OseiBrefo-Liang at SemEval-2025 Task 8: A Multi-Agent LLM code generation approach for answering Tabular Questions

Emmanuel Osei-Brefo[1] and **Huizhi Liang**[2]
[1] University of West London, Reading, UK
[2]School of Computing, Newcastle University, Newcastle upon Tyne, UK
emmanuel.osei-brefo@uwl.ac.uk,
huizhi.liang@newcastle.ac.uk

Abstract

This paper presents a novel multi-agent framework for automated code generation and execution in tabular question answering. Developed for the SemEval-2025 Task 8, our system utilises a structured, multi-agent approach where distinct agents handle dataset extraction, schema identification, prompt engineering, code generation, execution, and prediction. Unlike traditional methods such as semantic parsing-based SQL generation and transformer-based table models such as TAPAS, our approach leverages a large language model-driven code synthesis pipeline using the DeepSeek API. Our system follows a zero-shot inference approach, which generates Python functions that operate directly on structured data. Through the dynamic extraction of dataset schema and intergration into structured prompts, the model comprehension of tabular structures is enhanced, which leads to more precise and interpretable results. Experimental results demonstrate that our system outperforms existing tabular questioning and answering models, achieving an accuracy of 84.67% on DataBench and 86.02% on DataBench-lite, which significantly surpassed the performances of TAPAS (2.68%) and stable-code-3b-GGUF (27%). The source code used in this paper is available at https://github.com/oseibrefo/semEval25task8

1 Introduction

Large language models (LLMs) have significantly advanced natural language processing (NLP), demonstrating strong capabilities in question answering (QA), code generation, and structured data analysis (Brown et al., 2020; Chen et al., 2021). While LLMs excel in open-domain QA over unstructured text (Osei-Brefo and Liang, 2022), reasoning over tabular data presents additional challenges. These include schema understanding, multi-column aggregation, numerical computation, and execution reliability (Osés Grijalba et al., 2023; Pasupat and Liang, 2015).

Tabular question answering (TQA) has been traditionally approached using the following three main techniques:

Semantic Parsing Approaches: These are traditional methods where models such as Seq2SQL (Zhong et al., 2017) and SQLNet (Xu et al., 2017) translate natural language queries into SQL commands. While effective for structured databases, these approaches require predefined schemas and struggle with generalisation to diverse table structures.

Transformer-Based Table Models: Models such as TAPAS (Herzig et al., 2020) and TaBERT (Yin et al., 2020) jointly encode table structures and queries, enabling direct classification-based predictions. However, they are limited in their ability to perform dynamic computations beyond simple row-based retrieval.

Code-Based Approaches: Recent methods have explored prompting LLMs to generate executable Python functions to extract or compute answers (Chen et al., 2021; Fried et al., 2022). These approaches offer more flexibility than SQL-based models but require safeguards against execution errors, format inconsistencies, and incorrect column selection.

Tabular question answering (TQA) presents unique challenges since it requires models to understand structured data and perform reasoning over numerical and categorical values. Unlike standard text-based QA, TQA often involves multiple computational steps, such as aggregating values, filtering rows, or computing statistics. Traditional methods rely on SQL-based querying or transformer models fine-tuned on tabular data, such as TAPAS. However, these approaches often require extensive training and struggle to generalize to unseen tables.

A promising alternative is to leverage LLMs for program synthesis, where the model generates executable code to answer questions about tabular data. This approach allows for flexible and interpretable reasoning steps while enabling the processing of large datasets beyond the LLM's context window by delegating execution to external interpreters.
We propose a multi-agent system that integrates prompt-based code generation with structured dataset extraction and execution. The system consists of the following multi-agent system workflow:

- **Dataset Extraction Agent:** Loads the dataset files.
- **Schema Agent:** Loads the dataset files, such as parquet files, that correspond to each question and extracts the relevant columns and sample rows.
- **Prompt Engineering Agent:** Constructs a structured prompt that includes the extracted schema and sample data to guide the LLM in the generation of accurate code.
- **Code Generation Agent:** Uses a preferred LLM to generate Python functions designed to process the provided tabular data.
- **Execution Agent:** Runs the generated function on the dataset and returns the computed answer.
- **prediction agent:** Predicts the final answers for each question

Our contributions are as follows:

- A structured approach for table-based question-answering is developed to generate executable Python code.
- We introduce a method to extract structured dataset information and integrate it into the prompt, improving LLM performance for table reasoning.
- We evaluate our approach on the DataBench benchmark and show that it outperforms traditional SQL-based methods and zero-shot LLM prompting.

