Summary

This thesis addresses a persistent and underexplored challenge in Natural Language Interfaces to Databases (NLIDB): schema-induced ambiguity (SIA). While existing NL2SQL systems have achieved high benchmark performance with the help of large language models (LLMs) and schema-linking techniques, they remain vulnerable to failure in real-world applications, where database schemas often contain overlapping elements and inconsistent naming. These conditions give rise to SIA—ambiguity that occurs when parts of natural language plausibly refer to multiple schema elements (tables, columns, joins, or precomputed aggregates). Unlike lexical or linguistic ambiguity, SIA arises from the interaction between the user's question and the structure of the underlying schema. To formalize this, we adopt a taxonomy of four common SIA types: *column ambiguity* (e.g., synonyms or polysemy like "capacity" mapping to either seating or standing capacity), *table ambiguity* (overlapping semantic roles between tables), *join ambiguity* (unclear join paths in vertically partitioned schemas), and *precomputed aggregate ambiguity* (e.g., "total sales" referring to either an aggregated value or a static column).

To address this issue, we introduce **GATekeeper**, a schema-aware system designed to proactively detect SIA before query generation. GATekeeper consists of four distinct components: (1) a node relevance predictor using a fine-tuned BERT cross-encoder, (2) a contextual embedder to construct rich representations of schema elements in relation to the question, (3) a representation module that builds a graph from these elements using schema structure, and (4) a graph-based classifier that outputs a binary ambiguity prediction. The graph encodes tables and columns as nodes, and their relationships as edges. Each node's features are derived from BERT-encoded embeddings, capturing interactions between the NL question and schema elements. The final output indicates whether the input question exhibits SIA.

We evaluate GATekeeper on multiple datasets, including the AmbiQT-annotated Spider dataset, BIRD-Bench, and TrialBench. In in-domain settings, GATekeeper outperforms a GPT-4.1-based zero-shot baseline by a substantial margin, with an F1 score of 0.753 versus 0.552. Ablation studies show that while graph structure contributes to performance, the model's success primarily stems from the contextual embeddings generated by the BERT cross-encoder. Even when schema edges are randomized, GATekeeper retains high performance, suggesting that most of the semantic signal is captured in the embeddings. When these embeddings are removed, model performance drops significantly.

GATekeeper struggles with generalisation to unseen schemas. In cross-domain evaluation on BIRD-Bench, the model over-predicts ambiguity, flagging nearly all queries as ambiguous. However, few-shot adaptation using as few as 12 in-domain examples significantly improves performance, outperforming GPT-4.1. This result suggests that while GATekeeper's inductive bias is brittle, it is also adaptable. Integrating ambiguity detection into a full NL2SQL pipeline further demonstrates its value: execution accuracy improved from 14.7% to 22.1% by preemptively filtering ambiguous queries.

GATekeeper shows that explicit, schema-aware ambiguity detection is both feasible and valuable. Its architecture offers a foundation for more robust NL2SQL systems capable of handling the messy reality of production databases. While in-domain performance is strong, broader deployment will require training on more diverse schemas and principled approaches to few-shot adaptation. This thesis lays the groundwork for future work in ambiguity resolution, potentially involving user-guided clarification or hybrid systems that combine detection with interactive disambiguation.

This page is intended as a standalone summary. The main paper begins on the next page.

GATekeeper: Detecting Schema-Induced Ambiguity in Natural Language Interfaces to Databases

Jacob Skadborg
jskadb20@studen.aau.dk
Department of Computer Science, Aalborg University
Aalborg, Denmark

Martin Mortensen
mksm20@student.aau.dk
Department of Computer Science, Aalborg University
Aalborg, Denmark

Abstract

Natural Language to SQL (NL2SQL) systems translate natural language questions into executable SQL queries, enabling non-technical users to interact with databases. While recent advances in large language models (LLMs) and schema-aware techniques have driven performance on benchmarks such as Spider and BIRD, existing systems continue to struggle with ambiguity—particularly when queries admit multiple valid interpretations due to overlapping schema elements. This issue, termed Schema-Induced Ambiguity (SIA), arises when natural language tokens ambiguously refer to multiple tables, columns, or relations. SIA is especially common in real-world databases, where evolving and denormalised schemas diverge from the clean structure typically found in academic benchmarks.

Current approaches address ambiguity only implicitly or partially. LLMs can reduce lexical ambiguity, but fail to reliably detect structural ambiguities without explicit schema reasoning. Moreover, few systems are designed to proactively identify SIA before generating a query, leading to silent failures and misinterpretations. To address this gap, we propose a two-step detection framework: a fine-tuned BERT cross-encoder identifies schema elements likely to be involved in the intended query, followed by a Graph Attention Network (GAT) operating over the induced subgraph to predict the presence of ambiguity. Our method outperforms baseline approaches in-domain, yet generalisation to unseen schemas remains limited, as evidenced by performance drops on BIRD-bench and Trial-Bench. Nonetheless, in-context training demonstrates strong potential for scaling ambiguity detection. While this work focuses exclusively on schema-induced sources, future extensions must address other forms of ambiguity to ensure reliability in production deployments. Code available at: https://github.com/P10-NLIDB.

1 Introduction

Natural Language (NL) to SQL (NL2SQL) systems aim to convert users' natural language questions into SQL queries, thus enabling intuitive database interactions for non-technical users. Recent advances in large language models (LLMs), including GPT-4, Llama, and CodeT5, along with sophisticated schema-linking methods, have substantially enhanced NL2SQL accuracy and robustness, achieving impressive benchmark results on datasets like **Spider** [28], **BIRD** [11], and **WikiSQL** [31] [10, 14, 29, 32].

Modern NL2SQL solutions leverage LLMs via either in-context prompting [15] or supervised fine-tuning [10], supplemented by schema linking improvements [23], structured decoding [20], and ensemble or agent-based approaches [8]. Techniques such as retrieval-augmented generation (RAG) [26], conversational dialogue frameworks [27], and grammar-constrained decoding [12] have led to impressive results on benchmarks like **Spider** and **BIRD**.

Despite these advances, existing NL2SQL systems struggle with ambiguity—particularly *schema-induced ambiguity* (SIA), where parts of a question correspond to multiple plausible schema elements. This problem is common in real-world databases, which are often denormalised, loosely structured, or extended over time, introducing overlapping column names, redundant joins, and inconsistent naming conventions. Benchmark datasets, by contrast, typically feature clean and well-normalised schemas, under-representing such ambiguity and obscuring system limitations.

Ambiguity remains one of the most pressing challenges in NL2SQL [7, 10, 14]. It arises from the inherent imprecision of NL, contrasted with the strict semantic requirements of SQL. SIA is particularly problematic because current systems often guess mappings without explicitly identifying the ambiguity. This can lead to hallucinations, misinterpretations, or logically incorrect queries—errors that are hard to detect and explain post-hoc.

Current NL2SQL systems offer partial solutions for detecting SIA. LLMs effectively reduce lexical ambiguities by recognising paraphrases and uncommon phrasing, frequently mapping diverse user expressions to common intents. However, deeper schema-related ambiguities, often remain undetected due to insufficient contextual understanding, lack of schema-awareness, or inherent complexities in the schema.

Existing systems fall short in two critical ways. First, many implicitly guess the correct interpretation without flagging ambiguity, leading to silent errors that are difficult to trace or correct. Second, while some approaches incorporate schema structure or mapping heuristics, they rarely include explicit ambiguity detection mechanisms. As a result, ambiguity is often handled reactively—if at all—and only after incorrect queries are generated.

There is a clear need for methods that proactively identify SIA before query generation. As noted by Floratou et al. [7], ambiguity remains a fundamental and under-addressed limitation in current NL2SQL systems. Our work directly targets this gap.

