AlexNLP-MO at SemEval-2025 Task 8: A Chain of Thought Framework for Question-Answering over Tabular Data

Omar Mokhtar, Minah Ghanem, Nagwa ElMakky

Computer and Systems Engineering Department
Alexandria University
{es-omar.mokhtar2019, es-Minah.Sayed2020, nagwamakky}@alexu.edu.eg

Abstract

Table Question Answering (TQA) involves extracting answers from structured data using natural language queries, a challenging task due to diverse table formats and complex reasoning. This work develops a TQA system using the DataBench dataset, leveraging large language models (LLMs) to generate Python code in a zero-shot manner. Our approach is highly generic, relying on a structured Chain-of-Thought framework to improve reasoning and data interpretation. Experimental results demonstrate that our method achieves high accuracy and efficiency, making it a flexible and effective solution for real-world tabular question answering. The source code for our implementation is available on GitHub [1].

1 Introduction

As tabular data becomes increasingly prevalent across domains such as finance, healthcare, and scientific research, there is a growing need for systems that can effectively interpret and extract information from structured data using natural language queries. Table Question Answering (TQA) addresses this challenge by enabling users to query tables without requiring a structured query languages like SQL. However, TQA remains a difficult task due to the diversity in table structures, ambiguous question formulations, and the need for numerical and logical reasoning.

In this paper we present a full pipeline for TQA. We prompt large language models (LLMs) to generate executable Python code dynamically. This method allows for flexible reasoning over tabular data without requiring task-specific training. Additionally, we introduce a structured Chain-of-Thought (CoT) framework that enhances the model's interpretability by decomposing reasoning into explicit Hint Generation and Code Generation steps.

To make our framework as dynamic as possible, we performed extensive prompt tuning to adapt to variations in table structures and data distributions. Since tabular data can change significantly across different domains, we iteratively refined our prompts to ensure robustness across datasets. We also analyzed errors to identify common mistakes and adjusted our approach accordingly. Additionally, we conducted a grid search over model parameters to optimize performance, balancing computational efficiency and accuracy. These optimizations allow our system to generalize effectively across diverse TQA scenarios.

We evaluate our approach on the DataBench dataset (Grijalba et al., 2024), a diverse benchmark featuring real-world tables and human-generated queries. Our results demonstrate that our method effectively handles a wide range of question types, including direct retrieval, numerical reasoning, and categorical filtering, while maintaining computational efficiency.

2 Related Work

Recent advances in TQA typically rely on two main approaches: Text-to-SQL models (Zhong et al., 2017; Yu et al., 2019), which translate natural language into executable queries, and end-to-end (E2E) models (Yin et al., 2020), which directly infer answers from table representations. While Text-to-SQL excels in structured retrieval and arithmetic reasoning, it struggles with unstructured tables and ambiguous queries. E2E models are more flexible but often lack precision in structured data retrieval. Hybrid approaches (Zhang et al., 2024a) have attempted to combine these strengths, but they typically require extensive pretraining and fine-tuning, limiting their adaptability across datasets.
In parallel, recent work has explored enhancing

[1] https://github.com/omarmoo5/semeval2025-task8

these models through Chain-of-Thought (CoT) reasoning techniques. Building on this line of research, the Chain-of-Table framework (Wang et al., 2024) introduces a novel method for table-based reasoning by integrating structured tabular transformations into the reasoning process of Large Language Models (LLMs). It enables dynamic table evolution, where each reasoning step applies operations such as adding columns, selecting rows, grouping, or sorting. These operations create intermediate tables that serve as proxies for the model's thought process.

Despite these advances, existing methods often require pretraining, fine-tuning, or introduce significant computational overhead. To address these limitations, we propose a zero-shot TQA approach that eliminates the need for any model-specific training. Our method leverages the Chain-of-Thought (CoT) framework to improve interpretability and overall system performance. The resulting approach is lightweight and easily adaptable across different datasets.