Unlike zero-shot in-Context Learning (Z-ICL), which provides the entire dataset within the prompt, our approach generates Python functions that execute externally, making it more scalable for large datasets. Experimental results demonstrate the effectiveness of our approach and highlight its adaptability to unseen tabular data structures and its ability to generate accurate responses across multiple answer types.

2 Methodology

2.1 System Overview

Our proposed method follows a structured approach that consists of a pipeline made of extraction agents, schema agents, prompt engineering agents, main inference agents, code generation agents, execution agents, and prediction agents as components. It involves the implementation of a sequential multi-agent pipeline for structured code generation and execution in tabular question answering. The pipeline consists of distinct agents, each responsible for a specific subtask, as demonstrated in Figure 1. To facilitate text generation, GPT-2 Large, a causal language model (CausalLM), is employed. This processes natural language queries and generates structured Python code. The AutoTokenizer is used to tokenize the input queries and ensure they are formatted correctly for model inference.

This setup allows the system to efficiently encode input queries, generate structured responses, and execute inference tasks within the multi-agent framework. The model interacts with schema and prompt engineering agents to produce executable code, which ensures structured tabular reasoning.

2.2 Schema Agent: Extracting Table Structure

The Schema Agent loads the dataset files labelled as all.parquet and sample.parquet associated with each question and extracts the schema of the table. Given a dataset $\mathcal{D}$ with N rows and M columns:

$$\mathcal{D} = \{(C_1, C_2, ..., C_M) \mid R_1, R_2, ..., R_N\} \quad (1)$$

where C_i represents column names and R_j represents row entries. The Schema Agent performs:

- Extraction of the column names from the dataset.
- Retrieval of the first five rows of the dataset for inclusion in the structured prompt.

