Firefly Team at SemEval-2025 Task 8: Question-Answering over Tabular Data using SQL/Python generation with Closed-Source Large Language Models

Ho Thuy Nga[1,2] and **Ho Thi Thanh Tuyen**[1,2]
Le Minh Hung[1,2] and **Dang Van Thin**[1,2]
[1]University of Information Technology, Ho Chi Minh City, Vietnam
[2]National University, Ho Chi Minh City, Vietnam
{22520926,22521627}@gm.uit.edu.vn, {hunglm,thindv}@uit.edu.vn

Abstract

In this paper, we describe our official system of the Firefly team for two main tasks in the SemEval-2025 Task 8: Question-Answering over Tabular Data. Our solution employs large language models (LLMs) to translate natural language queries into executable code, specifically Python and SQL, which are then used to generate answers categorized into five predefined types. Our empirical evaluation highlights the superiority of Python code generation over SQL for this challenge. Besides, the experimental results show that our system has achieved competitive performance in two subtasks. In Subtask I: Databench QA, where we rank the Top 9 across datasets of any size. Besides, our solution achieved competitive results and ranked 5^{th} place in Subtask II: Databench QA Lite, where datasets are restricted to a maximum of 20 rows.

1 Introduction

The Shared Task 8 (Os'es Grijalba et al., 2025) aims at building Question-Answering (QA) systems for tabular data using the DataBench benchmark (Grijalba et al., 2024), which contains 65 real-world tabular datasets from different domains, allow to assess distinct sort of questions related to each data type. This shared task has two main tasks, including DataBench QA and DataBench Lite QA. The DataBench QA subtask requires developing a system that answers questions using datasets of any size. The DataBench Lite QA subtask used the sampled version of each dataset with a maximum of 20 rows per tabular dataset (DataBench Lite). The dataset involves questions with answers and their type (including boolean, number, category, list[number], and list[category]), name of specific columns used and their types, along with the associated dataset name. The system developed by the participants will need to provide an answer which would then be compared with a gold standard.

In this paper, we propose a system that uses large language models (LLMs) to solve Shared Task 8. Our system is based on the GPT models and a structured query generation approach by producing either SQL queries or Python code to answer questions on tabular data. The generated SQL or Python code is executed on the dataset, and refines the output to produce the final response.

The rest of the paper is organized as follows. Section 2 provides the related work. The system description is presented in Section 3, followed by evaluation results in Section 1. The experimental setup and conclusion is discussed in Section 4 and Section 6, respectively.

2 Related Work

Question-Answering over Tabular Data (QAoTD) has garnered significant interest due to its broad applications in structured data retrieval and decision support systems. Existing approaches can be broadly categorized into rule-based methods, transformer-based architectures, and code-generation techniques, each addressing different challenges associated with querying tabular data.

Building on previous research, Vakulenko and Savenkov (2017) proposed a system that enables non-technical users to query open datasets using natural language. Their approach extends an existing chat-bot interface for metadata-based search by incorporating content-based retrieval from tables, allowing users to pose queries and obtain answers directly from structured data. Compared to earlier deep learning-based models, such as Sun et al. (2016), which focus on cross-table search, and Yin et al. (2016), which explore learning aggregation operations, this system offers a more lightweight and user-friendly

Proceedings of the The 19th International Workshop on Semantic Evaluation (SemEval-2025), pages 1028–1033

solution, reducing computational overhead while enhancing usability for open data exploration.

To improve numerical reasoning and compositional query execution, Zhou et al. (2022) introduced UniRPG, a program synthesis approach designed for discrete reasoning over both tables and text. Their method leverages a neural programmer to generate executable programs, ensuring syntactic correctness and improving reliability in arithmetic computations. Prior research, such as Cao et al. (2023), explored converting natural language questions into executable Python programs, enabling flexible table processing and API integration. While these approaches enhance structured reasoning, they remain sensitive to execution errors and struggle to generalize to unseen table schemas.

