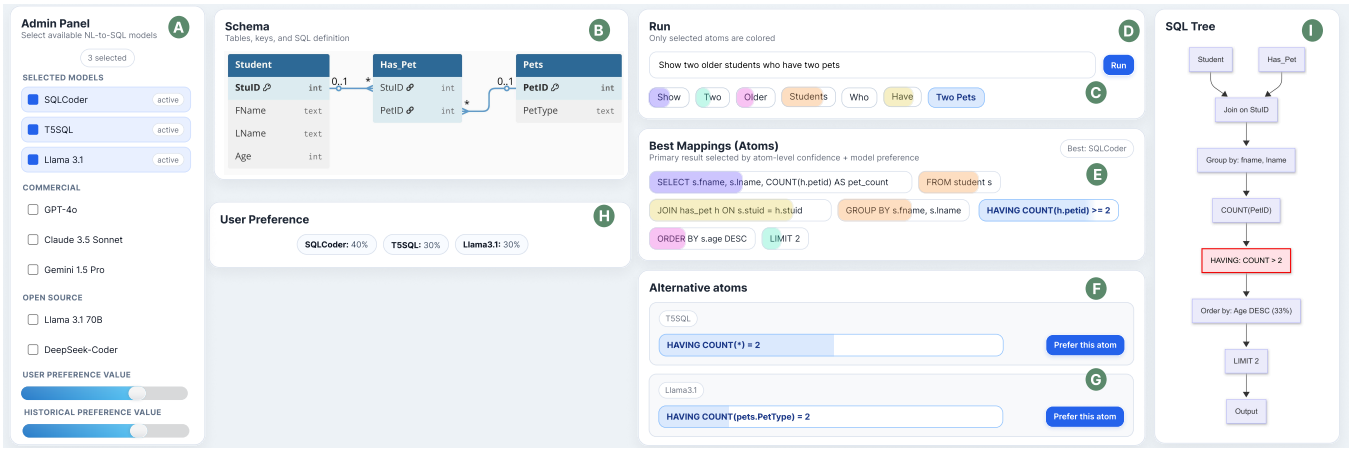

Figure 1: AtomSQL interface: Ⓐ select models, Ⓑ review schema, Ⓒ provide a natural-language query, Ⓓ run the selected models, Ⓔ inspect the best atom mappings, Ⓕ view alternative atoms, Ⓖ choose a preferred atom, Ⓗ update user preferences, and Ⓘ view the SQL tree.

also provides sliders for the *User Preference Value* $\beta_{\text{„}}$ and the *Historical Preference Value* λ. Nicole selects 0.8 for both of them. In Ⓑ, she reviews the "Schema" panel, which shows the database as an entity-relationship diagram with tables, columns, and foreign-key join paths.

Steps Ⓒ & Ⓓ (Entering and running the query). In Ⓒ, Nicole enters "Show two older students who have two pets" in the "Run" panel. In Ⓓ, she clicks *Run*, which dispatches the query to all three active models. AtomSQL splits the query into atoms and the interface shows them (*Show*, *Two*, *Older*, *Students*, *Who*, *Have*, *Two Pets*), highlighting the semantically significant phrases that AtomSQL aligns to SQL.

Step Ⓔ (Inspecting best mappings). In Ⓔ, AtomSQL presents the primary SQL interpretation in the "Best Mappings (Atoms)" panel, with the banner "Best: SQLCoder" because SQLCoder achieves the highest alignment score S_{final} for this query. The assembled query appears as color-coded atoms, where atoms mapped from the same NL phrase share the same color. For example, the atoms `FROM student s` and `GROUP BY s.fname, s.lname` are both mapped from the NL atom "Students".

Steps Ⓕ & Ⓖ (Alternatives and preference selection). In Ⓕ, the "Alternative Atoms" panel shows competing formulations for the `HAVING` atom: T5SQL proposes `HAVING COUNT(*)=2` and Llama proposes `HAVING COUNT(pets.PetType)=2`, each with a "Prefer This Atom" button. In Ⓖ, Nicole selects T5SQL's formulation.

Step Ⓗ (Updating model weights). Nicole's selection immediately updates the "User Preference" panel: the initial weights (SQLCoder 40%, T5SQL 30%, Llama 30%) shift so that T5SQL's share increases and SQLCoder's decreases.

Step Ⓘ (Viewing the SQL tree). In Ⓘ, the "SQL Tree" panel shows the query as a top-down tree, showing `Student` joined with `Has_Pet` on `StuID`, grouped by `fname` and `lname`, aggregated with `COUNT(*)`, filtered by `HAVING COUNT > 2`, ordered by `Age DESC`, and capped with `LIMIT 2`. This lets Nicole verify join paths, aggregation scope, and query structure beyond the flat atom view in Ⓔ.

Beyond the student-and-pets scenario, we will demonstrate AtomSQL on additional databases from the Spider benchmark[11] to illustrate ambiguity across diverse schemas. After the guided demonstration, participants will be able to load their own database schema and issue their own natural-language queries to interact with AtomSQL. The key takeaway from our demo is that users can inspect confidence, compare alternatives, and provide feedback on specific SQL atoms, while AtomSQL adapts to incorporate their preferences in future queries.

ACKNOWLEDGMENTS

This work was supported by the NSF under the Grant CNS-2346555 and a Stena Center for Financial Technology Seed Grant.

REFERENCES

[1] D. Bhattacharjya, B. Ganesan, M. Glass, J. Lee, R. Marinescu, K. Mirylenka, and X. Shou. 2024. Consistency-based Black-box Uncertainty Quantification for Text-to-SQL. In *NeurIPS*.
[2] Z. Ding, Y. Lin, and T. Zeng. 2026. AmbiSQL: Interactive Ambiguity Detection and Resolution for Text-to-SQL. *SIGMOD* (2026).
[3] M. Dong, N. A. Kumar, A. Chauhan, C.-W. Hang, S. Chang, L. Pan, W. Lan, H. Zhu, J. Jiang, P. Ng, and Z. Wang. [n.d.].
[4] A. Elgohary, S. Hosseini, and A. H. Awadallah. 2020. Speak to your Parser: Interactive Text-to-SQL with Natural Language Feedback. In *ACL*.
[5] H. Li, J. Zhang, C. Li, and H. Chen. 2023. RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL. In *AAAI*.
[6] H. Li, J. Zhang, H. Liu, J. Fan, X. Zhang, J. Zhu, R. Wei, H. Pan, C. Li, and H. Chen. 2024. CodeS: Towards Building Open-source Language Models for Text-to-SQL. *SIGMOD* (2024).
[7] L. Mo, A. Lewis, H. Sun, and M. White. 2022. INSPIRED: Towards Transparent Interactive Semantic Parsing via Step-by-Step Correction. In *ACL*.
[8] M. Pourreza and D. Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *NeurIPS*, Vol. 36.
[9] I. Saparina and M. Lapata. 2024. AMBROSIA: A Benchmark for Parsing Ambiguous Questions into Database Queries. In *NeurIPS*.
[10] O. Somov and E. Tutubalina. 2025. Confidence Estimation for Error Detection in Text-to-SQL Systems. In *AAAI*.
[11] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *EMNLP*.
[12] F. Zhao, S. Deep, F. Psallidas, A. Floratou, D. Agrawal, and A. E. Abbadi. 2024. Sphinteract: Resolving Ambiguities in NL2SQL through User Interaction. *Proceedings of the VLDB Endowment* 18, 4 (2024), 1145–1158.