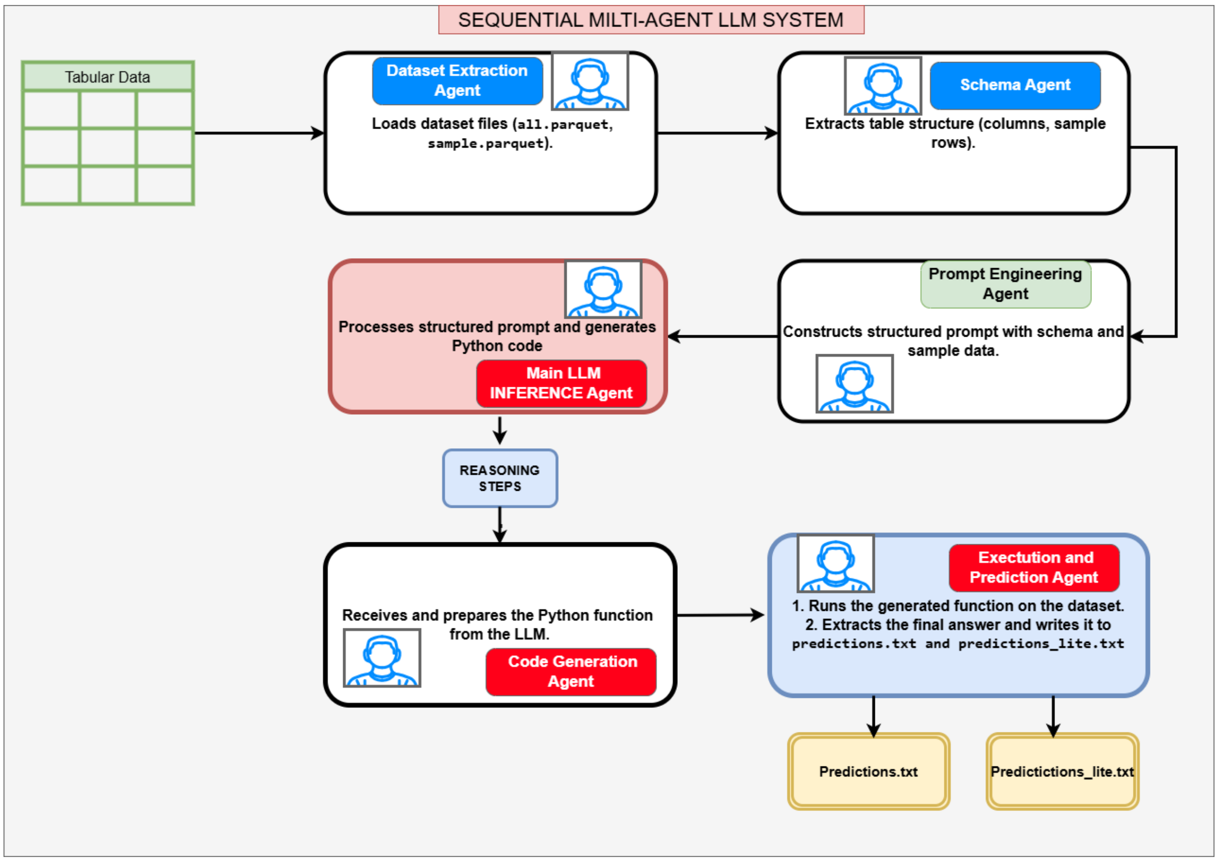

SEQUENTIAL MILTI-AGENT LLM SYSTEM
Tabular Data
Dataset Extraction Agent
Loads dataset files (all.parquet, sample.parquet).
Schema Agent
Extracts table structure (columns, sample rows).
Prompt Engineering Agent
Constructs structured prompt with schema and sample data.
Processes structured prompt and generates Python code
Main LLM INFERENCE Agent
REASONING STEPS
Receives and prepares the Python function from the LLM.
Code Generation Agent
Exectution and Prediction Agent
1. Runs the generated function on the dataset.
2. Extracts the final answer and writes it to predictions.txt and predictions_lite.txt
Predictions.txt
Predictictions_lite.txt

Figure 1: Overview of the multi-agent System, consisting of the extraction agent, schema agent, prompt engineering Agent, Main inference Agent, Code generation Agent, Execution and prediction agents

There is no explicit ranking mechanism to determine the importance of columns, and all columns are included in the prompt without prioritisation.

2.3 Prompt Engineering Agent: Generating Structured Inputs

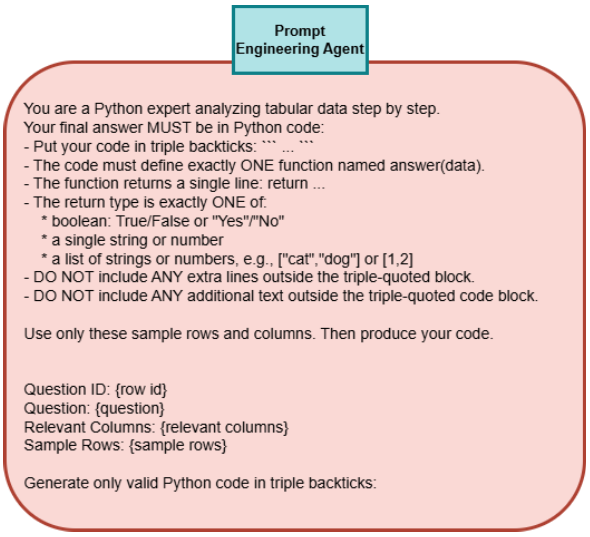
The Prompt Engineering Agent constructs structured prompts for the LLM based on the extracted schema. These prompts guide the LLM in generating accurate responses. An example of the prompts used in this work is demonstrated in figure 2:

For a given question Q and dataset schema, the prompt P is constructed using the following formula in equation 2:

$$P = f_{\text{prompt}}(Q, \{C_k\}_{k=1}^{K}, S) \tag{2}$$

Where:
-S is a set of sample rows and
-$\{C_k\}_{k=1}^{K}$ represents the columns or attributes of the dataset schema.
The current implementation follows these steps:

- Extracts column names and includes them in the prompt.


Prompt Engineering Agent

You are a Python expert analyzing tabular data step by step.
Your final answer MUST be in Python code:
- Put your code in triple backticks: ``` ... ```
- The code must define exactly ONE function named answer(data).
- The function returns a single line: return ...
- The return type is exactly ONE of:
  * boolean: True/False or "Yes"/"No"
  * a single string or number
  * a list of strings or numbers, e.g., ["cat","dog"] or [1,2]
- DO NOT include ANY extra lines outside the triple-quoted block.
- DO NOT include ANY additional text outside the triple-quoted code block.

Use only these sample rows and columns. Then produce your code.

