ScottyPoseidon at SemEval-2025 Task 8: LLM-Driven Code Generation for Zero-Shot Question Answering on Tabular Data

R Raghav[*1] **Adarsh Prakash Vemali**[*2]
Darpan Aswal[3] **Rahul Ramesh**[1] **Ayush Bhupal**[1]

[1]Independent Researcher [2]University of California, San Diego
[3]Université Paris-Saclay
{rraghav5600, vemali.adarsh}@gmail.com

Abstract

Tabular Question Answering (QA) is crucial for enabling automated reasoning over structured data, facilitating efficient information retrieval and decision-making across domains like finance, healthcare, and scientific research. This paper describes our system for the SemEval 2025 Task 8 on Question Answering over Tabular Data, specifically focusing on the DataBench QA and DataBench Lite QA subtasks. Our approach involves generating Python code using Large Language Models (LLMs) to extract answers from tabular data in a zero-shot setting. We investigate both multi-step Chain-of-Thought (CoT) and unified LLM approaches, where the latter demonstrates superior performance by minimizing error propagation and enhancing system stability. Our system prioritizes computational efficiency and scalability by minimizing the input data provided to the LLM, optimizing its ability to contextualize information effectively. We achieve this by sampling a minimal set of rows from the dataset and utilizing external execution with Python and Pandas to maintain efficiency. Our system achieved the highest accuracy amongst all small open-source models, ranking 1st in both subtasks.

1 Introduction

Tabular Question Answering (QA) is a critical area in Natural Language Processing (NLP) that enhances data accessibility and facilitates automated information extraction from structured datasets. The SemEval 2025 task on "Question Answering over Tabular Data" (Osés Grijalba et al., 2025) advances this field by introducing DataBench (Grijalba et al., 2024b), a benchmark comprising diverse, large-scale tabular datasets spanning multiple domains. The task challenges participants to develop systems capable of answering questions over these datasets, with two subtasks: DataBench QA (full datasets) and DataBench Lite QA (sampled datasets with a maximum of 20 rows) to support models with limited context windows. Expected answer types include boolean, category, number, or lists of these values.

	Accuracy	General	Open Models	Small Models
Databench	73.18%	25th	18th	1st
Databench Lite	73.75%	26th	18th	1st

Table 1: Performance on the DataBench QA and DataBench Lite QA subtasks, showing accuracy and rank among all systems (General), open-sourced models, and small open-sourced models ($\leq 8B$ parameters).

Our system employs a code generation approach using Python[1] and Pandas[2] to extract answers from tabular data using open-source Large Language Models (LLMs). Given a table and a question, our system generates executable code that loads the table, performs the necessary computations, and returns the answer. This approach ensures interpretability, as the reasoning process is encoded explicitly in the generated code, and it remains agnostic to the table structure. We explore both a multi-step agentic Chain-of-Thought (CoT) approach, where one LLM generates structured reasoning steps and another translates them into executable code, and a unified LLM approach, where the model simultaneously reasons and writes code. We prioritize a zero-shot setting to minimize the amount of table data passed to the LLM, optimizing for low resource consumption and scalability.

Our system achieved the highest accuracy among small open-sourced LLMs, ranking 1st in both subtasks (Table 1). Through our participation in this task, we discovered several key insights into optimizing LLM-driven code generation for tabular QA. First, prompt engineering plays a crucial role

*Equal contribution

[1]https://www.python.org/
[2]https://pandas.pydata.org

Proceedings of the The 19th International Workshop on Semantic Evaluation (SemEval-2025), pages 2197–2204
July 31 - August 1, 2025 ©2025 Association for Computational Linguistics

in improving performance - highlighting the importance of structuring inputs effectively to guide model reasoning. Second, we demonstrate that it is possible to achieve strong results while minimizing the amount of table data passed to the LLM. Notably, our results indicated that a unified LLM approach outperformed the agentic CoT method. We hypothesize that discrepancies in reasoning styles between different models led to cascading errors from the reasoning step to the final answer. This motivated our shift towards a unified system. Our approach relied on code execution, hence a major challenge was to ensure the generation of syntactically and semantically correct parsable code. We implemented iterative retry mechanisms that provided the LLM with error codes to refine its output. Our work is publicly available[3] for reproducibility.

2 Background

Our work is based on the dataset collection originally presented in the paper (Grijalba et al., 2024a). The dataset comprises 65 tables designed to evaluate LLMs on the task of QA over structured real-world tabular data.

