IUST_Champs at SemEval-2025 Task 8: Structured Prompting and Retry Policy for Tabular Question Answering

Arshia Hossein Zadeh[1], Aysa Mayahinia[1], Nafiseh Ahmadi[1]

[1]Iran University of Science and Technology
{arshia_hossein,aysa_mayahinia,nafiseh_ahmadi}@comp.iust.ac.ir

Abstract

This paper presents a novel approach to the task of Question Answering over Tabular Data, as part of SemEval-2025 Task 8. Our system generates executable Python code to derive answers directly from structured data, leveraging open-source large language models. Key innovations include structured prompting, semantic column filtering, and a one-time retry mechanism to enhance accuracy and robustness.
We evaluate our approach on the DataBench and DataBench_Lite datasets, significantly outperforming the baseline accuracy (26-27%) with our best system achieving 70.49% accuracy on the test set. Ablation studies confirm that few-shot prompting and rule-based type classification are crucial for improved performance. Despite these advancements, challenges remain in handling complex table structures and ambiguous queries. Our findings highlight the effectiveness of code-generation-based methods for tabular question answering and provide insights for further research in this area.

1 Introduction

Question answering is to generate precise answers by efficiently interacting with unstructured, structured, or heterogeneous contexts, such as paragraphs, knowledge bases, tables, images, and their combinations. Among these, question answering over tabular data or Tabular Question Answering (TQA) is a challenging task that requireserstanding of table semantics, as well as the ability to reason and infer over relevant table cells (Jin et al., 2022; Wang et al., 2024; Zhao et al., 2023). TQA has been studied with a specific focus as it allows conveniently querying the table in natural language to extract desired information (Patnaik et al., 2024).

Recently, Large Language Models (LLMs) have demonstrated their effectiveness and versatility across diverse tasks, leading to significant advances in natural language processing. This success has spurred researchers to investigate the application of LLMs to table-related tasks (Lu et al., 2025). Many text-related tasks, particularly in the domains of science, technology, engineering and mathematics (STEM), often require intricate reasoning and the use of external tools. However, table processing tasks differ in nature due to the inherent structure of tables and the specific user intent of extracting knowledge from them. For example, LLMs must comprehend table schemas, navigate data within two-dimensional tables, and execute SQL queries to retrieve relevant information. The distinct challenges associated with table processing underscore the importance of adapting LLMs to meet these specialized requirements. Early research, such as TaBERT (Yin et al., 2020), TaPas (Herzig et al., 2020), TURL (Deng et al., 2022), and TaPEx (Liu et al., 2021), adheres to the paradigm of pre-training or fine-tuning neural language models for tables.

DataBench (Os'es Grijalba et al., 2025) is a comprehensive benchmark designed for TQA. Its primary objective is to provide a standardized framework for evaluating and comparing LLMs as tabular reasoners, while also offering flexibility for the comparison of other types of question answering models.

The main strategy of our system for TQA is executable Python code generation to compute answers directly from the provided tabular data. Instead of relying on in-context learning, which struggles with long-table contexts, our approach ensures accurate execution by generating and running Python functions over the dataset. Key techniques include column filtering using semantic similarity to reduce prompt length while retaining relevant information, few-shot prompting to guide concise function generation, and a one-time retry mechanism to handle execution failures. The system employs Qwen2.5-Coder-32B as the primary model for function generation, enhancing accuracy and robustness.

Additionally, we discovered that structured prompting, column filtering, and a one-time retry mechanism significantly enhance accuracy. Our system outperformed the baseline accuracy reported in the original DataBench paper (26% for DataBench and 27% for DataBench_Lite). Empirical analysis demonstrated that few-shot prompting improved response consistency, while the retry mechanism reduced execution failures by enabling the model to regenerate more robust code. Column filtering effectively minimized ambiguity by restricting input to the most relevant attributes. However, our system struggled with complex multi-column reasoning, particularly when implicit relationships between columns needed to be inferred. These findings highlight both the strengths of our approach and areas for future improvement in handling more intricate tabular reasoning tasks. Our code is publicly available at `https://github.com/Arshia-HZ/DataBench-TQA`