Question ID: {row id}
Question: {question}
Relevant Columns: {relevant columns}
Sample Rows: {sample rows}

Generate only valid Python code in triple backticks:

Figure 2: prompts used for the prompt engineering age

- Provides five sample rows from the dataset.
- Formats the prompt to request a Python function that adheres to specific constraints.

2.4 Code Generation Agent: Production of Executable Code

The Code Generation Agent is responsible for generating Python functions to process tabular data. The implementation invokes an external LLM via the DeepSeek API instead of a fine-tuned local model:

$$\text{Code} = f_{\text{LLM}}(P) \tag{3}$$

Where P represents a structured prompt provided to the LLM, and the output is executable Python code. The generated function follows a specific format as:

```
def answer(data):
    # Model-generated logic
    return result
```

2.5 Execution Agent: Running Generated Code

The generated Python function is executed to produce an answer. However, there is no explicit validation step before execution. Due to resource limitations, the code was assumed to be correct, and execution was attempted without verifying syntax or logical consistency.

2.6 Prediction Generation and Output Formatting

Once execution is completed, the predictions are written to output files (predictions.txt and predictions_lite.txt). This step concludes the process, which involves extracting the dataset, generating code using the LLM, and executing the generated function.

3 Experiments

The system follows a zero-shot approach, relying on prompt engineering to instruct the model on how to retrieve and compute answers from structured tabular data. The experimental setup involved the use of the DeepSeek API.

The dataset was extracted from a competition archive in a ZIP file. The extracted data had multiple subdirectories, with each representing a distinct dataset. Each dataset contained Parquet files, which stored structured tabular data for question-answering. Each dataset ID also corresponded to a folder that contained two files, which are the all.parquet and sample.parquet files. The sample.parquet file contained the first 20 rows of the all.parquet files.

The test data also contained the 522 set of test questions to be answered along with the ID of the corresponding dataset. Table 1 shows the various folders and IDs of the datasets.

Dataset Folder	Description
`066_IBM_HR`	Employee-related data
`067_TripAdvisor`	Travel and hotel reviews
`068_WorldBank_Awards`	Economic and financial data
`069_Taxonomy`	Classification-based dataset
`070_OpenFoodFacts`	Food product information
`071_COL`	Cost of living statistics
`072_Admissions`	Academic admissions and student data
`073_Med_Cost`	Medical and healthcare expenses
`074_Lift`	Ride-sharing or transport statistics
`075_Mortality`	Mortality and health data
`076_NBA`	Basketball statistics
`077_Gestational`	Pregnancy and maternal health data
`078_Fires`	Fire incident reports
`079_Coffee`	Coffee-related statistics
`080_Books`	Book sales and metadata

Table 1: Dataset folders and their descriptions.

A GPT-based model known as deepseek-chat was used as a multi-agent for zero-shot inference through API calls. The model was prompted to generate Python executable codes that extracted the required answer. The prompt ensures that the model adheres to a structured format for consistent and interpretable results.

3.1 Model Configuration and Hyperparameters

The GPT-based model used for inference is `DeepSeek Chat Deepseek-V3`, accessed via an API. The model is configured to generate structured Python code, which is then executed to extract answers. The key hyperparameters used during inference are shown in Table 2:

Category	Parameter	Value / Description
Base Model	Model Name	`gpt2-large`
	Tokenizer	`AutoTokenizer`
Tokenizer	Padding Token	`tokenizer.pad_token`
	Max Length (Default)	1024 tokens (GPT-2 limitation)
	Input Truncation	Enabled (`truncation=True`)
	Padding Strategy	`max_length`
LoRA Configuration	Rank (r)	8
	LoRA Alpha	32
	Target Modules	`["c_attn"]` (Attention Layer)
	LoRA Dropout	0.1
	Bias Handling	`"none"`
Inference	Max Input Token Length	1024 (GPT-2 Large max token limit)
	Model Used for Inference	`DeepSeek-V3 API`
	Temperature	0.0 (Deterministic Outputs)
	API Used	DeepSeek-V3 API)

Table 2: Hyperparameter used for the tokenization and inference.

4 Results and Discussion

Our system was evaluated on the DataBench test benchmark, which consists of real-world datasets with a total of 522 manually curated questions. The evaluation assessed the effectiveness of our approach in generating and executing Python code to answer tabular questions. Performance was measured using the Datatech Eval package, which ensured a standardized assessment across various dataset structures and question types.

The evaluation was carried out to compare the performance of our multi-agent system with existing models, particularly in handling structured tabular data.