3 Task Overview

The main objective of the shared task (Osés Grijalba et al., 2025) is to develop a system capable of accurately answering questions based on real-world datasets presented in tabular formats.
The dataset utilized in this task is DataBench (Grijalba et al., 2024), a diverse collection of tabular datasets presented in English. It comprises 65 tables from various domains.
Each table is accompanied by 20 human-generated questions, resulting in a total of 1,300 questions and answers.
The task includes two subtasks:

- **Task I:** DataBench QA. Answer the questions using only the data provided in the dataset.

- **Task II:** DataBench Lite QA. Similar to Task I but uses a sampled version of each dataset (maximum of 20 rows per dataset), which is particularly useful for evaluating models with smaller input window sizes.

4 Approach

Our approach is primarily code-based, building upon the baseline model (Grijalba et al., 2024). We leverage LLMs by prompting them to generate executable Python code snippets for answering questions over tabular data.

In our initial experiments, the LLaMA-3 model served as the baseline, tested using a simple prompt (Figure 5 & 6) to evaluate its raw performance. Due to computational limitations, the model weights were quantized to 4 bits.
The baseline analysis revealed several challenges, detailed in the Experiments section. Specifically, the model struggled with non-intuitive data units, often misinterpreting values in the thousands, and lacked awareness of column values, leading to incorrect filtering and null outputs. To address these issues, we introduced several improvements:

- **Meta-Information Enrichment:** Prompts were enhanced by providing detailed meta-information, including all possible categories for categorical columns and statistical summaries (mean, minimum, and maximum) for numerical columns. This helped the model better interpret the data.

- **Chain-of-Thought (CoT) Paradigm:** Inspired by CoT reasoning, we divided the workflow into two distinct modules:
  - **Hints Generation:** predicts two key aspects: the expected answer type and the relevant columns needed to answer the question.
  - **Code Generation:** Takes the generated hints, along with the meta-information, and prompts the LLM to generate Python code capable of querying the dataframe and extracting the correct answer.

- **Post-Processing Module:** To further improve robustness, we introduced a post-processing step. If the generated code fails to execute, an automatic retry with an additional LLM call is triggered. This module also refines outputs—for instance, by converting a Pandas Series to a list to ensure type consistency with the ground truth.

This structured, multi-step approach, illustrated in Figure 1, significantly improved the model's reasoning abilities and produced more accurate and interpretable outputs.

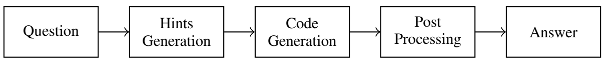


Figure 1: Final Approach

5 Experiments

5.1 Experiment 1: Enhancing the Baseline Prompt

In our initial attempts, we used the **Meta-Llama-3.1-8B-Instruct** model and modified the baseline prompt to provide more detailed information about the dataset. Specifically: For categorical columns, we listed all possible categories. For numerical columns, we included key statistics such as the mean, minimum, and maximum values. This enhancement improved the model's understanding of the dataset and helped prevent errors such as incorrect value indexing.

Example:

Question: How many billionaires are there from the 'Technology' category?

Part of the Generated Code:

```
billionaires = df[df['finalWorth']
>= 100000000]
```

However, in the dataset, 100000000 is expressed as 100 million, leading to potential indexing mistakes.

5.1.1 Experiment 2: Evaluating Alternative Models

To further improve performance, we tested other models using the same prompt structure. Specifically, we experimented with **Qwen2.5-Coder-7B-Instruct** (Hui et al., 2024), which significantly improved the results.
We also attempted to use **TableLLAMA** (Zhang et al., 2024b), but it performed poorly, as it frequently hallucinated answers rather than retrieving them directly from the dataset. Unlike other models, TableLLAMA takes the table as input and generates an answer without explicitly retrieving values, making evaluation and output constraints more challenging.