Our method named GATekeeper focuses primarily on the detection of SIA in NL questions. This is achieved through a two-step process. First, we identify the most likely schema elements involved in the corresponding SQL query using a fine-tuned BERT cross-encoder. Next, we construct a graph consisting of these schema elements, embedding each with the final hidden state from the cross-encoder. This graph is then processed by a fine-tuned Graph Attention Network (GAT), which outputs a score in the range [0,1],

where values closer to 1 indicate higher ambiguity and values near 0 suggest the question is unambiguous.

Our results show that the model performs well in-domain, i.e., when applied to database schemas seen during training, outperforming state-of-the-art baselines. However, its generalisation to unseen schemas is limited. On manually annotated benchmarks such as BIRD-bench and Trial-Bench, performance drops significantly. In particular, on BIRD-bench, the model lags well behind GPT-4.1, highlighting a key weakness in schema generalisation.

Training with in-context examples demonstrates the model's strong potential to detect ambiguous queries, offering a promising direction for mitigating ambiguity in NL2SQL systems. However, this only addresses SIA. Questions that are ambiguous due to other factors, such as linguistic underspecification or semantic vagueness unrelated to the schema, remain out of scope. Future work will need to extend ambiguity detection beyond SIA sources to improve robustness in real-world applications.

2 Related Work

In this section, we review prior research on ambiguity in NL2SQL, structured around two core areas: (1) ambiguity arising from factors outside the schema (e.g., linguistic ambiguity) and (2) established ambiguity detection and resolution methods.

2.1 Linguistic Ambiguity in Natural Language Questions

In contrast to SIA, many ambiguities arise from properties of NL itself, independent of any particular database schema. Such linguistic ambiguities include phenomena like lexical polysemy, pronoun coreference, syntactic ambiguity, and underspecified context. For example, a user question may use an ambiguous word (e.g., "bank" as a financial institution vs. river bank) or omit crucial context ("Who is the president?" without specifying a country or organisation), leading to multiple interpretations even if the database schema is perfectly clear. These forms of ambiguity have been widely recognised in the broader NL interface and question-answering literature. Alharbi et al. [1], for instance, identify lexical semantic ambiguity in NL questions as a key challenge for question answering systems, noting that words with multiple meanings can confuse automated interpretation. Their work proposes an ontology-driven approach (the CKCO framework) to resolve such ambiguities by mapping terms to domain concepts, illustrating a taxonomy of ambiguity types focused on word-sense and intent ambiguities in NL questions. More generally, Floratou et al. [7] emphasise that many NL2SQL errors originate from linguistic underspecification—the question itself is incomplete or too imprecise, irrespective of schema. Recent surveys and studies of text-to-SQL with LLMs concur that handling these non-schema ambiguities remains an open problem [10, 14]. Outside of the database domain, researchers have developed taxonomies and methods to detect when a user's query is ambiguous and requires clarification. For example, Zhang and Choi [30] focus on identifying ambiguous user intents in NL questions, proposing to clarify when necessary by automatically detecting uncertainty in the query's intent and prompting for clarification. Such work treats ambiguity as a property of the question (e.g. an NL question having multiple possible answers or interpretations) rather than of the schema, and it complements schema-centric taxonomies by addressing linguistic sources of confusion. In summary, ambiguity taxonomies beyond the schema tend to classify types of language ambiguity (lexical, semantic, contextual, etc.) and have led to techniques for disambiguation that leverage context or interactive clarification.

2.2 Ambiguity Detection Methodologies

A range of methodologies have been proposed to detect or mitigate ambiguity in NL2SQL systems, differing in how they conceptualise ambiguity and whether they operate proactively (flagging ambiguity before or during query generation), reactively (handling ambiguity post hoc), or through user interaction. Broadly, these approaches fall into three main families: probabilistic uncertainty estimation, ontology-based disambiguation, and schema-aware structural modeling.

One prominent line of work leverages uncertainty estimation to detect ambiguity by identifying low-confidence schema alignments. Zhang et al.[30] use entropy-based metrics to quantify uncertainty. This probabilistic method is effective at flagging low-confidence mappings, especially for linguistic or intent ambiguity, but it is made for general question answering and therefore lacks grounding in database semantics.

Ontology-based approaches incorporate domain-specific mappings between words and schema elements. A representative example is the CKCO framework [1], which uses a curated ontology to map lexical items in user questions to database concepts. These methods improve ambiguity detection by resolving polysemy and synonymy in domain-specific contexts. However, they depend heavily on manually engineered ontologies, which limits scalability across schemas or domains. This trade-off between precision and generalisability remains a core limitation of ontology-based strategies.

A third class of methods targets structural modeling, where ambiguity is addressed through schema-aware neural architectures. For instance, RASAT [17] introduces relational attention mechanisms into sequence-to-sequence parsers, allowing the model to attend to table and column relationships during decoding. Similarly, RAT-SQL [23] enhances text-to-SQL parsing with schema linking via relation-aware transformers. These models can disambiguate between competing schema elements more effectively than schema-agnostic baselines, but they do so implicitly—by improving alignment and decoding quality—rather than targeting ambiguity detection as a standalone objective. As such, they may resolve some schema-induced ambiguity as a byproduct of better modeling but still fail to explicitly flag ambiguous cases when schema alignment fails.

A further methodological distinction is whether user interaction is involved in the resolution process. Interactive systems, such as AmbiQT [3], surface top-k candidate SQL queries and prompt the user to choose the correct interpretation, transforming ambiguity resolution into a human-in-the-loop task. While effective, this approach assumes user availability and introduces latency. In contrast, non-interactive approaches seek to identify and handle ambiguity autonomously. While relatively few systems focus explicitly on this,

recent benchmarks such as AmbiQT and AMBROSIA [18] have enabled new research in this space. These benchmarks annotate NL questions with known ambiguities—schema-induced in the case of AmbiQT, and linguistic or syntactic in AMBROSIA—and serve as training and evaluation grounds for ambiguity detectors.

Importantly, these benchmarks do not themselves provide detection models, but their annotations have catalysed the development of models focused specifically on ambiguity detection. However, robust schema-aware ambiguity detection, particularly in a proactive, automatic fashion, remains an emerging area. Most current solutions still handle ambiguity reactively or indirectly, and few models are designed to detect ambiguity as a first-class task. This leaves the field open for systems that explicitly identify schema-induced ambiguity prior to SQL generation, offering a critical foundation for reliable, ambiguity-aware database querying. Refer to Table 1 for a final overview of methods discussed and relevant to ambiguity detection.

2.3 Summary

Although prior research has investigated various strategies for ambiguity resolution and interactive clarification in NL2SQL systems, explicit detection of ambiguity—particularly schema-induced ambiguity—as a standalone capability remains underexplored. While many existing models improve schema alignment implicitly, they typically do not identify or address ambiguity directly. This gap motivates our work: we focus on the proactive detection of SIA, aiming to build systems that not only generate SQL queries but also recognise when a question is inherently ambiguous with respect to the schema. By making ambiguity explicit, our approach lays a critical foundation for more robust disambiguation strategies and improved query accuracy, paving the way for robust and more reliable NLIDB.

3 Preliminaries

This section introduces the core concept underpinning our work, *schema-induced ambiguity* (SIA) in NL2SQL systems. Understanding the nature of these ambiguities is essential, as they form the basis for our annotation strategy and model architecture.

3.1 Schema-Induced Ambiguity

Ambiguity in NL2SQL queries arises predominantly due to interactions between NL questions and database schema design decisions. This means that even though the question it self does not seem linguistically ambiguous, multiple interpretations of how to answer the question can occur when considering a specific schema. Bhaskar et al. [3] present a detailed taxonomy categorising SIA into four distinct classes. We present the classes, and figure 1 illustrates the problems:

- **Column Ambiguity (C)**: Lexical ambiguity arising from synonyms or polysemous column names (e.g., *"capacity"* mapping to `standing_capacity` or `seating_capacity`).
- **Table Ambiguity (T)**: Semantic overlaps among different tables representing similar concepts (e.g., `Artist` vs. `Performer`).
- **Join Ambiguity (J)**: Ambiguity caused by vertically partitioned tables, where necessary joins to retrieve complete information are unclear (e.g., `Employee` and `Employee_details`).
- **Precomputed Aggregate (P)**: Ambiguity due to underspecified aggregation requirements (e.g., *"total sales"* could imply a precomputed column or the aggregation `SUM()`).