4.1 Evaluation

To benchmark the performance of our system, we compared it against two baseline models, which are:

Stable-code-3b-GGUF: A transformer-based generative code model.

TAPAS: A transformer-based tabular QA model designed for direct classification-based predictions. Additionally, we tested our system on two datasets:

DataBench: The full benchmark dataset that contains diverse real-world tabular QA tasks.

DataBench-lite: A smaller subset with simplified queries and table structures.

Models	Accuracy (%)
Stable-code-3b-GGUF (Baseline)	26.00
Transformer-Based (TAPAS)	0.19
Multi-Agent Tabular QA with DeepSeek-V3)	**84.67**

Table 3: Performance comparison on the test DataBench dataset.

As shown in Table 3, our proposed DeepSeek API-based multi-agent model significantly outperformed both TAPAS and Stable-code-3b-GGUF models. The system achieved an accuracy of 84.67%, which is 225.65% higher than that of the stable-code-3b-GGUF model, which is the baseline code used by the organisers.

The table also shows the **TAPAS** model, despite being optimised for tabular data, performed poorly with mere 0.19% accuracy due to its limitations in handling complex aggregation and multi-step computation.

These results demonstrate that a multi-agent system made up of structured prompt engineering and code execution-driven approaches is significantly more effective for tabular question answering than pure transformer-based classification methods.
Table 4 presents results for DataBench-lite, which is a reduced version of the DataBench benchmark with simplified table structures and fewer multi-step reasoning tasks. Here too, our proposed DeepSeek API-based multi-agent model significantly outperformed both TAPAS and Stable-code-3b-GGUF models. It achieved an accuracy of 86.02%, which is 218.59% higher than that of the stable-code-3b-GGUF model, which is the baseline code used by the organisers. The TAPAS model achieved an accuracy of 2.68% under this data.

Models	Accuracy (%)
Stable-code-3b-GGUF (Baseline)	27.00
Transformer-Based (TAPAS)	2.68
Multi-Agent Tabular QA with DeepSeek-V3	86.02

Table 4: Performance comparison on the test DataBench-lite dataset.

The DeepSeek API-based model consistently performed well across both DataBench and DataBench-lite, which demonstrates their robustness in structured data comprehension. On the other hand, the TAPAS model struggled significantly, even with simplified tasks. This demonstrates that direct transformer-based classification models are not well-suited for complex table reasoning. Additionally, the baseline model, Stable-code-3b-GGUF, showed only slight improvement on DataBench-lite, which suggests that generative approaches without structured execution are not sufficiently robust for tabular QA.

Our proposed multi-agent pipeline excels due to the following key factors:

Structured Prompt Engineering Ensures that the LLM receives well-formatted input, including schema details and sample rows.

Code Generation and Execution: Enables the system to dynamically compute answers, unlike transformer models that rely solely on training-based pattern recognition.

Dataset Adaptability: By extracting relevant schema details before inference, the system can generalize to unseen datasets without requiring additional fine-tuning.

4.2 Limitations and Areas for Improvement

While our approach significantly outperforms existing models, a few limitations remain:

Execution Overhead: Running generated code adds an additional processing step, which can increase inference time.

Error Handling Mechanisms: Some syntax errors in generated code require manual validation or debugging, which could be optimized with automated syntax correction.

Scalability for Large Datasets: While the system performs well on structured datasets, handling extremely large tables may introduce computational bottlenecks.

The evaluation results clearly demonstrate that our DeepSeek API-based multi-agent system significantly improves accuracy in tabular question-answering tasks. The ability to dynamically generate and execute Python code allows for greater flexibility compared to purely transformer-based approaches like TAPAS. Future work will focus on enhancing execution efficiency, refining error handling, and optimizing scalability for even larger datasets.

4.3 Error Analysis

Our system follows a structured multi-agent pipeline for generating and executing Python code to answer tabular questions. However, inference errors were observed due to syntax issues, type mismatches, and execution failures. The most frequent error types encountered during inference are categorised in Table 5.

Error Type	Occurrence (%)
Syntax Error in Generated Code	62.5
Data Type Handling Errors	12.5
Column Selection Errors	12.5
Execution Failure (Timeout)	12.5

Table 5: Common failure cases in model predictions.

Syntax errors were the most prevalent issue and contributed to 62.5% of the total error cases encountered. These failures occurred due to missing commas, incorrect indentation, or malformed expressions within the generated Python code.