5.2 Experiment 3: Adding a Hint Module for Answer Type Prediction

To refine the model's responses, we introduced a hints module (Figure 9) to enrich the input prompt. The first hint required the model to predict the expected answer type, which was well-defined in the competition:

- `list[number]`
- `category`
- `number`
- `boolean`
- `list[category]`

Then the expected type is then passed to a code generation prompt. This adjustment helped prevent errors where the model focused too much on the logic of the question rather than retrieving the actual answer.

Example:

Question: Is the city with the most billionaires in the United States?

Incorrect Answer: Pottsville

By guiding the model with an answer-type constraint as shown in Figure 10, we reduced such errors.

5.3 Experiment 4: Building Upon Hints Module

To further improve accuracy, we introduced a second hint that predicted the relevant columns in the data frame needed to answer each question (Figure 12). This strategy narrowed the search space, helping the model focus on the most relevant data and reducing errors.

6 Results

All experiments were conducted on a V100 GPU (32GB memory), with all models quantized to 4 bits to fit within memory constraints.

For each experiment, we tested multiple temperature settings to assess their impact on performance.

The experiments were evaluated using the databench eval package.

6.1 Experiment 1:

Temperature	0.4	0.5	0.6	0.7	0.8
Full Dataset	0.5290	**0.5405**	0.5237	0.5366	0.5054
Lite Dataset	**0.5489**	0.5382	0.5175	0.5428	0.5306

Table 1: Results of Experiment 1.

6.2 Experiment 2:

Temperature	0.4	0.5	0.6	0.7	0.8
Full Dataset	**0.6758**	0.6758	0.6651	0.6689	0.6643
Lite Dataset	**0.6941**	0.6865	0.6827	0.6766	0.6766

Table 2: Results of Experiment 2.

6.3 Experiment 3:

Temperature	0.4	0.5	0.6	0.7	0.8
Full Dataset	**0.7000**	**0.7000**	0.6957	0.6888	0.6957
Lite Dataset	**0.7056**	0.7049	0.7000	0.6918	0.6972

Table 3: Results of Experiment 3.

6.4 Experiment 4:

Temperature	Full Dataset	Lite Dataset
0.1	0.7484	0.7629
0.2	0.7454	**0.7691**
0.3	0.7500	0.7660
0.4	0.7500	0.7637
0.5	0.7400	0.7607
0.6	0.7484	0.7610
0.8	0.7385	0.7599
0.9	**0.7561**	0.7614
1.0	0.7431	0.7607

Table 4: Results of Experiment 4.

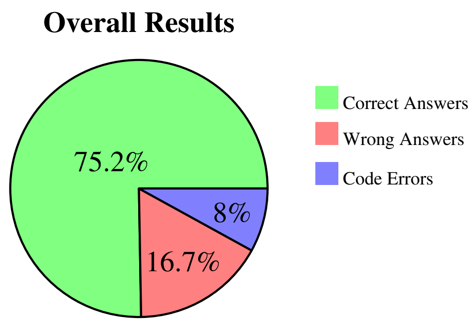
7 Results and Analysis

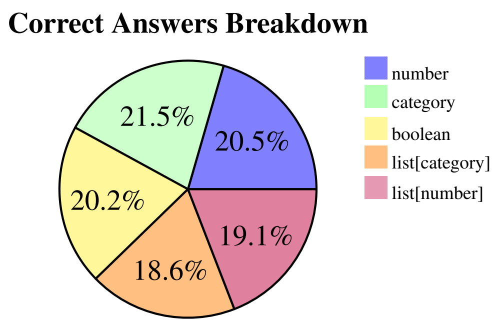
The analysis in the table below is conducted using Experiment 4 with a temperature of 0.7 on the dev set. The results were obtained by using the databench eval package.