We adopt this taxonomy and coin it schema induced ambiguities (SIA). This issue is often accelerated in production databases, which, over time, have deviated from their original, well-defined normalised form.

3.2 Problem Formulation

The input to a NL2SQL system consists of a NL question Q and a database schema D, and the system's goal is to predict the corresponding SQL query that answers Q.

We define the database schema as a structured set $D = T \cup C$ representing schema components of a target database which contains a set of tables $T = \{t_1, t_2, \ldots, t_{|T|}\}$, where each table t_i is associated with a set of columns $C_i = \{c_{i,1}, c_{i,2}, \ldots, c_{i,|C_i|}\}, \forall i = 1, 2, \ldots, |T|$, such that $C = \bigcup_{i=1}^{|T|} C_i$. We refer to a *single schema element* (either a table or a column) as $d \in D$. We define $PK = \{k_1, \ldots, k_{|K|}\}$ where $k_i \subseteq C_i$ as the set of primary keys, where each k_i defines the primary key of table t_i, and $FK = \{f_1, \ldots, f_{|F|}\}$ as the set of foreign keys, where $f_k = (c_{i,s}, c_{j,p})$ indicates that column $c_{i,s}$ of the table t_i references the column $c_{j,p}$ of the table t_j.

A key challenge in this setting is SIA, as described in 3.1. The input to solve this issue is the same— Q and D, and the task is to accurately detect SIA without requiring full SQL generation—enabling systems to flag or handle ambiguous questions early in the pipeline. We formulate this as a binary classification problem: given a question-schema pair (Q, D), determine whether Q exhibits schema item ambiguity with respect to D. Specifically, we aim to define a function

$$f : (Q, D) \mapsto \{0, 1\} \tag{1}$$

where $f(Q, D) = 1$ if Q contains at least one instance of SIA, and $f(Q, D) = 0$ otherwise.

4 Methodology

Our objective is to determine whether a NL question posed over a relational database schema is ambiguous, in the specific sense of admitting multiple valid interpretations due to SIA. To this end, we design a modular system, called GATekeeper, composed of distinct components, each responsible for a targeted aspect of the overall task.

The system has the following modules:

Node Relevance Prediction: A Predictor module evaluates every schema element $d \in T \cup C$ with respect to Q, assigning a relevance score $P(d) \in [0, 1]$, which is the output. This score reflects the likelihood that the element is involved in answering the question.

Node Embedding: The Embedder constructs contextual embeddings for each schema element. Each embedding x_d captures the interaction between Q and the element d, incorporating both lexical and relational information. This embedding is used as the features

Table 1: Features of existing NL2SQL benchmarks are categorised into measures and scope, with each benchmark marked as either including (+) or excluding (-) a given feature

Method	Domain	Detection Technique	Explicit / Implicit	Schema-aware?	Resolution vs. Detection	Type of Ambiguity Handled
AmbiQT[3]	NL2SQL	Ambiguity detection module	Explicit	**No**	Both (Detection & Resolution)	Lexical & semantic ambiguities
RAT-SQL[23]	NL2SQL	Relation-aware self-attention	Implicit	**Yes**	Resolution	Schema linking errors
RASAT[17]	NL2SQL	Schema-guided disambiguation via attention mechanisms	Implicit	**Yes**	Resolution	Structural ambiguities
CKCO[1]	NLQA	Contextual keyword and concept matching	Explicit	**No**	Resolution	Intent ambiguities
Intent-Sim[30]	NLQA	Similarity-based intent recognition	Explicit	**No**	Resolution	Ambiguous user intents
FANDA [13]	NL2SQL (Multi-turn)	Context-aware detection using dialogue history	Explicit	**No**	Both (Detection & Resolution)	Context-dependent follow-up ambiguities
XiYan-SQL [8]	NL2SQL	Multi-generator ensemble with in-context learning and supervised fine-tuning	Implicit	**Yes**	Resolution	Semantic, Structural, Syntactical ambiguities
CHASE-SQL [16]	NL2SQL	Multi-path reasoning	Implicit	**Yes**	Resolution	Semantic, Structural, Logical ambiguities

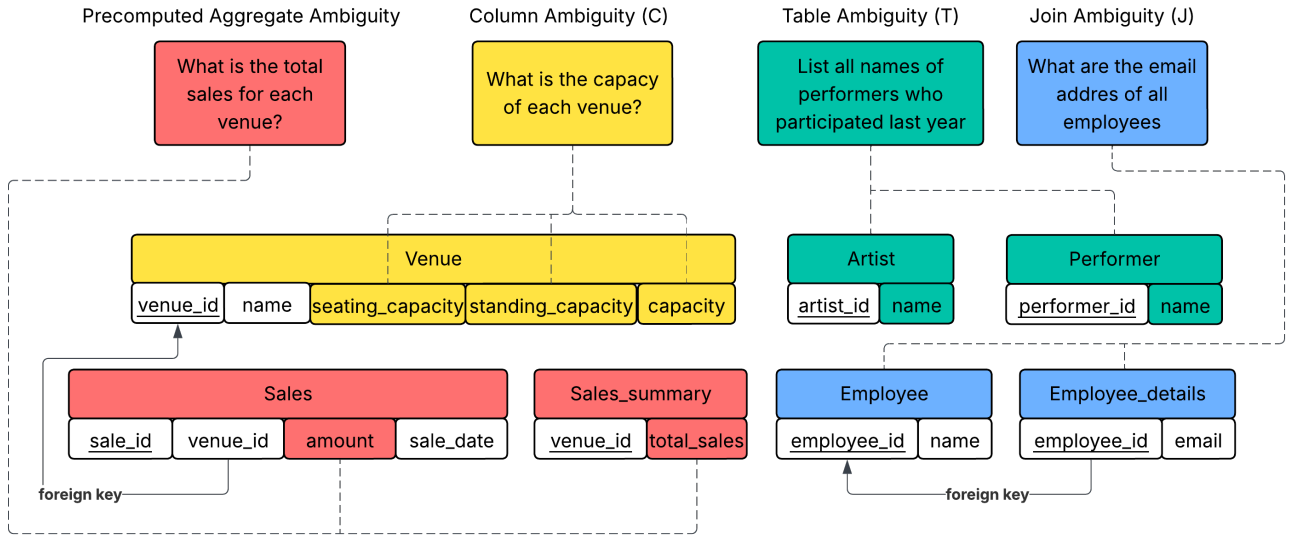

Figure 1: An example of how a schema design can create SIA for each type of SIA

of each node during classification. The output is the embedding of each schema element $\{e_d\}_{d \in D}$.

Representation Construction: This module encodes schema relationships (e.g., belongs-to, foreign key, primary key) and serves as the structural context for further reasoning. Schema elements are nodes and schema relationships are edges. The output is a graph $\mathcal{G}$.

Ambiguity Classification: The Ambiguity Classifier takes as input the graph $\mathcal{G}$ along with node embeddings x, and uses a graph based classifier to assess the global coherence of the question with the selected schema elements. The classifier outputs a binary label—YES if the question is ambiguous, NO otherwise.

Each module operates independently but interfaces with the others through shared representations (e.g., node scores, embeddings, and structural graphs). This modular design enables precise control over each stage and facilitates targeted evaluation and ablation.

Figure 2 provides a high-level view of this modular structure. The subsequent subsections describe the individual modules and their interactions in more detail.

4.1 Preprocessing

The initial input comprises a NL question Q and the associated database schema D (i.e the database that the question is being asked for). We preprocess both the Q and D to normalise their representations before constructing the interaction graphs. This ensures consistency and facilitates accurate alignment between linguistic and structural elements.