More recently, advances in large-scale pre-trained language models have significantly impacted table-based question answering. Deng et al. (2024) conducted a study assessing the table reasoning capabilities of large language models (LLMs) and multimodal LLMs, analyzing their performance across different prompting techniques and table formats. Their findings, published in the Findings of ACL 2024, indicate that while LLMs exhibit strong generalization abilities, they often face challenges in numerical reasoning and logical consistency. Compared to earlier table-based QA models such as TAPAS (Herzig et al., 2020), which leverage weak supervision for table parsing or TAPEX (Liu et al., 2022), which achieves table pre-training by learning a neural SQL executor on a synthetic corpus, LLMs offer greater flexibility but remain prone to hallucinations when dealing with structured information.

3 System Description

3.1 Approach

The diagram in Figure 1 illustrates our approach for Subtask I and Subtask II. The system is composed of five main components: Pre-processing, Select Relevant Columns, Generate Code, Execute Generated Code and Fixing, and Generate Answer. Pre-processing involves normalizing raw input data to improve consistency and quality. The system then using GPT models to identify relevant columns and extract sample data to aid code generation. In the Generate Code stage, relevant dataset attributes and query context are provided to the model to generate executable code. This process follows one of two strategies: generating Python code for direct data manipulation or SQL queries for structured database retrieval. The generated code is then executed in the Execute Generated Code and Fixing stage to extract the necessary information from the dataset. Finally, in the Generate Answer stage, the extracted results are transformed to align with one of five predefined answer types. The detailed structure of the system is described in the following.

Pre-processing Dataset: Pre-processing is one of the essential components in building an effective Question-Answering system for tabular data. In this task, we apply a standardized pre-processing step that focuses on normalizing null values to ensure consistency across different datasets.

Select Relevant Columns: As illustrated in Figure 1, we leverage the power of GPT models through designed structured prompts to identify relevant columns for answering a given question. To achieve this, we first provide the model with the full set of column names and the input question. Specifically, given a dataset with M columns $C = \{c_1, c_2, ..., c_M\}$ and an input question Q, we construct a structured prompt that presents both the column metadata and the question to the language model. The model then processes this input and predicts a subset of relevant columns $C_{\text{rel}} \subseteq C$ that are most likely to contain the necessary information for answering Q.
Once the relevant columns are selected, we proceed to extract sample data from these columns to provide additional context for subsequent stages. The extracted sample data serves as a representative subset of the information within the selected columns, helping to enhance the accuracy of code generation in the later steps. By leveraging the contextual understanding of large language models, our approach ensures that both the selected columns and the extracted sample data are semantically aligned with the question, thereby structuring the information effectively for downstream processing.

Generate Code: Given the input question, the selected relevant columns from the *Select*

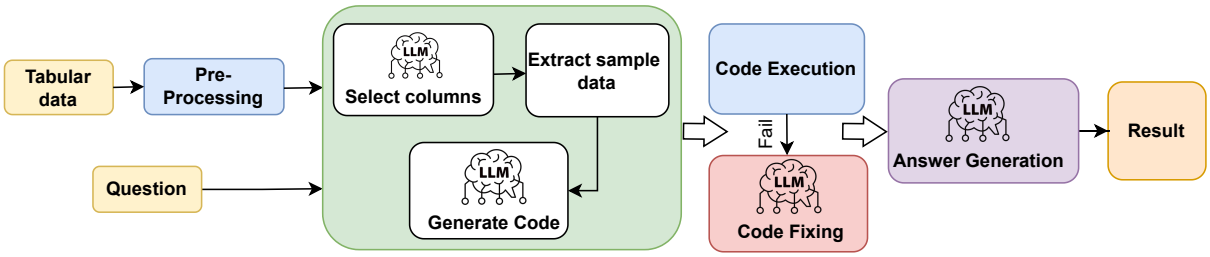

Figure 1: Pipeline for Question-Answering over Tabular Data Using Large Language Models.