Category	**Full**	**Lite**
Avg Accuracy	0.75225	0.77824
Boolean	0.75954	0.79389
Number	0.77692	0.78462
Category	0.80608	0.80989
List[Category]	0.70115	0.73946
List[Number]	0.71756	0.76336

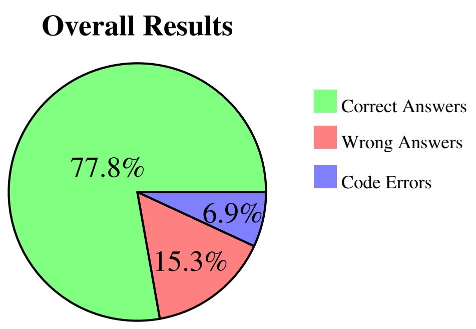
Table 5: Accuracy of Predicting Different Question Types for Full and Lite Datasets

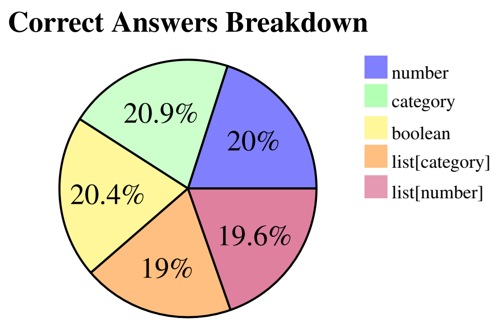
7.1 Results on the Full Dataset (Dev)





7.2 Results on the Lite Dataset (Dev)





7.3 Results on the Test Dataset

On the test dataset, we used larger models. We made two submissions as follows:

Model	**Full**	**Lite**
Qwen2.5-Coder(14B) Q4	71.64	75.28
Qwen2.5-Coder(32B) Q4	75.67	78.73

Table 6: Accuracy on the test Dataset

7.4 Error Analysis

In this section, we present an error analysis of our final pipeline, evaluated using the model **Qwen2.5-Coder-32B-Instruct** on the training dataset with a temperature setting of **0.7**. Due to computational constraints, we were unable to conduct a large number of experiments with this model.

There are multiple reasons that we identified for the errors:

7.4.1 Data cleaning issues

Some tables, such as **054_Joe** and **007_Fifa**, have columns renamed with type annotations (e.g., **column_name<gx:data_type>**). However, the LLM sometimes incorrectly omitted the datatype suffix during code generation, leading to code failures. An example from the **054_Joe** table is shown below (Figure 2).

```
{
    "Question": "What␣are␣the␣two␣
        highest␣numbers␣of␣retweets␣a␣
        tweet␣in␣the␣dataset?",
    "Table": "054_Joe",
    "Generated␣Code":
    def answer(df):
        return df['retweets'].nlargest
            (2).tolist(),
    "Answer": "Code␣Error",
    "Ground␣Truth": [205169, 101314]
}
```

Figure 2: Example of Data Cleaning Error

In this case, the column should have been referenced as **retweets<gx:number>** instead of **retweets**, resulting in a code failure. These types of errors require further investigation of the answer tables and additional cleaning to remove such inconsistencies.

7.4.2 Limitations in Handling Corner Cases Requiring Multiple Steps of Reasoning

Some questions involved corner cases that required multiple steps of reasoning, which the LLMs struggled to handle. One such issue occurred in table **046_120**, where the same athlete appeared multiple times, leading to incorrect results in the top 3 weights (Figure 3).

```
{
    "Question": "What␣are␣the␣three␣
        highest␣weights␣of␣athletes?",
    "Table": "046_120",
    "Generated␣Code":
    def answer(df):
        return df['Weight'].nlargest(3).
            tolist(),
    "Answer": [214.0, 214.0, 198.0],
    "Ground␣Truth": [214.0, 198.0,
        190.0]
}
```

Figure 3: Example of Handling Corner Cases Errors

This problem arises because the LLM does not account for duplicate entries of the same athlete, resulting in multiple counts of the same weight. This error in such questions could be mitigated through few-shot training, which encourages the model to consider all possible scenarios, or by adding an additional layer for Chain of Thought (CoT) while solving the question