For D, we perform text normalisation - the tables and columns are tokenized, lemmatized and lowercased using the Stanza NLP toolkit, this ensures consistent schema representations and reduces lexical variance during matching. Furthermore, relations among tables (foreign key and primary key links) and columns (same-table relations and foreign key columns) are computed and stored in a relational matrix. These steps yield a processed schema with standardised names and a matrix of schema relations.

For Q, all quotation symbols are standardised to reduce tokenisation variability; for instance, *What are the first name and last name of all “candidates”?* becomes *What are the first name and last name of all "candidates"?* before further processing. The question is then tokenised, lemmatised, and lowercased; for example, *candidates* becomes *candidate*, while the sentence is split into individual tokens such as *["what", "are", "the", "first", "name", "and", "last", "name", "of", "all", """, "candidate", """, "?"]*.

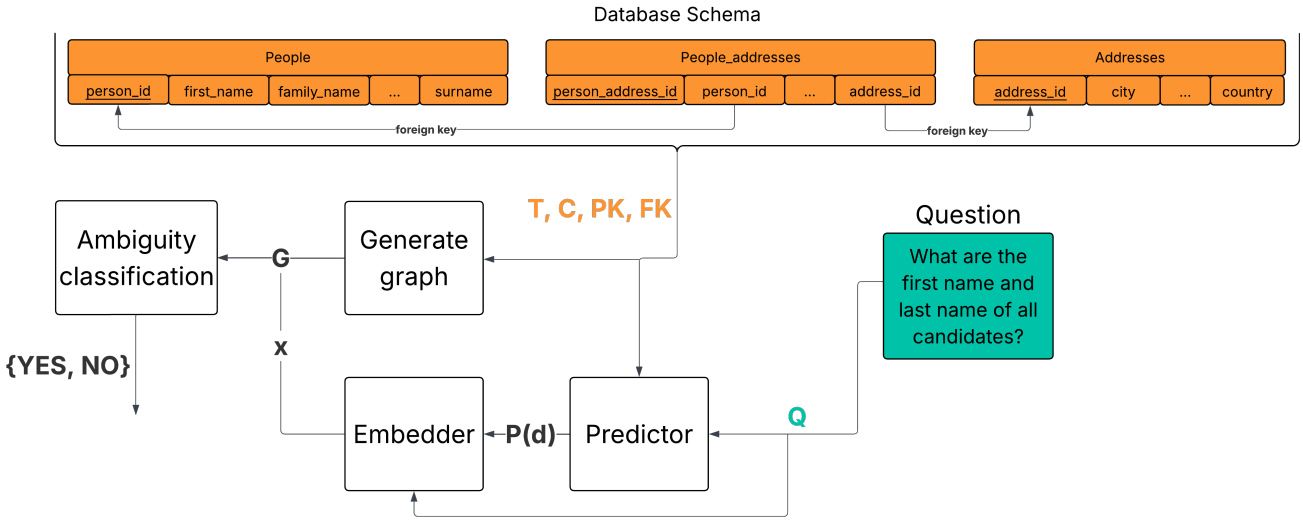

Figure 2: Overview of the ambiguity detection method. T, C denote tables and columns in the database schema. PK, FK denotes primary and foreign key relations. Given a question Q, the Predictor scores each schema element. A graph $\mathcal{G}$ is generated and elements are embedded. The ambiguity classifier uses both the graph structure and embeddings to make a final binary prediction.

4.2 Node Relevance Prediction

This part of the method is responsible for identifying which tables and columns are likely involved in any SQL generated to answer an incoming NL question. A prediction of relevant tables and columns serves two critical purposes: (1) it constrains the search space for query generation by focusing on schema elements pertinent to the question, and (2) it provides essential signals for downstream modules to disambiguate references and construct semantically accurate queries.

To train a model to predict table and column relevance, we first construct a training dataset. For each question-query pair in a NL2SQL dataset (e.g., Spider or BIRD), we generate positive and negative examples. The set of positive examples for a question Q, denoted as $Pos(Q)$, consists of the tables and columns referenced in the gold SQL query corresponding to Q. Specifically, the parser extracts:

- a list of tables that are referenced in the gold query, and
- a dictionary where each key is a table name and the corresponding value is the set of columns from that table that appear in the gold query.

The set of negative examples, denoted as $Neg(Q)$, is constructed by randomly sampling schema elements that do not appear in the gold SQL query. To ensure that the training set remains balanced and does not bias the model toward predicting irrelevance, we sample an equal number of negative and positive examples for each question-query pair.

During training, we assign a binary label to each (question, schema element) pair based on membership in $Pos(Q)$ or $Neg(Q)$:

$$r(Q,d) = \begin{cases} 1 & \text{if } d \in Pos(Q) \\ 0 & \text{if } d \in Neg(Q) \end{cases} \tag{2}$$

Each (question, schema) pair is fed into a BERT cross-encoder as:

$$[CLS]Q[SEP]d[SEP]$$

CLS is a special token placed at the very beginning of every sequence, where SEP indicates seperators between the different segments of the sequence—in this example, the separator between the natural language question, and the schema elements that are being scored. The BERT cross encoder uses a small regression head that maps the $[CLS]$ token output to a scalar score, between 0 and 1. At inference time, we run the question against every candidate element, apply a sigmoid to the raw logits (vector of raw non-normalised predictions) to obtain scores in $[0, 1]$ as seen in Figure 3.

The output of the cross-encoder BERT model is a prediction of the relevance of each schema element given Q. A threshold $\theta \in [0, 1]$ can be applied on the predicted score to determine whether a schema element is considered relevant.

4.3 Node Embedding

Once the relevant schema elements have been predicted through the previous step, we construct contextualized embeddings for each element. These embeddings, denoted as $e_d \in \Re^H$, capture both the lexical semantics and the predicted relevance of each schema element $d \in D$ with respect to Q.

Each embedding e_d is derived by encoding a triplet consisting of the question Q, the schema element d, and a relevance indicator

$R_d \in \{0, 1\}$, a value of 1 indicates that the schema element is relevant to answering the question, while 0 denotes that it is irrelevant. The input sequence is formatted as:

$$[CLS]Q[SEP]d[SEP]R_d[SEP]$$

and passed through a BERT-based encoder. The final hidden state of the `[CLS]` token is extracted as the embedding:

$$e_d = BERT_{CLS}(Q, d, R_d)$$

These embeddings $\{e_d\}_{d \in D}$ serve as the contextualised representations of schema elements and will be used as node features in subsequent components of the model.

4.4 Graph Construction

We build a directed graph $\mathcal{G} = (V, E)$ to represent the structural and semantic relationships within the database schema, which serves as the foundation for graph-based reasoning in the final classification step.

The set of nodes V corresponds to all schema elements:

$$V = T \cup C = \{t_1, t_2 \dots t_{|T|}\} \cup \bigcup_{i=1}^{|T|} C_i \tag{3}$$

That is, each table $t_i \in T$ and each column $c_{i,j} \in C_i$ for all $i \in [1, |T|]$ are represented as nodes in the graph. The set of directed edges E encodes three types of relational structures present in the schema:

$$E = E_{belongs} \cup E_{pk} \cup E_{fk} \tag{4}$$

where:

- $E_{belongs} = \{(c_{ij}, t_i) | c_{ij} \in C_i\}$ encodes *belongs-to* relationships between columns and the tables they reside in.
- $E_{PK} = \{(t_i, k_i) | k_i \in C_i$ is the primary key of $t_i\}$ encodes *primary key* relationships from each table to its designated primary key column(s), as specified in the set PK.
- $E_{FK} = (c_a, c_b) | (c_a, c_b) \in F$ encodes *foreign key* relationships where column c_a references column c_b in another (or the same) table, as defined in the set FK.

Each node in $\mathcal{G}$ is initialised with its generated corresponding contextual embedding, capturing both its relevance to the input question and its schema semantics. The resulting graph structure allows a GNN to reason jointly over question-aware node features and schema connectivity, enabling effective detection of schema item ambiguity in natural language questions.