Relevant Columns step, and sample data from these columns, we generate executable code to retrieve the answer. We formulate a structured prompt that encapsulates the extracted column names, representative data samples from each selected column, and additional context information—specifically, the table name for SQL-based generation or the CSV path for Python-based generation. Based on this input, we employ two distinct code generation strategies: one for generating Python code and another for generating SQL code. Figure 2 illustrates two examples of code generation. The first example shows Python code, while the second demonstrates SQL code. We designed specific prompts to guide the model behavior (Appendix A.1). This approach ensures adaptability in handling different types of data retrieval and computation tasks efficiently.

Execute Generated Code and Fixing: *For Python-generated code*, we execute the generated script directly on the provided CSV dataset. The script is designed to process the extracted columns, perform necessary computations, and return the relevant answer. Execution is handled within a controlled environment to ensure correctness and prevent unintended operations. The output of this execution serves as the extracted answer to the input question. *For SQL-generated code*, we first convert the CSV file into a structured database format using SQLite. The extracted columns and sample data are loaded into an SQLite database, where the generated SQL query is executed. This ensures efficient and structured querying over tabular data. The result of the SQL query is then returned as the answer to the question.

To maintain execution reliability, we implement an error-handling mechanism that detects and addresses failures occurring in either execution method. However, execution may encounter errors such as syntax errors or invalid operations. To handle these issues, the system implements a two-step fixing strategy: first, capturing the exact error message along with the code snippet; second, feeding these into the large language model using an error-correction prompt. The model then predicts a corrected version of the code, which is re-executed iteratively until successful or a maximum retry limit is reached. This iterative refinement process enhances robustness and adaptability, ensuring more accurate and reliable extraction of answers while minimizing execution failures.

Generate Answer: The result from the *Execute Generated Code* step is processed to align with the expected answer type. Based on the question's context and extracted data, the output is transformed into one of the predefined types: *Boolean, Number, Category, List[Number], or List[Category]*. This ensures consistency and accuracy in answering different types of questions while preserving the structure of the extracted information.

3.2 Large Language Models

We utilized two models from the GPT family of large language models (Kalyan, 2023) in this work.

- **GPT-4o**: GPT-4o is an autoregressive omni-model developed by OpenAI (OpenAI et al., 2023), capable of handling complex reasoning tasks and generating high-quality text responses efficiently.
- **GPT-3.5-turbo**: GPT-3.5-turbo, also developed by OpenAI, is an advanced version of GPT-3.5 with improved performance in understanding natural language and text generation.

4 Experimental Setup

Data and Preprocessing: We utilized the official development set for system development. As the

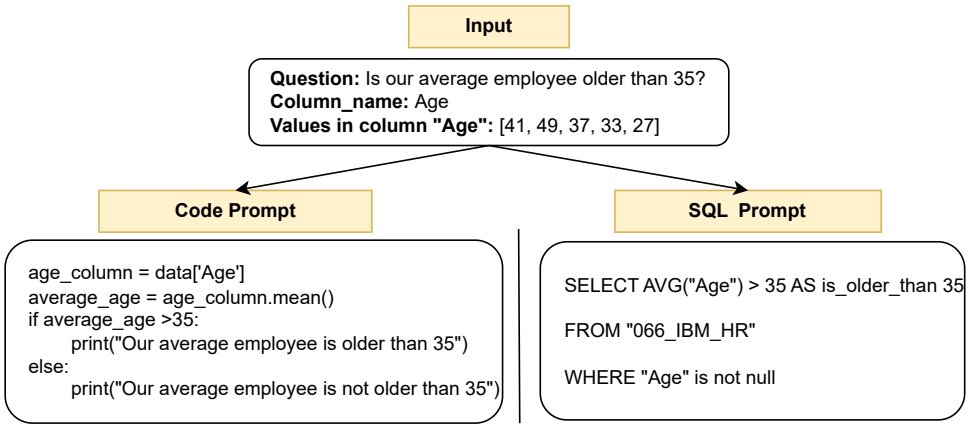

Figure 2: Examples of Generated Python and SQL Code.

competition rules stipulated, no additional data was used during the system development process.