7.4.3 Logic Code Errors

There also exist some logical code errors. In the case of table **002_Titanic**, the generated code failed due to an incorrect use of the **.any()** function (Figure 4).

```
{
    "Question": "Did␣any␣children␣below␣
        the␣age␣of␣18␣survive?",
    "Table": "002_Titanic",
    "Generated␣Code":
    def answer(df):
        return (df['Age'] < 18) & (df['
            Survived']).any(),
    "Answer": [False, False, False,
        False, False, False, False,
        False, True, False, False, False
        , False, False, False, True,
        False, False, False, False],
    "Ground␣Truth": True
}
```

Figure 4: Example of Logic Code Error in Titanic Dataset

The issue arises because the **.any()** function is applied to the **df['Survived']** column only, rather than evaluating the condition for each row. This leads to an incorrect output. To reduce such errors in the future, the model code generation capabilities must be improved.

8 Conclusion

In this work, we explored multiple LLMs and experimented with various prompt modifications. Our findings indicate that enhancing prompts with richer data context significantly improves the model's understanding of the dataset and its values. Additionally, decomposing the problem into smaller tasks—such as predicting the answer type, identifying relevant columns, and then integrating this information to generate the final code—proved to be more effective than attempting to generate the code in a single step.

This structured approach helped us improve upon baseline results by reducing errors and enhancing accuracy. Furthermore, systematic data analysis and tracking model errors played a crucial role in identifying the model's weaknesses, allowing us to refine our approach and achieve better performance.

Our final system achieved a score of **79.31** in the final ranking of the *full DataBench* test set, ranking **24th overall** and **16th among open-source models**. On the *DataBench Lite* test set, our system ranked **17th overall** and **11th among open-source models**. Notably, these results were obtained using a **quantized version of the LLM**. Given these results, we believe that running our approach on a full-precision model could further enhance accuracy by leveraging richer representations and reducing potential quantization-related losses. This highlights the adaptability of our framework and its potential to achieve even stronger performance with greater computational resources.

9 Future Work

In the future, we plan to explore the use of larger models to further improve our results. Due to resource constraints, we were unable to use larger models and instead relied on quantized versions. We strongly believe that scaling up to larger models would significantly enhance performance.

Additionally, adopting a more interactive approach to analyze model errors and dynamically modify prompts could lead to better outcomes. Another area for exploration involves testing more models, particularly recent ones such as DeepSeek. Although we attempted to use this model, we encountered challenges related to constraining the generated output, which required parsing effort during the development phase. Further efforts will be necessary to refine and optimize the use of this model for improved results.

References

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.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Jorge Osés Grijalba, L. Alfonso Uref́1-Lf́3pez, Eugenio Martédnez Cé1mara, 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.

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*.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 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, Online. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2019. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Preprint*, arXiv:1809.08887.

Siyue Zhang, Anh Tuan Luu, and Chen Zhao. 2024a. Syntqa: Synergistic table-based question answering via mixture of text-to-sql and e2e tqa. *arXiv preprint arXiv:2409.16682*.

Tianshu Zhang, Xiang Yue, Yifei Li, and Huan Sun. 2024b. Tablellama: Towards open large generalist models for tables. *Preprint*, arXiv:2311.09206.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *Preprint*, arXiv:1709.00103.

10 Appendix

```
{
    "role": "system",
    "content": "You␣are␣a␣helpful␣
        assistant␣and␣an␣expert␣in␣
        Python␣programming."
}
```