2.1 Dataset Details

The dataset collection consists of 3,269,975 rows and 1,615 columns in total, covering a diverse range of domains such as business, health, travel, social networks, sports, and more. Across all datasets, a total of 1,300 questions were designed to evaluate QA performance. The number of questions per dataset varies, with the *Forbes* dataset containing 25 questions, while all other datasets contain 20 questions each. Every question in the train data can be answered by using ~ 1.45 columns. Table 2 summarizes the distribution of answer types.

Answer Type	Sample	Number of Questions
Boolean	True/False	262
Number	4, 10	260
Category	Automotive, United States	263
List[category]	[apple, mango]	261
List[number]	[2, 4, 6, 8, 10]	262

Table 2: Distribution of Answer Types in train data

2.2 Dataset Composition

The dataset includes a variety of sources, covering different domains and real-world scenarios. Table 3 presents an overview of the datasets used.

[3] GitHub Repo

Name	Rows	Cols	Domain	Source
Forbes	2,668	17	Business	Forbes
Titanic	887	8	Travel	Kaggle
Love	373	35	Social Networks	Graphext
Taxi	100,000	20	Travel	Kaggle
NYC Calls	100,000	46	Business	City of New York
London Airbnbs	75,241	74	Travel	Kaggle
Fifa	14,620	59	Sports	Kaggle
Tornados	67,558	14	Health	Kaggle
Central Park	56,245	6	Travel	Kaggle
ECommerce Reviews	23,486	10	Business	Kaggle

Table 3: Dataset Overview

2.3 Related Work

LLMs have significantly advanced automated code generation, particularly in the domain of zero-shot reasoning and structured prompting techniques. Traditional approaches to Verilog code generation, such as VRank (Zhao et al., 2025), emphasize self-consistency by clustering and ranking multiple generated candidates, thereby improving functional correctness. CoT prompting (Kojima et al., 2022) has demonstrated the effectiveness of reasoning through multi-step logical processes before generating outputs. However, CoT alone can struggle with syntax correctness, which CodeCoT (Huang et al., 2023) addresses by introducing a self-examination mechanism — iteratively refining outputs based on execution feedback.

AutoAgent (Tang et al., 2025) introduces a fully automated LLM-based framework that eliminates the need for manual intervention, enabling users to deploy intelligent agents through natural language alone. Similarly, SCoT prompting (Li et al., 2025) enhances CoT by explicitly incorporating program structures such as sequences, branches, and loops, achieving more structured and correct code outputs. Meanwhile, fine-tuning and prompting techniques such as those applied to Code Llama (Roziere et al., 2023) and GPT-based models (Haider et al., 2024) show that augmenting metadata, function call graphs, and iterative refinements can further optimize performance.

While LLMs excel in general-purpose code generation, their application to tabular data processing remains limited. TableGPT (Zha et al., 2023) proposes an approach for interacting with structured tables using natural language, offering functionalities like data manipulation, visualization, and analysis. However, its reliance on external functional commands and constrained dataset sizes limits its scalability for large-scale applications. This contrasts with the broader adaptability of agentic AI methods like AutoAgent, which can dynamically