This error appeared five times out of eight total error cases encountered, indicating a systematic issue in the LLM-generated function. The issue likely stems from improper prompt structuring, leading the model to generate incomplete expressions or incorrectly formatted function calls.

The next category of errors was the data type handling errors. These were the data type mismatches that were a notable issue and contributed to 12.5% of the total error cases. These failures occurred when the model-generated function incorrectly applied string-based operations on numerical values. Some possible causes of this were that the function incorrectly assumed all table values were strings, leading to operations like `.split()` being applied to numeric values. Others could also be due to the model's failure to validate column data types before execution.

There were also column selection errors, which occurred during the selection of the correct column for computation, and they accounted for 12.5% of the total failures. Some of the possible causes of these were: The selection of a categorical column by the model when a numerical column was expected. Other causes include the attempted performance of a numeric comparison on string values by the generated function.

The final category of errors was due to execution failures, which resulted from incorrectly structured generator expressions that led to runtime errors. These made up 12.5% of the total error cases. The possible cause of these errors is the attempt of the function to use a generator expression without proper parentheses. It could also be due to the occurrence of the execution timeout due to an inefficient or infinite loop. Refer to Appendix A for a list of all the sample errors encountered

5 Conclusion

This work has proposed the use of a multi-agent system that integrates prompt-based code generation with structured dataset extraction and execution for tabular question answering. The results demonstrate that multi-agent, execution-driven LLM pipelines are superior to direct prompting techniques and other traditional tabular QA approaches, achieving higher accuracy on real-world tabular data reasoning tasks.

By improving execution validation, incorporating LoRA fine-tuning, and enhancing schema comprehension, this approach could further improve how LLMs interact with structured data. Future work will focus on enhancing the code execution efficiency and the incorporation of automated error correction.

References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yuri Burda, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Daniel Fried, Julian Lu, Josh Wang, Fabian Trummer, Sebastian Riedel, Maarten Bosma, I-Hung Hsu, et al. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Jonathan Herzig, Pawel Nowak, Julian Martin Eisenschlos, Alvin Muis, and Andreas Ernst. 2020. Tapas: Weakly supervised table parsing via pre-training. *arXiv preprint arXiv:2004.02349*.

Emmanuel Osei-Brefo and Huizhi Liang. 2022. UoR-NCL at SemEval-2022 task 6: Using ensemble loss with BERT for intended sarcasm detection. In *Proceedings of the 16th International Workshop on Semantic Evaluation (SemEval-2022)*, pages 871–876, Seattle, United States. Association for Computational Linguistics.

Jorge Osés Grijalba, Luis Alfonso Ureña-López, Eugenio Martínez Cámara, and Jose Camacho-Collados. 2023. Question answering over tabular data with databench: A large-scale empirical evaluation of llms. *arXiv preprint arXiv:2312.XXXXX*.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics*, pages 1470–1480.

Zhong Xu, Sheng Liu, Jialong Sun, Tao Yu, and Dragomir Radev. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Pengcheng Yin, Tao Yu, Graham Neubig, and Dragomir Radev. 2020. Tabert: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 552–561.

A Sample errors encountered

```
__INFERENCE_ERROR__: Error
calling answer(data):
invalid syntax. Perhaps you
forgot a comma? (<string>,
line 1)
```

Mitigation Strategies

- Use an `AST`-based syntax validation step to detect and reject malformed code before execution.
- Modify structured prompts to enforce syntactically complete function generation.
- Implement a post-processing step to correct common syntax errors before execution.

```
__INFERENCE_ERROR__: Error
calling answer(data):
'float' object has no
attribute 'split'
```

Mitigation Strategies

- Enforce explicit type conversion (`float()`, `str()`, `int()`) before processing table values.
- Modify prompts to instruct the model to validate column types before execution.
- Implement exception handling to catch and fix type-related errors dynamically.

```
__INFERENCE_ERROR__: Error
calling answer(data): '<'
not supported between
instances of 'float' and
'str'
```

Mitigation Strategies

- Implement column selection heuristics to improve the relevance of retrieved data.
- Modify prompts to explicitly specify the expected column type for computation.

```
__INFERENCE_ERROR__:
Invalid syntax after fix:
Generator expression must
be parenthesized (<unknown>,
line 4)
```

Mitigation Strategies

- Modify prompt instructions to favor list comprehensions (`[]`) over generator expressions (`()`).