Evaluation Metrics: The evaluation metric for two Subtasks is Accuracy Score between submission and test gold set. These evaluation metrics were provided by the task organizers and are available on GitHub.

Configuration Settings: We use the OpenAI API to access two large language models for setting up our experiments.

5 Results and Discussion

In this section, we present the official results of our final submission model for two tasks and the accuracy results split by question type for each task in the Task 8 Shared Task competition. We also compare results with the results from the five top teams for each task.

Subtask I: DataBench QA Table 1 presents the performances of our system. Table 2 shows the performances of our system compared with the five top teams on Task I. The official results on CondaBench show that we achieved an accuracy of 86.40% on the test set (Top 9).

Subtask II: DataBench Lite QA Table 1 presents the results of our submission on Subtask II. Table 3 presents the performances of our system compared with five top teams based on the final rankings. According to the official results on CondaBench, we ranked 5^{th} with an accuracy of 86.21% on the test set of this subtask.

Table 1 presents the experimental results obtained using the official evaluation function provided by the organizers on GitHub. These results highlight the performance of different approaches across various question types. Experimental results show that the Python-based approach with GPT-4o achieves the highest accuracy, with 79.69% on DataBench QA and 81.99% on DataBench Lite QA. The model performs particularly well on Boolean (96.90% and 95.35%) and Category (87.84%) questions. The SQL-based approach also demonstrates stable performance, especially for Category questions (90.54% in both tasks). Overall, GPT-4o outperforms GPT-3.5-turbo across all metrics, highlighting its superior ability to handle complex queries.

However, the system struggles with list-type questions (list[category] and list[number]), achieving only 47.22% and 55.56% accuracy with Python GPT-4o. Additionally, GPT-3.5-turbo performs significantly worse in Python-based queries, particularly for Category questions (35.14% and 45.95%). The main cause of low performance in these types of questions is errors in applying conditional filters that remove important data or retain unwanted values. Moreover, errors in duplicate processing can remove essential data, leading to inaccurate or incomplete results. The accuracy across different question types remains uneven, showing that improvements in list extraction and numerical reasoning are needed to enhance overall performance.

Subtask	Method	boolean	category	number	list[category]	list[number]	Average
Subtask I	GPT-3.5-turbo (SQL)	59.69	59.46	52.56	41.67	58.24	54.79
	GPT-4o (SQL)	73.64	90.54	73.72	59.72	81.32	75.48
	GPT-3.5-turbo (Python)	72.09	35.14	66.03	43.06	58.24	58.62
	GPT-4o (Python)	96.90	87.84	77.56	47.22	78.02	**79.69**
Subtask II	GPT-3.5-turbo (SQL)	66.67	70.27	58.33	50.00	61.54	61.49
	GPT-4o (SQL)	85.27	90.54	80.13	55.56	79.12	79.31
	GPT-3.5-turbo (Python)	72.09	45.95	73.72	50.00	75.82	66.48
	GPT-4o (Python)	95.35	87.84	80.77	55.56	81.32	**81.99**

Table 1: Results for Subtasks I and II: Databench and Databenchlite Question-Answering on the test set.

Rank	Team	Accuracy
Top 1	TeleAI	95.01
Top 2	AILS-NTUA	89.85
Top 3	SRPOL AIS	89.66
Top 4	sonrobok4	89.46
Top 5	langtechdata61	88.12
Ours (Top 9)	Firefly	86.40

Table 2: Results of our best system compared with five top systems for Subtask I: Databench

Rank	Team	Accuracy
Top 1	TeleAI	92.91
Top 2	AILS-NTUA	88.89
Top 3	langtechdata61	88.70
Top 4	SRPOL AIS	86.59
Ours(Top 5)	Firefly	86.21

Table 3: Results of our best system compared with five top systems for Subtask II: Databenchlite

6 Conclusion

In this paper, we presented a system for Question-Answering on tabular data in the SemEval-2025 Task 8. Our approach leverages large language models to generate column descriptions, select relevant columns, and produce executable code to derive final answers. The system efficiently processes diverse tabular datasets and generates accurate responses without requiring additional training data. Our experiments demonstrate competitive performance across different datasets in DataBench. For future work, we plan to enhance our system's ability to handle more complex table structures and improve its accuracy in missing or ambiguous data. Additionally, exploring better prompting strategies for large language models could further optimize system performance.