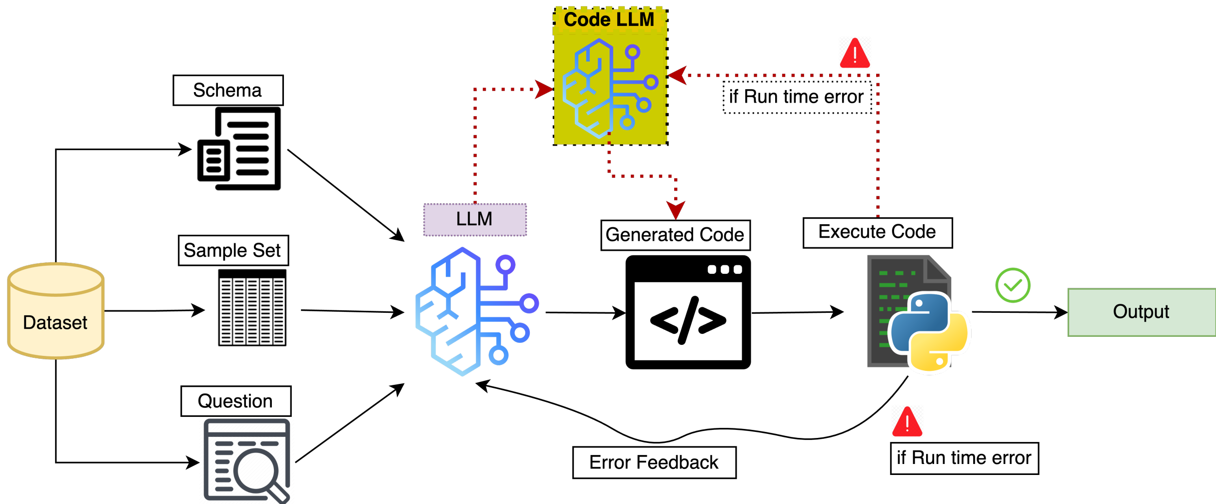

Figure 1: Flowchart illustrating the data preprocessing and model workflow for Subtasks 1 and 2 using a unified and an agentic approach. In the agentic setting the central 'LLM' turns into a 'reasoner LLM' which delineates steps for the 'Code LLM' to write code, which on execution feeds back the error codes to both the reasoner and code LLMs. This is illustrated by using red-dashed arrows in the figure.

generate and refine responses without predefined dataset limitations.

Recent works (Mullick et al., 2022b,a, 2023; Raghav et al., 2023) has also explored fine-grained task-specific adaptations of LLMs across various domains highlighting the importance of domain-aware prompting and structured reasoning. Similarly, generative techniques have been applied to structured tasks like sentiment analysis (Raghav et al., 2022).

In parallel, advancements in retrieval-augmented models and robustness enhancements through knowledge conflict augmentations (Carragher et al., 2025a,b) illustrate emerging techniques for improving generalization and resilience in large model architectures, many of which are transferable to structured tabular reasoning. Additionally, (Raghav et al., 2025) demonstrate the efficacy of large-scale instructive fine-tuning of LLMs – technique that bears promise for improving reasoning quality over structured data.

As LLMs continue to evolve, the integration of zero-shot reasoning, structured CoT prompting, agentic retries, and task-specific augmentations presents a promising direction for improving code generation and complex reasoning over diverse paradigms, including tabular QA.

3 System Overview

Our system Figure 1, reframes the tabular QA task as a code generation problem in zero-shot setting. Given an input dataset table D and a natural language question Q, the system follows a structured pipeline. First, the dataset schema is extracted, including column names, data types, and a sample of the first three rows. This information is then used to construct a structured prompt using the system-user-assistant format that provides essential context while reducing the amount of information provided to the LLM. The system proceeds to generate Python code designed to load the dataset and execute the necessary computations to extract the answer. Subsequently, the generated code is parsed for errors and executed to obtain the final output.

To enhance robustness, we incorporate an execution-aware retry mechanism. If the generated code encounters errors, the error message is fed back to the LLM, allowing it to iteratively refine its output (up to 3 retries). This significantly improves the accuracy of generated code and reduces failure.

By reducing the input size, we optimize both computational efficiency and scalability, while also improving the LLM's ability to effectively contextualize the given data. Additionally, because our system follows a code generation approach and provides only the essential information about the tables, it can be seamlessly applied to both subtasks without requiring any modifications. This design ensures adaptability across different task configurations, further enhancing the system's versatility.

3.1 Agentic CoT Approach

In this approach, we use two separate LLMs: one dedicated to reasoning and another responsible

for code generation. The first LLM decomposes the given question into structured reasoning steps, breaking it down into logical sub-components. These steps are then passed to the second LLM, which translates them into executable Python code.

We conduct experiments with combinations of LLMs for both reasoning and code generation. Specifically, we explore the following model pairs:

- LLaMA 3.1 (8B Instruct) (Touvron et al., 2023) + CodeLLaMA (7B) (Roziere et al., 2023)
- LLaMA 3.1 (8B Instruct) + Phi-4 (8.48B) (Abdin et al., 2024)
- Phi-4 8.48B + CodeLLaMA (7B)

where the first model in each pair was responsible for reasoning and the second for generating executable code. These configurations allowed us to evaluate the impact of model specialization on logical coherence and code correctness.

While the agentic CoT approach provides explicit reasoning traces, discrepancies between the reasoning and code generation LLMs led to cascading errors. CodeLLaMA often struggled with syntax errors or misinterpreted reasoning instructions, while LLaMA failed to generate accurate reasoning steps, resulting in flawed code. Although larger models might have improved performance, computational constraints forced us to use smaller versions. This motivated us to explore a unified approach using a single LLM.