4.5 Graph-based Classification

We frame schema ambiguity detection as a graph-level classification task and employ a multi-layer Graph Attention Network (GAT) to propagate and aggregate information over the constructed schema graph [22]. Standard Graph Neural Networks (GNNs) operate by iteratively updating each node's embedding based on a fixed aggregation of its neighbours' representations. GAT extends this by incorporating learnable attention weights, which dynamically modulate the influence of neighbouring nodes during message passing.

In GAT, each node v updates its representation at layer l by attending to its neighbours $\mathcal{N}(v)$. The updated embedding is computed as:

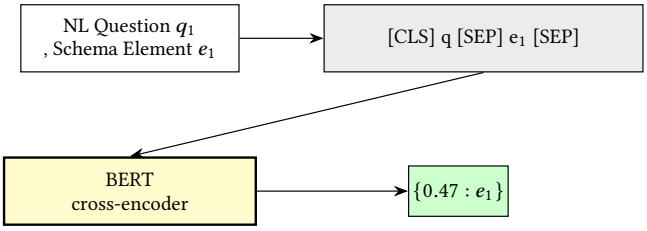

Figure 3: BERT cross-encoder used to weigh (predict) schema elements; - every schema element in the database is weighed against the NL question

$$h_v^{(l)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{vu}^{(l)} W^{(l)} h_u^{(l-1)}\right) \tag{5}$$

where $h_v^{(0)} = e_v$ is the initial contextualised embedding of node v, $W^{(l)}$ is a learnable projection matrix, σ is a non-linear activation function, and $\alpha_{vu}^{(l)}$ is the attention coefficient between node v and its neighbour u, defined as:

$$\alpha_{vu}^{(l)} = \frac{\exp(\text{LeakyReLU}(a^\top [W^{(l)} h_v^{(l-1)} \,\|\, W^{(l)} h_u^{(l-1)}]))}{\sum_{k \in \mathcal{N}(v)} \exp(\text{LeakyReLU}(a^\top [W^{(l)} h_v^{(l-1)} \,\|\, W^{(l)} h_k^{(l-1)}]))} \tag{6}$$

Here, a is a learnable weight vector and $\|$ denotes vector concatenation. This mechanism allows the model to assign different importance to each neighbouring node.

To improve expressiveness and stabilise learning, GAT uses *multi-head attention* [5], where K independent attention mechanisms are applied in parallel and their outputs are either concatenated or averaged.

After L GAT layers, we obtain the final node embeddings $\{h_v^{(L)}\}_{v \in V}$, which encode both structural and contextual relevance, weighted by attention. These node embeddings are aggregated via mean pooling to produce a graph-level representation:

$$z_Q = \frac{1}{|V|} \sum_{v \in V} h_v^{(L)} \tag{7}$$

The pooled vector z_Q serves as a summarised representation of the entire schema graph. It is computed as the mean of the final node embeddings, each of which integrates attention-weighted information from its neighbours. As such, nodes that receive greater attention contribute more prominently to z_Q, allowing the model to focus on the most relevant schema elements when predicting ambiguity. This pooled vector is passed through a feed-forward classification layer to obtain the ambiguity score:

$$f(Q, D) = \sigma(W z_Q + b) \tag{8}$$

A final binary prediction is made using a threshold θ:

$$\hat{y} = \begin{cases} 1 & \text{if } f(Q, D) \geq \theta \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

5 Experiments

To evaluate the effectiveness of our SIA detection model GATekeeper, we design experiments that address model accuracy, model ablations, domain generalisation, training diversity, and downstream impact on NL2SQL execution. These experiments are motivated by distinct research questions as seen in table 2.

Table 2: Index of experiments conducted in this study, including core research questions and references to relevant sections and tables.

Experiment	Research Question
Baseline Experiment	Is GATekeeper able to classify SIA?
Impact of Graph Structure	How much does GATekeeper's performance depend on the structure of the input graph, as opposed to its node features?
Role of Contextual Node Embeddings	What is the impact on ambiguity detection performance when replacing rich contextual node embeddings with simplified scalar relevance scores?
Comparison: GAT vs. Standard GNN	Does GAT attention lead to better ambiguity detection performance than uniform aggregation in standard GNNs?
Effect of Training Domain Diversity	Does training on data from multiple domains improve performance on in-domain ambiguity detection?
Generalisation Across Domains	How well does GATekeeper generalise to unseen domains or schemas, and can few-shot adaptation improve its performance?
Downstream Impact on NL2SQL Execution	Does incorporating ambiguity detection into a full NL2SQL pipeline improve execution accuracy by reducing query failures?

5.1 Datasets and Annotation

We utilize four complementary datasets, selected to capture diverse aspects of SIA in NL2SQL tasks. Together, these datasets span synthetic, controlled, and real-world ambiguity scenarios, enabling robust model training and thorough evaluation across varying schema complexities and ambiguity types.

The **AmbiQT-flagged Spider dataset** [3] serves as our primary pretraining resource. It is derived from the original Spider benchmark, a widely used large-scale NL2SQL dataset featuring complex queries over a diverse set of relational schemas. While Spider was not designed with ambiguity in mind, the AmbiQT methodology retroactively annotates its questions with weak ambiguity labels, enabling the study of SIA at scale. Although these labels are not manually verified, they provide sufficient supervision for large-scale training, thanks to the richness and variety of the underlying schemas.

The **BIRD-Bench** dataset is a modified version of the BIRD[11] benchmark, curated to introduce ambiguity in a controlled and reproducible manner. We manually inject ambiguity by modifying NL questions to include lexically or semantically overloaded terms, and by altering database schemas to introduce overlapping table and column names—following the SIA definitions.

The **AMBROSIA** dataset is a curated benchmark designed to evaluate NL2SQL systems under linguistically ambiguous input conditions. AMBROSIA focuses on ambiguity arising solely from NL interpretation, independent of schema design, meaning this is not SIA. Following the taxonomy introduced by Saparina et al. [18], AMBROSIA introduces three distinct ambiguity types: *scope ambiguity*, *attachment ambiguity*, and *vagueness*. These ambiguities are injected by modifying NL questions to contain quantifier scoping conflicts, syntactic attachment uncertainty, or context-dependent lexical imprecision. Each ambiguous question is paired with multiple valid interpretations and associated SQL queries.

The **TrialBench** dataset consists of real-world clinical trial queries drawn from enterprise applications at Novo Nordisk. TrialBench reflects authentic schema complexity, including features such as denormalisation, inconsistent naming conventions, and a schema that has evolved over time.

To supervise GATekeeper, we employ a structured, taxonomy-driven annotation scheme. Every NL question is labeled as either ambiguous or unambiguous with respect to its schema.

Each dataset is split into training, validation, and test sets using an 80/10/10 split with same distribution of labels to maintain the balance between ambiguous and unambiguous examples across splits. This ensures consistency and comparability in downstream evaluation. AmbiQT provides both training and testing sets, so we use those.

5.2 Experimental Setup

All models are implemented in PyTorch, with HuggingFace Transformers used for BERT components and PyTorch Geometric for GAT implementation [2, 24].

We train all ambiguity classifiers using binary cross-entropy loss, with each NL question labeled as either ambiguous or unambiguous. For ambiguous questions, additional fine-grained labels per the SIA taxonomy are included but not directly supervised.

All models are trained using the AmbiQT training split. We perform a parameter sweep over learning rates 1e-2, 1e-3, 1e-4, hidden dimensions 64, 128, 256, GAT attention heads 2, 4, 8, and GAT layers 2, 3, 4 using the validation F1 score as the primary selection criterion. Early stopping with a patience of 40 epochs is applied to prevent overfitting. All models are evaluated on the AmbiQT validation and test splits, as well as on the out-of-distribution BIRD (annotated).