2 Related Work

Tabular question answering task requires both the ability to reason over structured data and to understand textual contents in the table. Traditional methods utilize semantic parsing to convert the natural language question into executable commands, which retrieve and process data in the table to obtain answers (Liu et al., 2024). LLMs can learn from a few samples as prompts through in-context learning. This strategy is widely used to give models additional instructions to better solve downstream tasks (Wang et al., 2024)

- **Table Tuning** (Wang et al., 2024; Lei et al., 2023) focuses on LLMs' understanding of tables. This type of research utilizes general-purpose foundation LLMs (e.g., Llama) and a substantial volume of table-related data for instruction tuning. Table tuning examples include Dater (Ye et al., 2023) is the only model that modifies the tabular context while solving table-based tasks. However, the table decomposition in Dater is motivated by the idea that tables could be too large for LLMs to conduct reasoning. It is, therefore, more similar to an LLM-aided data pre-processing than to a part of the reasoning chain since the tabular operations are limited to column and row selections, and fixed for all tables and questions.
- **Code Tuning** (Liu et al., 2024; Kweon et al., 2023; Wang et al., 2025; He et al., 2024) To better solve table-based tasks with LLMs, researchers go beyond general text and resort to using external tools. Propose solving reasoning tasks by generating Python programs and executing them using the Python interpreter. This approach greatly improves the performance of arithmetic reasoning. To further push the limits of programs, Binder (Cheng et al., 2022) generates SQL or Python programs and extends their capabilities by calling LLMs as APIs in the programs.
- **Hybrid of table and code** research reveals that table instruction tuning based on code LLMs is more effective, i.e., code LLMs are tuned using table instruction datasets (Liu et al., 2024).

In this study, we utilize DataBench, a benchmark consisting of 65 real world datasets, with 3,269,975 and 1615 columns in total from different domains, widely different numbers of rows and columns and heterogeneous data types. Moreover, DataBench has 20 hand-made questions per dataset, with a total number of 1300 questions. Questions are further split in different types depending on the type of answer (i.e., true/false, categories from the dataset, numbers or lists), and each question is accompanied by their corresponding gold standard answer. The dataset is entirely in English. The benchmark consists of two subtasks:

- Subtask A: Utilizes the original DataBench dataset.
- Subtask B: Employs a sample of 20 rows extracted from the original DataBench dataset, referred to as DataBench_Lite.

3 System Overview

Our Tabular Question Answering (TQA) system generates and executes concise Python functions to answer natural language queries over structured tables. By leveraging code execution, we guarantee both accuracy and interpretability, avoiding common failures of pure language-model-based approaches on long or wide tables.

3.1 Column Selection

To reduce prompt length and noise, we filter the most relevant columns for each question:

1. **Semantic Similarity Scoring**: We embed column names and the input question using a pre-trained sentence embedding model (e.g., SBERT) and compute cosine similarity.

2. **Top-k Selection**: We select the top-k columns with highest similarity scores, ensuring key attributes are retained while minimizing context size.

3.2 Answer Type Classifier

Inferring the expected output type guides structured responses:

- **Heuristic Analysis**: We parse question tokens (e.g., How many", What is the average") to predict return types such as `integer`, `float`, `string`, or `list`.

- **Template Enforcement**: The predicted type constrains the generated function signature and final `return` statement, reducing malformed outputs.

3.3 Code Generation via Few-Shot Prompting

Our system uses a few-shot prompting strategy to generate Python functions that return the final answer in a single line. Each prompt includes a number of carefully selected demonstrations that cover a range of typical operations. All examples follow a consistent format, using minimal syntax and a simple `def answer(table):` signature to guide the model toward generating clean and executable code.

To maintain output quality and reliability, we employ Qwen2.5-Coder-32B, a high-performance language model optimized for code generation. Rather than relying on rigid templates, we organize the examples in a coherent, narrative format that exposes the model to diverse query styles and computational needs. This approach helps the model identify and apply the right patterns when faced with new questions.