Acknowledgements

This research was supported by The VNUHCM-University of Information Technology's Scientific Research Support Fund.

References

Yihan Cao, Shuyi Chen, Ryan Liu, Zhiruo Wang, and Daniel Fried. 2023. Api-assisted code generation for question answering on varied table structures. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Naihao Deng, Zhenjie Sun, Ruiqi He, Aman Sikka, Yulong Chen, Lin Ma, Yue Zhang, and Rada Mihalcea. 2024. Tables as texts or images: Evaluating the table reasoning ability of llms and mllms. In *Findings of the Association for Computational Linguistics: ACL 2024*. Association for Computational Linguistics.

Jorge Osés Grijalba, Luis Alfonso Ureña-López, Eugenio Martínez Cámara, and Jose Camacho-Collados. 2024. Question answering over tabular data with databench: A large-scale empirical evaluation of llms. In *Proceedings of LREC-COLING 2024*, Turin, Italy.

Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. 2020. Tapas: Weakly supervised table parsing via pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Katikapalli Subramanyam Kalyan. 2023. A survey of gpt-3 family large language models including chatgpt and gpt-4. *arXiv preprint arXiv:2310.12321*.

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2022. Tapex: Table pre-training via learning a neural sql executor. *arXiv preprint arXiv:2107.07653*.

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

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.

Huan Sun, Hao Ma, Xiaodong He, Wen-tau Yih, Yu Su, and Xifeng Yan. 2016. Table cell search for question answering. In *Proceedings of the 25th International Conference on World Wide Web (WWW '16)*, pages 771–782, Montréal, Canada. ACM.

Svitlana Vakulenko and Vadim Savenkov. 2017. Tableqa: Question answering on tabular data. *arXiv preprint*, arXiv:1705.06504.

Pengcheng Yin, Zhengdong Lu, Hang Li, and Kao Ben. 2016. Neural enquirer: Learning to query tables in natural language. In *Proceedings of the Workshop on Human-Computer Question Answering*. Association for Computational Linguistics.

Yongwei Zhou, Junwei Bao, Chaoqun Duan, Youzheng Wu, Xiaodong He, and Tiejun Zhao. 2022. Unirpg: Unified discrete reasoning over table and text as program generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

A Appendix

A.1 Prompt Engineering

Prompt Design for Python Code Generation.

Write a Python script to extract data related to the question: `{question}`
Do not write anything except the python code.

Follow the instruction below:
1. Reads the CSV file from path `{csv_file_path}`
2. The columns used in the data are `{relevant_col}`.
3. The columns information are `{relevant_info}`, this includes column name and several different values extracted from that column (which may or may not be the entire value in the column).
4. Just print the complete answer sentence based on the answer of the last question and the question.

Prompt Design for SQL Code Generation.

Write an SQL script to extract data related to the question: `{question}`
Follow the instruction below:
1. Need to keep the table name in `" "`
2. If any column name matches a reserved keyword in SQLite (e.g: Transaction,...), ensure it is wrapped in double quotes (`" "`) to avoid syntax errors.
3. Do not use `DISTINCT` unless the question explicitly requires unique values, such as when it contains terms like "different value", "unique", or "distinct".
4. Use `ORDER BY column DESC` when sorting in descending order. Use `ORDER BY column ASC` when sorting in ascending order.
5. Only provide the SQL query; do not include any explanations or additional text.
Now write an SQL query to answer the question: `{question}` based on the table name `{table_name}`, the columns used `{relevant_col}`, and several different values extracted from those columns `{relevant_info}`.