We report precision, recall, and F1 score for all experiments. Given the tradeoff between recall and precision in ambiguity detection, we use macro F1 as the primary evaluation metric and rank models by their best F1 score on the validation set. High recall ensures that most truly ambiguous questions are identified, but may lead to over-detection—flagging queries that are actually unambiguous. Conversely, high precision minimizes false positives, reducing unnecessary interruptions in downstream systems, but may come at the cost of under-detection—failing to catch all ambiguous cases. The optimal balance depends on deployment context: systems prioritizing safety or correctness may prefer high recall,

Table 3: Distribution of ambiguity labels across AmbiQT training data, BIRD annotated benchmark, and AMBROSIA dataset, and total number of databases, tables, columns, SQL functions, and joins.

Dataset	Ambiguous	Not Ambiguous	Ambiguous (%)	DBs	Tbls	Cols	Avg Tbls	Avg Cols	Avg Funcs	Avg Joins
AmbiQT (Train)	3909	4451	46.75%	166	873	4497	5.2	27.1	0.47	0.59
BIRD (Annotated)	90	39	69.77%	2	22	135	11.0	67.5	0.67	0.48
AMBROSIA	1277	2965	30.10%	1064	5345	21249	5.0	20.0	0.54	1.40

while user-facing applications may value high precision to avoid burdening users with unwarranted clarification requests.

All experiments are run on a single Macbook Air M1 8Gb. Each training run takes approximately 200 minutes for GATekeeper and 100 minutes for BERT-only baselines. In total, we run over 80 hyperparameter configurations across model variants and datasets.

5.3 Baseline Results

We compare GATekeeper against a GPT-4.1-based zero-shot classifier, prompted with the schema and NL question, and asked to predict whether the question is ambiguous with respect to the schema. Results are summarised in Table 4.

GATekeeper outperforms GPT-4.1 in F1 score, with a substantial gap on (0.753 vs. 0.552). While GPT-4.1 achieves relatively high precision (0.621), its recall remains consistently low (0.497), indicating a tendency to under-predict ambiguity. In contrast, GATekeeper both balances precision and recall more effectively and outperforms GPT-4.1 in all measured metrics in-contex, benefiting from both structured supervision and contextual embeddings.

These results suggest that zero-shot LLMs, despite their broad generalisation capabilities, struggle with fine-grained schema-sensitive ambiguity detection. Supervised models like GATekeeper can learn dataset-specific decision boundaries and schema-question interactions that general-purpose LLMs fail to capture.

The decision to use GPT-4.1 as the baseline stems from the absence of any purpose-built solution specifically designed to address the task of identifying SIA. While systems such as AmbiQT[3] and ODIN[21] do engage with SIA to some extent, they do not explicitly evaluate their models' ability to detect it. Instead, their focus lies in assessing how well the models generate queries in response to SIA-related prompts, measuring the resulting execution accuracy of their methods (does the generated query produce the correct result when executed against the database).

5.4 GATekeeper Graph Structure

To assess how much GATekeeper relies on graph structure versus node features, we ran two ablations: (1) replacing schema-derived edges with random ones. We maintain the original set of schema nodes and their embeddings, but we replace the relationally grounded schema edges with random edges—we test with 0.5 edges per node and 2 edges per node. (2) removing the graph entirely in favour of a BERT-only classifier as seen in Table 4.

Performance dropped only slightly under random edge conditions (F1: 0.723/0.690 vs. 0.753), with even higher precision than the full model. This suggests that GATekeeper's contextual BERT embeddings carry enough semantic and relational signal to drive classification. Since these embeddings already reflect question-schema interactions, the GAT primarily acts as a shallow aggregator rather than a structure-sensitive learner.

This behaviour reflects a known pattern in recent GNN literature: when input features are rich (e.g., from pretrained language models), graph structure contributes little. Especially in shallow networks, message passing adds marginal value if long-range dependencies are already embedded in node features. Studies in recommender systems[9], NLP[19], and general graph studies[4, 6], report similar findings, even random or fully connected graphs can perform competitively in such settings.

Several factors likely amplify this effect in our setting. First, GATekeeper uses only 2 layers of GAT, limiting the model's capacity to capture higher-order structural patterns, but preventing over-smoothing. Second, the schemas in AmbiQT are relatively clean and normalised, meaning many ambiguous cases are lexically evident and do not require deep structural disambiguation.

The BERT-only classifier achieves a lower F1 score (0.701), but the highest recall among all models (0.768). This supports the interpretation that strong node features alone can detect many ambiguous cases, but tend to cause over-prediction, reducing precision. The GAT, even when using random edges, moderates this tendency slightly. This may be due to the GAT's architectural constraints: it aggregates information from local neighbourhoods and applies the same parameters across all nodes. Even when the edge structure is uninformative, this consistent aggregation acts as a form of regularisation, helping the model make more conservative predictions.

In summary, GATekeeper's performance appears to rely primarily on the strength of its contextual node embeddings, which already encode rich semantic and relational information. In this setting, the GAT functions mainly as a shallow aggregator, with limited dependence on schema structure. This suggests that in relatively clean and well-structured databases, explicit graph connectivity contributes only marginally to ambiguity detection, though its role may be more pronounced in complex or denormalised schemas.

5.5 GATekeeper Feature Embeddings

We set up a test where we skip the Node Embedding module, and instead use the output from the prediction module as features for the nodes. I.e., the features of each node are the predicted likelihood that the node is relevant to the question.

As shown in Table 4, this leads to a substantial drop in performance.

The scalar relevance scores provide only coarse signals, lacking the semantic detail needed for effective ambiguity detection. Without the high-dimensional, question-aware embeddings from the full model, the GAT cannot distinguish subtle cases of ambiguity.

Table 4: Ambiguity detection performance comparison across model types, ablation studies on graph structure, and embedding strategies. GATekeeper consistently outperforms other configurations, with results indicating that contextual embeddings are more critical than graph structure in this setting.

Category	Model	Accuracy	Precision	Recall	F1 Score
Graph Architecture	GATekeeper(GNN)	0.690	0.765	0.715	0.733
Embedding Ablation	GATekeeper - No embeddings	0.421	0.505	0.417	0.439
Base Performance	GPT-4.1 (No Fine-tuning)	0.500	0.621	0.497	0.552
	BERT Classifier (No Graph)	0.593	0.645	**0.768**	0.701
	Random Edges (0.5 per node)	0.668	**0.817**	0.598	0.690
	Random Edges (2 per node)	0.692	0.817	0.649	0.723
	GATekeeper	**0.702**	0.775	0.732	**0.753**

These results confirm that GATekeeper relies heavily on the representational depth of its learned node features. When those are replaced with simplified inputs, the model's ability to reason about ambiguity degrades significantly.

We also ran parameter sweeps across multiple configurations of both GAT and standard GNN models. Across the board, GAT models consistently outperformed their GNN counterparts, as shown in Table4. The results reported reflect the strongest configuration from each model type.

The GAT's advantage likely stems from its ability to assign adaptive attention weights to neighbouring nodes, enabling more selective and context-sensitive aggregation. In contrast, standard GNNs treat all neighbours uniformly, which may dilute important signals. This suggests that attention mechanisms offer a small benefit for SIA detection, even when input features are already strong.

5.6 Effect of Training Domain Diversity

We evaluate the impact of domain diversity in training data. Specifically, we test whether training on multiple domains leads to significantly better performance on in domain questions.

For this, we train two separate models: one using only AmbiQT and another using a combination of AmbiQT and Ambrosia [18]. Both models are evaluated on AmbiQT test set.

As shown in Table5, adding a second training domain, yields only marginal differences in performance on the AmbiQT test set. The single-domain model trained solely on AmbiQT slightly outperforms the multi-domain variant, but this performance is within training variability. Our interpretation is that the signals captured by the GATekeeper methodology is already well represented in AmbiQT. Consequently, additional training from data in Ambrosia does not offer significant returns because it does not introduce qualitatively different signals for detecting SIA with GATekeeper.