This strategy enhances the model's ability to generalize across diverse queries and tabular structures. Compared to a templated baseline, our method achieves a noticeable improvement in robustness and code quality (see Table 1).

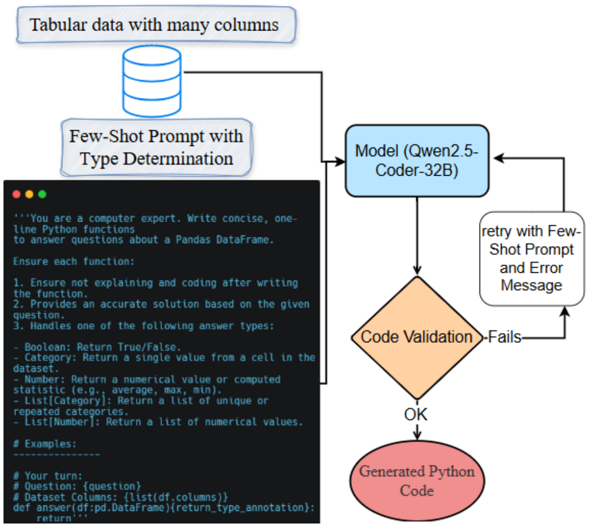
A high-level overview of the code generation workflow is shown in Figure 1.

3.4 Execution and Retry Mechanism

To handle occasional generation errors:

- **Automatic Execution**: We run each function in an isolated Python environment with the filtered DataFrame.

- **One-Time Retry**: If execution fails (e.g., `KeyError`, `TypeError`), we append the error message to our prompt and request a corrected function.


Tabular data with many columns
Few-Shot Prompt with Type Determination
Model (Qwen2.5-Coder-32B)
retry with Few-Shot Prompt and Error Message
Code Validation
Fails
OK
Generated Python Code

```
'''You are a computer expert. Write concise, one-
line Python functions
to answer questions about a Pandas DataFrame.

Ensure each function:

1. Ensure not explaining and coding after writing
the function.
2. Provides an accurate solution based on the given
question.
3. Handles one of the following answer types:

- Boolean: Return True/False.
- Category: Return a single value from a cell in the
dataset.
- Number: Return a numerical value or computed
statistic (e.g., average, max, min).
- List[Category]: Return a list of unique or
repeated categories.
- List[Number]: Return a list of numerical values.

# Examples:
---------------

# Your turn:
# Question: {question}
# Dataset Columns: {list(df.columns)}
def answer(df:pd.DataFrame){return_type_annotation}:
    return'''
```

Figure 1: Tabular Question Answering Workflow Using Few-Shot Prompting and Retry Mechanism for Robust Code Generation.

4 Experimental Setup

All experiments utilize the official train/dev/test splits provided in DataBench and DataBench_Lite. DataBench_Lite consists of tables with a maximum of 20 rows, while DataBench includes tables with significantly larger numbers of rows. Accuracy is used as the primary evaluation metric, computed using the databench_eval package.
During development, we tested different few-shot strategies, varying the number of examples between 1 and 5, and validated their effectiveness on the dev set. The final configuration was determined through manual tuning, selecting the number of examples that maximized accuracy without exceeding context limits. No additional text preprocessing was applied beyond standard column filtering, ensuring fair evaluations across datasets. The following definitions differentiate between model configurations:

- Baseline: Standard prompting without column filtering or retry mechanisms.
- Few-shot: Incorporates in-context examples for improved generalization.
- Retry-enabled: Implements a one-time retry policy for execution failures.

Experiments were conducted in two parts:

1. Code generation using Qwen2.5-Coder-32B with structured few-shot prompting.
2. Error handling through execution retry strategies.