3.2 Unified LLM Approach

To overcome the limitations of the CoT approach, we developed a streamlined pipeline using a single LLM to jointly perform reasoning and code generation. We hypothesized that this method should improve consistency and eliminate the error propagation observed in the multi-step approach.

We experimented with several LLMs, including LLaMA, CodeLLaMA, and Phi-4. Our initial hypothesis was that LLaMA's reasoning capabilities would enhance its code generation, whereas CodeLLaMA would exhibit the inverse relationship. However, Phi-4 demonstrated superior consistency by effectively handling both reasoning and code generation within a single inference pass. This proved more adaptable to diverse question types and table structures, resulting in more robust performance across different datasets.

3.3 Challenges and Solutions

We encountered several challenges in this task:

- **Generating Correct and Parsable Code:** Ensuring the syntactic and semantic correctness of generated code remains a significant challenge. To address this, we employ an error feedback loop, where execution-triggered error messages are used as signals for iterative refinement, guiding the LLM toward producing correct outputs. We allow up to **three retries** within this loop: most simple syntax or small logical errors are typically corrected within the first two attempts, while errors persisting beyond three retries are rarely resolved with additional attempts, as they often stem from fundamental limitations of the model (e.g., ambiguous prompts, missing dependencies, or deeper logical flaws). To validate this choice, we conducted a small experiment by sampling examples that failed even after three retries and allowing them up to ten retries; however, only $\sim 2\%$ of these cases succeeded, confirming that the gains diminish sharply beyond three retries.
- **Handling Large Tables:** It is hard for LLMs to reason on long and wide tables. Instead of providing full datasets, we sample the first three rows along with the schema information, ensuring that the system receives sufficient context with minimal tokens.
- **Ensuring Robust Answer Formatting:** As the task requires specific output data type, we enforce strict formatting constraints on the generated code's output through our prompts.
- **Prompt Engineering:** We adopted a System-User-Assistant chat template as it yielded the best results for our task. The *System* message included the task introduction and LLM conditioning to guide it through the expertise it would need, while the *User* message contained the dataframe schema, sample rows and generation instructions. A detailed illustration of the prompt can be found at subsection A.1 and in our code repository.

4 Experimental Setup

Our experimental setup is designed to evaluate the effectiveness of our approach across both Subtasks 1 and 2. We focus on zero-shot prompting using

quantized LLMs, leveraging prompt engineering techniques to optimize performance. Our methodology incorporates the system-user-assistant chat template, ensuring structured interactions with LLMs. To facilitate efficient inference, we employ Unsloth's (Daniel Han and team, 2023) dynamically quantized 4-bit versions of these models, allowing for computationally efficient execution while maintaining strong performance. We used dymamic 4-bit quantized LLaMA 3.1 (8B Instruct)[4], CodeLLaMA (7B)[5] and dynamic 4-bit quantized Phi-4 (8.48B)[6] models from Unsloth. Experiments were conducted on a single NVIDIA T4 GPU using Google Colab and Kaggle, emphasizing the feasibility of achieving high performance with limited compute resources. Predictions were generated on the validation and blind test set.

4.1 Evaluation Function

The evaluation function for this task measures accuracy while allowing for minor variations in formatting. This ensures that models are not overly penalized for trivial differences in output representation.

For boolean values, the function accepts different valid representations such as "true/false" or "yes/no." In the case of categorical outputs, string matching is applied, while date values undergo parsing to check equivalence. Numerical outputs are evaluated by extracting relevant digits and rounding to two decimal places. List-based outputs are assessed using a set comparison approach, allowing for minor differences in ordering. The evaluation function has been iteratively refined during the competition to be more lenient while maintaining robustness. A manual review of leading systems was conducted before final ranking to ensure fair assessment and identify discrepancies that automated evaluation may overlook.

5 Results

Our system demonstrated exceptional performance, ranking 1st in the "Small Open-Source Models" category ($\leq 8B$ parameters) (refer to Table 1). The single LLM approach consistently outperformed the CoT method, highlighting the importance of maintaining logical consistency within a single model. Additionally, our optimizations in prompt engineering and execution-aware retry mechanisms significantly improved efficiency and accuracy, making our approach well-suited for real-world tabular QA tasks. Detailed results and ablation studies can be found in Table 4.