5.7 Cross-Domain Generalisation

We evaluate the model's ability to generalise across domains. Since SIA stems from structural overlaps—such as repeated column names or ambiguous joins—we hypothesize that these patterns will transfer across domains regardless of domain-specific vocabulary.

We train GATekeeper on AmbiQT and evaluate it on BIRD-Bench.

The results seen in Table 5 highlights two key findings. Firstly, GATekeeper fails to generalise in cross-domain settings, despite structural commonalities in SIA. When trained on both ambiQT and Ambrosia, the model achieves high preicison but very low recall, indicating that it struggles to identify ambiguous cases outside its training domain. Conversely, in single-domain training (AmbiQT only), it achieves perfect recall by trivially predicting all examples as ambiguous, this also leads to seemingly strong accuracy, precision and F1, but this is merely a product of a dataset imbalance, rather than meaningful generalisation.

Secondly, a few-shot adaptation dramatically improves model performance. When provided with just a small number (12) of in-context examples from the target domain (BIRD-Bench), GATekeeper surpasses GPT-4.1 in both F1 score and overall balanace between precision and recall. This suggests that the model's inductive bias, while brittle in isolation, can be steered effectively with minial domain-specific supervision.

These results highlight the limitations of relying solely on architectural modeling for schema-level generalisation. When the model encounters schema structures it has not seen during training, such as unfamiliar table layouts, naming conventions, or join patterns, it tends to treat most queries as ambiguous, leading to widespread over-flagging. In practice, this means that structural complexity or novelty can trigger false positives, as seen in BIRD-BENCH whose schemas are significantly more complex than those in AmbiQT. This sensitivity to unseen schema configurations suggests that few-shot learning may offer a practical and efficient way to adapt SIA detectors across diverse database environments.

5.8 Downstream Impact on NL2SQL Execution

As a use-case demonstration, we evaluate the practical benefit of incorporating ambiguity detection into a full NL2SQL pipeline, the particular NL2SQL solution we have chosen in this use-case is OpenSearch[25]. It is one of the highest-scoring open-source models on the BIRD benchmark. Its strong performance makes it a suitable candidate for evaluating the practical impact of integrating ambiguity detection into end-to-end SQL execution.

This experiment evaluates the hypothesis that ambiguity detection can improve end-to-end SQL execution accuracy by preemptively flagging problematic queries. We run two versions of an

Table 5: Performance comparison of GATekeeper and baselines on two evaluation sets: AmbiQT (left) and BIRD-Bench (right). Models vary in training scope (single-domain vs. cross-domain) and evaluation setup (zero-shot, few-shot, LLM baselines).

Training Domain	Model	Evaluated on AmbiQT				Evaluated on BIRD-Bench			
		Accuracy	Precision	Recall	F1 Score	Accuracy	Precision	Recall	F1 Score
Single-domain	GATekeeper (AmbiQT only)	**0.709**	**0.775**	0.732	**0.753**	0.620	0.620	**1.000**	**0.765**
Cross-domain	GATekeeper (AmbiQT + Ambrosia)	0.699	0.770	**0.733**	0.751	0.357	0.640	0.178	0.278
GPT-4.1 (in-context)		-	-	-	-	0.5504	**0.796**	0.478	0.597
GATekeeper (Few-Shot, BIRD only)		-	-	-	-	**0.674**	0.773	0.756	0.764

NL2SQL system—one standard and one augmented with our ambiguity detector—on a dataset known to contain ambiguous queries (BIRD-Bench). In the augmented version, queries identified as ambiguous are either filtered out in our experiment, but could also be rerouted to resolution mechanisms in future implementations. We compare SQL execution accuracy across the full dataset and within the subset of queries flagged as ambiguous.

The experimental results in Table 6 show the effects of incorporating ambiguity detection into an end-to-end NL2SQL system. Without ambiguity detection, the system attempted execution on all 129 queries, achieving 19 correct results and incurring 110 execution failures. In contrast, the ambiguity-aware variant executed only 76 queries—having filtered out 53 identified as ambiguous—but maintained a comparable number of correct outputs (17) while reducing execution failures to 59.

This suggests a marked improvement in execution accuracy. Specifically, the success rate increased from 14.7% (19/129) to 22.1% (17/77), reflecting a 7.4 percentage point improvement in execution accuracy. Although the absolute number of correct executions decreased slightly, this decline is outweighed by the substantial reduction in incorrect executions.

These results highlight a fundamental trade-off introduced by ambiguity detection: coverage versus robustness. While the baseline system executes all inputs regardless of quality, the augmented system adopts a conservative strategy, filtering uncertain inputs to ensure higher quality execution. In scenarios where precision is more critical than recall—such as in production systems with user-facing outputs—this trade-off is advantageous.

Furthermore, the filtered queries represent an opportunity rather than a loss: although skipped in this experiment, they could be redirected in future work to clarification modules, fallback strategies, or human-in-the-loop review systems. Thus, ambiguity detection acts not as a rejection mechanism, but as a decision-making layer that enables safer NL2SQL operation.

It is important to note, however, that these results are inherently sensitive to the design of the evaluation datasets—specifically, how aggressively ambiguous the input questions are constructed. A more lenient ambiguity detector or a dataset with fewer genuinely ambiguous queries could shift the balance between filtered and retained queries, potentially altering both precision and coverage outcomes. This underscores the need to carefully align ambiguity detection thresholds with downstream performance goals and user expectations.

Overall, the integration of ambiguity detection improves the robustness of the system by reducing execution errors, and establishes a promising path for deploying NL2SQL systems in settings that demand high reliability.

Table 6: Comparison of SQL execution results with and without ambiguity detection. Skipped queries in the augmented pipeline correspond to those flagged as ambiguous.

System	Total	Correct	Incorrect	Skipped
OpenSearch	129	19	110	0
OpenSearch + GATekeeper	129	17	59	53

6 Conclusion

This work tackles the under-addressed yet essential problem of proactively detecting SIA in NL2SQL systems. Our method, GATekeeper, leverages Graph Attention Networks and semantically rich BERT embeddings to model schema-question interactions, offering structural awareness and improved interpretability over conventional text-only models. When trained on a single domain (AmbiQT), GATekeeper decisively outperforms GPT-4.1 in ambiguity detection accuracy, validating the benefit of modeling the schema-question interaction. However, cross-domain performance remains limited. Schema complexity and distributional shifts in datasets such as BIRD lead to high recall but poor precision, due to the model over-predicting ambiguity in unfamiliar schema environments. We show that the embeddings derived from BERT play a vital role in the model's performance, removing them causes a significant drop in all metrics. While training on additional domains yields only marginal improvements in generalisation, few-shot adaptation significantly boosts performance—outperforming GPT-4.1 even with as few as 12 examples. This indicates that GATekeeper's inductive bias is steerable and that minimal supervision can enable domain transfer. Incorporating GATekeeper into an end-to-end NL2SQL system demonstrates the potential benefit of SIA detection for downstream tasks. By flagging ambiguous queries before SQL generation, it helps prevent execution errors and enables selective handling strategies—such as clarification or fallback—ultimately improving system robustness. Taken together, our findings paint a clear but qualified picture, schema-aware ambiguity detection is viable and effective in-domain, but brittle out-of-domain. Robust production deployment will require broader training diversity and principled few-shot adaptation.

7 Future Work

A current limitation of our evaluation is the absence of direct comparison to other ambiguity detection models, as SIA classification remains an underexplored task. Most prior work in the NL2SQL domain focuses on full query generation and evaluates systems using execution accuracy or exact match, without isolating ambiguity detection as a distinct subtask. However, as of writing, several related efforts−−particularly around ambiguity-aware NL2SQL generation−−are in pre-release or under review. These may offer valuable points of reference, and indirect comparisons based on execution performance and reported handling of ambiguous inputs could be conducted once such methods are formally published. We hope our work helps establish a foundation for such targeted evaluations in the future.