Hyperparameter tuning focused on key parameters such as temperature (0.7), max tokens (512), and prompt format variations. These were manually adjusted based on dev set performance. Since execution failures were a significant concern, our retry mechanism involves resubmitting the prompt with error details when execution fails, enabling the model to generate more robust solutions.
The databench_eval package is used to compute accuracy, ensuring automated comparison against gold-standard answers. The original DataBench paper reported a baseline accuracy of 26% for both DataBench and 27% for DataBench_Lite. Our best-performing system, which leverages open-source models alongside structured prompting and retry mechanisms, significantly improves upon this baseline. Through systematic preprocessing, prompt engineering, and controlled error handling, our system achieves substantial accuracy gains over the benchmark.

5 Results

Our system demonstrates substantial improvements over the baseline accuracy reported in the original DataBench paper. The baseline accuracy for DataBench and DataBench_Lite was 26% and 27%, respectively, whereas our best-performing configuration, utilizing structured prompting and a one-time retry mechanism, achieves significantly higher accuracy on both benchmarks (see Table 1).
To analyze the impact of different design choices, we did some research. Few-shot prompting resulted in a noticeable improvement over zero-shot prompting, as it provided structured examples that guided the model toward more accurate predictions. The retry mechanism contributed to further accuracy gains by allowing the model to regenerate more robust code when execution failures occurred. We also evaluated the effect of column filtering, observing that restricting the input to the most relevant columns reduced ambiguity and improved performance.Figure 2.
Error analysis reveals that the system still struggles with complex multi-column reasoning, especially when implicit relationships between columns must be inferred.

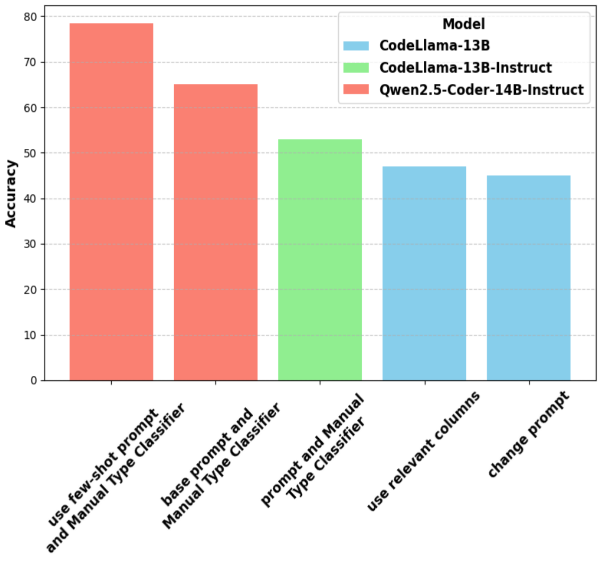


Figure 2: Accuracy of different model configurations and prompting strategies on the development set.

6 Conclusion

In this work, we introduced a Tabular Question Answering (TQA) system that generates executable Python code to compute answers directly from tabular data. Our approach incorporates structured prompting, semantic column filtering, and a retry mechanism to enhance robustness. We evaluated our system on the DataBench and DataBench_Lite datasets, achieving significant improvements over the reported baselines. Experimental results demonstrate that few-shot prompting combined with a manual type classifier leads to the highest accuracy, with Qwen2.5-Coder-32B achieving 70.49% accuracy on the test set. Table 1.

Future work includes refining the column selection process by incorporating adaptive filtering strategies, integrating more advanced program synthesis techniques to improve code reliability, and expanding the system to handle more complex table structures. Additionally, exploring alternative

Experiment	**Model**	**Databench**	**Databench_lite**
Zero-shot prompt + Type Classifier	CodeLlama-13b-Instruct	53.26	54.40
Few-shot prompt + Type Classifier	Qwen2.5-Coder-14B-Instruct	65.33	68.96
Few-shot prompt + Type Classifier + Retry mechanism	Qwen2.5-Coder-14B-Instruct	68.77	69.73
Few-shot prompt + Type Classifier + Retry mechanism	Qwen2.5-Coder-32B	**70.49**	**69.73**

Table 1: Accuracy results for different experiments and models on the test set.

models and optimizing inference efficiency will further enhance the system's practicality for real-world applications.