Model	η	DataBench	DataBench Lite
LLaMA + CodeLLaMa	**1**	**0.13**	0.17
	3	0.10	**0.18**
	5	0.10	0.15
	7	0.10	0.14
LLaMA + Phi-4	**1**	**0.21**	**0.26**
	3	0.20	0.26
	5	0.20	0.25
	7	0.19	0.25
Phi-4 + CodeLLaMA	1	0.29	0.31
	3	**0.32**	**0.34**
	5	0.32	0.34
	7	0.31	0.32
LLaMA	**1**	**0.50**	**0.52**
	3	0.48	0.50
	5	0.49	0.51
	7	0.50	0.49
CodeLLaMA	1	0.55	0.57
	3	**0.55**	**0.57**
	5	0.54	0.57
	7	0.55	0.55
Phi-4	1	0.70	0.70
	3	**0.71**	**0.72**
	5	0.71	0.70
	7	0.69	0.70

Table 4: Ablation study on the validation dataset to decide on the ideal number of rows to be provided to the model. η is the number of rows chosen from the dataset which is sampled and provided to the model. We choose η based on the performance on both the datasets

5.1 Key Findings

Our key findings through our ablations (Table 4) include:

- The single LLM approach consistently outperformed the CoT method, highlighting the benefits of maintaining logical consistency within a unified model.
- Consolidating reasoning and code generation into a single inference step reduced error propagation and improved system stability.
- Minimizing the amount of table data passed to the LLM optimized computational efficiency, while maintaining performance.
- Optimizations in prompt engineering and execution-aware retry mechanism contributed to improvements in both speed and accuracy.

[4] https://huggingface.co/unsloth/Meta-Llama-3.1-8B-Instruct-unsloth-bnb-4bit

[5] https://huggingface.co/unsloth/codellama-7b-bnb-4bit

[6] https://huggingface.co/unsloth/phi-4-unsloth-bnb-4bit

- Our system is highly adaptable to diverse datasets, ensuring robust performance across a wide range of tabular QA tasks.
- Although there are five distinct types of possible answers, providing more than three examples does not yield any performance improvement. We hypothesize that this behavior arises from the limited context window available to large language models (LLMs) to handle long prompt templates.
- In particular, we observe that LLaMA-based models tend to perform slightly better with shorter prompts, and the Phi-4 model demonstrates a stronger ability to handle longer prompts.

5.2 Error Analysis

Error analysis revealed several recurring issues. CodeLLaMA, when used in isolation for code generation, often produced syntactically incorrect or semantically flawed code. This was particularly evident in cases with complex queries requiring intricate data manipulations. LLaMA struggled with reasoning tasks, which led to incomplete or inaccurate code generation. These findings reaffirmed our decision to use a single LLM approach, which alleviated many of these issues by ensuring consistency between reasoning and code generation.

Furthermore, retry mechanisms were essential for handling edge cases and failed executions. In future work, we plan to explore fine-tuning techniques to improve the handling of more complex queries and further reduce the occurrence of errors in generated code.

6 Conclusion

Our system demonstrates the effectiveness of LLM-driven code generation for zero-shot question answering on tabular data. We highlight the advantages of a unified LLM approach for maintaining logical consistency and the importance of prompt engineering for guiding model reasoning effectively. Additionally, we emphasize the benefits of minimizing LLM input for improved contextualization, computational efficiency, and scalability. Further exploration of fine-tuning techniques could improve the generation of complex aggregation queries and reduce errors in generated code.

Agentic systems have greater potential than what has been portrayed in this work. While our current system demonstrates the effectiveness of LLMs for tabular QA, we recognize that further engineering is needed to fully exploit the potential of agentic systems in this domain. Future research will focus on refining the agentic framework to enable more complex reasoning and data manipulation capabilities.

References

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. 2024. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*.

Peter Carragher, Abhinand Jha, R Raghav, and Kathleen M Carley. 2025a. Quantifying memorization and retriever performance in retrieval-augmented vision-language models. *arXiv preprint arXiv:2502.13836*.

Peter Carragher, Nikitha Rao, Abhinand Jha, R Raghav, and Kathleen M Carley. 2025b. Koala: Knowledge conflict augmentations for robustness in vision language models. *arXiv preprint arXiv:2502.14908*.

Michael Han Daniel Han and Unsloth team. 2023. Unsloth.

Jorge Osés Grijalba, L Alfonso Urena Lopez, Eugenio Martínez-Cámara, and Jose Camacho-Collados. 2024a. Question answering over tabular data with databench: A large-scale empirical evaluation of llms. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 13471–13488.