This work uses the schema-question relation to identify SIA. However, while detection is a necessary foundation, it does not solve the SIA problem alone. Ambiguity resolution could build upon this work to create interactions with users to clarify intent. This involves two key components: (i) presenting the detected ambiguity in a clear and interpretable manner, and (ii) enabling users to disambiguate questions interactively, for example by pruning irrelevant schema elements from the underlying graph representation.

Some existing approaches rely on oracle-style mechanisms, where LLMs attempt to reformulate or resolve the ambiguity autonomously. However, this reintroduces many of the same issues that motivated this work: opacity, inconsistency, and lack of user agency.

Ultimately, effective ambiguity resolution will likely require hybrid approaches—combining model-driven detection with user-guided clarification—to ensure that systems behave reliably in real-world, high-stakes settings.

Acknowledgement

We thank Novo Nordisk, and in particular Henning Pontoppidan Föh and Rasmus Stenholt, for their invaluable assistance with data provision, gold query generation, and support throughout this project. We also extend our gratitude to our advisors Daniele Dell'Aglio, Juan Manuel Rodriguez and Matteo Lissandrini for their guidance and insights, which have been instrumental in shaping this work.

References

[1] Omar Alharbi, Shaidah Jusoh, and Norita Md Norwawi. 2012. Handling Ambiguity Problems of Natural Language Interface for Question Answering. *IJCSI International Journal of Computer Science Issues* 9 (05 2012), 17–25.

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM. https://doi.org/10.1145/3620665.3640366

[3] Adithya Bhaskar, Tushar Tomar, Ashutosh Sathe, and Sunita Sarawagi. 2023. Benchmarking and Improving Text-to-SQL Generation under Ambiguity. arXiv:2310.13659 [cs.CL] https://arxiv.org/abs/2310.13659

[4] Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. 2021. Deep Neural Networks and Tabular Data: A Survey. *CoRR* abs/2110.01889 (2021). arXiv:2110.01889 https://arxiv.org/abs/2110.01889

[5] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. 2021. Multi-Head Attention: Collaborate Instead of Concatenate. arXiv:2006.16362 [cs.LG] https://arxiv.org/abs/2006.16362

[6] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. 2019. A Fair Comparison of Graph Neural Networks for Graph Classification. *CoRR* abs/1912.09893 (2019). arXiv:1912.09893 http://arxiv.org/abs/1912.09893

[7] Avrilia Floratou, Fotis Psallidas, Fuheng Zhao, Shaleen Deep, Gunther Hagleither, Wangda Tan, Joyce Cahoon, Rana Alotaibi, Jordan Henkel, Abhik Singla, et al. 2024. NL2SQL is a solved problem... Not!. In *14th Annual Conference on Innovative Data Systems Research (CIDR 2024)*. CIDR.

[8] Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, et al. 2024. Xiyan-sql: A multi-generator ensemble framework for text-to-sql. *arXiv preprint arXiv:2411.08599* (2024).

[9] Mingxuan Ju, William Shiao, Zhichun Guo, Yanfang Ye, Yozen Liu, Neil Shah, and Tong Zhao. 2024. How Does Message Passing Improve Collaborative Filtering? arXiv:2404.08660 [cs.IR] https://arxiv.org/abs/2404.08660

[10] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The dawn of natural language to SQL: are we fully ready? *arXiv preprint arXiv:2406.01265* (2024).

[11] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Ma Chenhao, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 42330–42357. https://proceedings.neurips.cc/paper_files/paper/2023/file/83fc8fab1710363050bbd1d4b8cc0021-Paper-Datasets_and_Benchmarks.pdf

[12] Kevin Lin, Ben Bogin, Mark Neumann, Jonathan Berant, and Matt Gardner. 2019. Grammar-based neural text-to-sql generation. *arXiv preprint arXiv:1905.13326* (2019).

[13] Qian Liu, Bei Chen, Jian-Guang Lou, Ge Jin, and Dongmei Zhang. 2019. FANDA: A Novel Approach to Perform Follow-up Query Analysis. arXiv:1901.08259 [cs.CL] https://arxiv.org/abs/1901.08259

[14] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2024. A Survey of NL2SQL with Large Language Models: Where are we, and where are we going? arXiv:2408.05109 [cs.DB] https://arxiv.org/abs/2408.05109

[15] Dai Quoc Nguyen, Cong Duy Vu Hoang, Duy Vu, Gioacchino Tangari, Thanh Tien Vu, Don Dharmasiri, Yuan-Fang Li, and Long Duong. 2025. SQ-Long: Enhanced NL2SQL for Longer Contexts with LLMs. *arXiv preprint arXiv:2502.16747* (2025).

[16] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. 2024. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. arXiv:2410.01943 [cs.LG] https://arxiv.org/abs/2410.01943

[17] Jiexing Qi, Jingyao Tang, Ziwei He, Xiangpeng Wan, Yu Cheng, Chenghu Zhou, Xinbing Wang, Quanshi Zhang, and Zhouhan Lin. 2022. RASAT: Integrating Relational Structures into Pretrained Seq2Seq Model for Text-to-SQL. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 3215–3229. https://doi.org/10.18653/v1/2022.emnlp-main.211

[18] Irina Saparina and Mirella Lapata. 2024. AMBROSIA: A Benchmark for Parsing Ambiguous Questions into Database Queries. arXiv:2406.19073 [cs.CL] https://arxiv.org/abs/2406.19073

[19] Michael Sejr Schlichtkrull, Nicola De Cao, and Ivan Titov. 2020. Interpreting Graph Neural Networks for NLP With Differentiable Edge Masking. *CoRR* abs/2010.00577 (2020). arXiv:2010.00577 https://arxiv.org/abs/2010.00577

[20] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093* (2021).

[21] Kapil Vaidya, Abishek Sankararaman, Jialin Ding, Chuan Lei, Xiao Qin, Balakrishnan Narayanaswamy, and Tim Kraska. 2025. ODIN: A NL2SQL Recommender to Handle Schema Ambiguity. arXiv:2505.19302 [cs.DB] https://arxiv.org/abs/2505.19302

[22] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. arXiv:1710.10903 [stat.ML] https://arxiv.org/abs/1710.10903

[23] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2021. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. arXiv:1911.04942 [cs.CL] https://arxiv.org/abs/1911.04942

[24] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. arXiv:1910.03771 [cs.CL] https://arxiv.org/abs/1910.03771

[25] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. OpenSearch-SQL: Enhancing Text-to-SQL with Dynamic Few-shot and Consistency Alignment. arXiv:2502.14913 [cs.CL] https://arxiv.org/abs/2502.14913

[26] Shicheng Xu, Liang Pang, Mo Yu, Fandong Meng, Huawei Shen, Xueqi Cheng, and Jie Zhou. 2024. Unsupervised information refinement training of large language models for retrieval-augmented generation. *arXiv preprint arXiv:2402.18150* (2024).

[27] Tao Yu, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter S Lasecki, and Dragomir Radev. 2019. CoSQL: A Conversational Text-to-SQL Challenge Towards Cross-Domain Natural Language Interfaces to Databases. arXiv:1909.05378 [cs.CL] https://arxiv.org/abs/1909.05378

[28] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2019. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. arXiv:1809.08887 [cs.CL] https://arxiv.org/abs/1809.08887

[29] Kun Zhang, XieXiong Lin, Yuanzhuo Wang, Xin Zhang, Fei Sun, Cen Jianhe, Hexiang Tan, Xuhui Jiang, and Huawei Shen. 2023. ReFSQL: A Retrieval-Augmentation Framework for Text-to-SQL Generation. In *The 2023 Conference on Empirical Methods in Natural Language Processing*. https://openreview.net/forum?id=zWGDn1AmRH

[30] Michael JQ Zhang and Eunsol Choi. 2024. Clarify When Necessary: Resolving Ambiguity with Language Models. https://openreview.net/forum?id=XgdNdoZ1Hc

[31] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.00103 (2017).

[32] Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. 2024. Large language model enhanced text-to-sql generation: A survey. *arXiv preprint arXiv:2410.06011* (2024).