References

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2022. Binding language models in symbolic languages. *arXiv preprint arXiv:2210.02875*.

Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2022. Turl: Table understanding through representation learning. *ACM SIGMOD Record*, 51(1):33–40.

Xinyi He, Mengyu Zhou, Xinrun Xu, Xiaojun Ma, Rui Ding, Lun Du, Yan Gao, Ran Jia, Xu Chen, Shi Han, et al. 2024. Text2analysis: A benchmark of table question answering with advanced data analysis and unclear queries. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 18206–18215.

Jonathan Herzig, Paweł Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. Tapas: Weakly supervised table parsing via pre-training. *arXiv preprint arXiv:2004.02349*.

Nengzheng Jin, Joanna Siebert, Dongfang Li, and Qingcai Chen. 2022. A survey on table question answering: recent advances. In *China Conference on Knowledge Graph and Semantic Computing*, pages 174–186. Springer.

Sunjun Kweon, Yeonsu Kwon, Seonhee Cho, Yohan Jo, and Edward Choi. 2023. Open-wikitable: Dataset for open domain question answering with complex reasoning over table. *arXiv preprint arXiv:2305.07288*.

Fangyu Lei, Xiang Li, Yifan Wei, Shizhu He, Yiming Huang, Jun Zhao, and Kang Liu. 2023. S Θ3 hqa: A three-stage approach for multi-hop text-table hybrid question answering. *arXiv preprint arXiv:2305.11725*.

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2021. Tapex: Table pre-training via learning a neural sql executor. *arXiv preprint arXiv:2107.07653*.

Yujian Liu, Jiabao Ji, Tong Yu, Ryan Rossi, Sungchul Kim, Handong Zhao, Ritwik Sinha, Yang Zhang, and Shiyu Chang. 2024. Augment before you try: Knowledge-enhanced table question answering via table expansion. *arXiv preprint arXiv:2401.15555*.

Weizheng Lu, Jing Zhang, Ju Fan, Zihao Fu, Yueguo Chen, and Xiaoyong Du. 2025. Large language model for table processing: A survey. *Frontiers of Computer Science*, 19(2):192350.

Jorge Os'es Grijalba, Luis Alfonso Ure na-L'opez, Eugenio Mart'inez C'amara, and Jose Camacho-Collados. 2025. SemEval-2025 Task 8: Question Answering over Tabular Data. In *Proceedings of the 19th International Workshop on Semantic Evaluation (SemEval-2025)*, Vienna, Austria. Association for Computational Linguistics.

Sohan Patnaik, Heril Changwal, Milan Aggarwal, Sumit Bhatia, Yaman Kumar, and Balaji Krishnamurthy. 2024. Cabinet: Content relevance based noise reduction for table question answering. *arXiv preprint arXiv:2402.01155*.

Yuxiang Wang, Junhao Gan, and Jianzhong Qi. 2025. Tabsd: Large free-form table question answering with sql-based table decomposition. *arXiv preprint arXiv:2502.13422*.

Zilong Wang, Hao Zhang, Chun-Liang Li, Julian Martin Eisenschlos, Vincent Perot, Zifeng Wang, Lesly Miculicich, Yasuhisa Fujii, Jingbo Shang, Chen-Yu Lee, et al. 2024. Chain-of-table: Evolving tables in the reasoning chain for table understanding. *arXiv preprint arXiv:2401.04398*.

Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. 2023. Large language models are versatile decomposers: Decomposing evidence and questions for table-based reasoning. In *Proceedings of the 46th international ACM SIGIR conference on research and development in information retrieval*, pages 174–184.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. Tabert: Pretraining for joint understanding of textual and tabular data. *arXiv preprint arXiv:2005.08314*.

Wenting Zhao, Ye Liu, Yao Wan, Yibo Wang, Zhongfen Deng, and Philip S Yu. 2023. Localize, retrieve and fuse: A generalized framework for free-form question answering over tables. *arXiv preprint arXiv:2309.11049*.