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

Md Asif Haider, Ayesha Binte Mostofa, Sk Sabit Bin Mosaddek, Anindya Iqbal, and Toufique Ahmed. 2024. Prompting and fine-tuning large language models for automated code review comment generation. *arXiv preprint arXiv:2411.10129*.

Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Codecot: Tackling code syntax errors in cot reasoning for code generation. *arXiv preprint arXiv:2308.08784*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23.

Ankan Mullick, Ishani Mondal, Sourjyadip Ray, Raghav R, G Chaitanya, and Pawan Goyal. 2023. Intent identification and entity extraction for healthcare queries in Indic languages. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 1870–1881, Dubrovnik, Croatia. Association for Computational Linguistics.

Ankan Mullick, Abhilash Nandy, Manav Kapadnis, Sohan Patnaik, Raghav R, and Roshni Kar. 2022a. An evaluation framework for legal document summarization. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, pages 4747–4753, Marseille, France. European Language Resources Association.

Ankan Mullick, Abhilash Nandy, Manav Nitin Kapadnis, Sohan Patnaik, and R Raghav. 2022b. Fine-grained intent classification in the legal domain. *arXiv preprint arXiv:2205.03509*.

Jorge Osés Grijalba, L. Alfonso Ureña-López, Eugenio Martínez Cámara, 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)*, pages 1015–1022, Vienna, Austria. Association for Computational Linguistics.

R Raghav, Jason Rauchwerk, Parth Rajwade, Tanay Gummadi, Eric Nyberg, and Teruko Mitamura. 2023. Biomedical question answering with transformer ensembles. In *CLEF (Working Notes)*.

R Raghav, Adarsh Vemali, and Rajdeep Mukherjee. 2022. Etms@ iitkgp at semeval-2022 task 10: Structured sentiment analysis using a generative approach. In *Proceedings of the 16th International Workshop on Semantic Evaluation (SemEval-2022)*, pages 1373–1381.

R Raghav, Adarsh Prakash Vemali, Darpan Aswal, Rahul Ramesh, Parth Tusham, and Pranaya Rishi. 2025. Tartantritons at semeval-2025 task 10: Multilingual hierarchical entity classification and narrative reasoning using instruct-tuned llms. In *Proceedings of the 19th International Workshop on Semantic Evaluation*, SemEval 2025, Vienna, Austria.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Jiabin Tang, Tianyu Fan, and Chao Huang. 2025. Autoagent: A fully-automated and zero-code framework for llm agents. *arXiv e-prints*, pages arXiv–2502.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Liangyu Zha, Junlin Zhou, Liyao Li, Rui Wang, Qingyi Huang, Saisai Yang, Jing Yuan, Changbao Su, Xiang Li, Aofeng Su, et al. 2023. Tablegpt: Towards unifying tables, nature language and commands into one gpt. *arXiv preprint arXiv:2307.08674*.

Zhuorui Zhao, Ruidi Qiu, Ing-Chao Lin, Grace Li Zhang, Bing Li, and Ulf Schlichtmann. 2025. Vrank: Enhancing verilog code generation from large language models via self-consistency. *arXiv preprint arXiv:2502.00028*.

A Appendix

A.1 Prompt Template

We present the prompt template which follows the system-user-assistant chat paradigm. The same template was used for both Databench and Databench Lite datasets.

System

```
You are an expert Python data engineer.
Your task is to generate pandas code based on a structured reasoning process
You only generate code, no references or explanation - just code
You generate only 20 lines of code at max
```

User

```
Dataframe Schema:
<The schema of the dataset which would contain the column name and its data type>


Sample Rows:
<The first 3 rows of the dataset serialized into a list of dictionaries, where each dictionary
represents a row with column names as keys>

User Question:
<The question that is asked about the dataset>

Expected Output Format:
Generate runnable Python code that follows the given reasoning using pandas.
The code should assume that the dataframe is already loaded as `df`.
The final output should be stored in a variable named `result`.

The expected answer type is unknown, but it will always be one of the following:
    * Boolean: True/False, "Y"/"N", "Yes"/"No" (case insensitive).
    * Category: A value from a cell (or substring of a cell) in the dataset.
    * Number: A numerical value from a cell or a computed statistic.
    * List[category]: A list of categories (unique or repeated based on context). Format: ['cat',
    'dog'].
    * List[number]: A list of numbers.

Given the user question, you need to write code in pandas, assume that you already have df.
Generate only the code.
```

The assistant prompt was left blank during inference. The code obtained from the model would then be run on the dataset to evaluate the answer to the question.