Figure 5: Exp1 - System Message

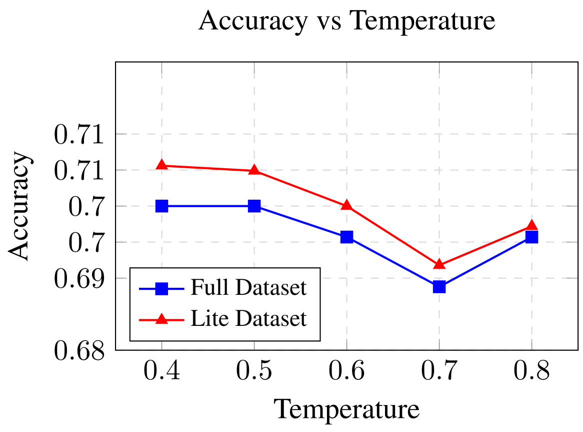


Figure 11: Experiment 3 Results

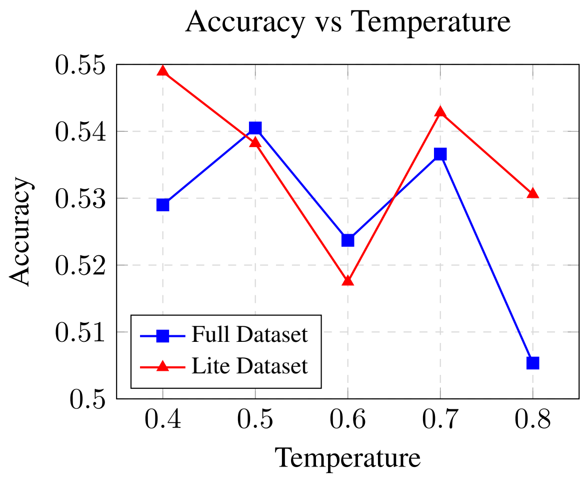


Figure 7: Experiment 1 Results

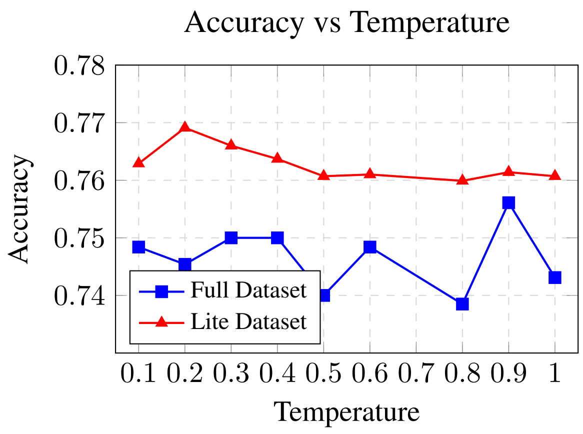


Figure 13: Experiment 4 Results

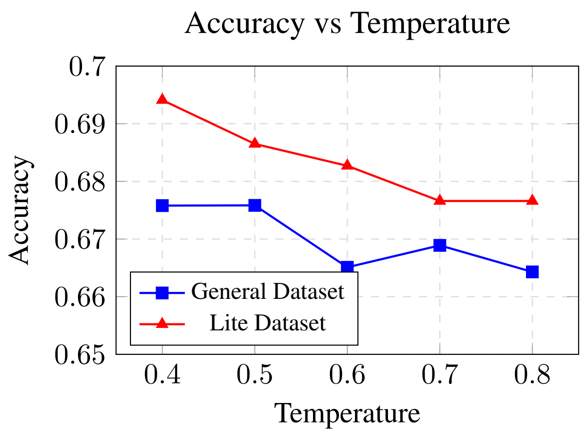


Figure 8: Experiment 2 Results

```
{
    "role": "user",
    "content": (
    "**python\n"
    "import␣pandas␣as␣pd\n"
    "import␣numpy␣as␣np\n\n"
    "def␣answer(df)␣->␣bool:\n"
    "␣␣'**\n"
    "␣␣␣The␣DataFrame␣('df')␣has␣the␣following␣dtypes:\n"
    f"␣␣(meta)"
    "␣␣\n\n"
    f"␣␣Returns:␣{question}\n"
    "␣␣'**\n"
    "␣␣␣#␣Sample␣data␣rows␣for␣insights:\n"
    f"␣␣data␣=␣{data}"
    "***\n"
    "Return␣only␣the␣Python␣function␣implementation␣only␣as␣a␣single␣code␣block.␣Don
        't␣write␣comments␣or␣docstrings."
}
```

Figure 6: Exp1 - Baseline User Message

```
    """
    You are Qwen, created by Alibaba Cloud. You are a helpful assistant.
    Given a question you are requested to predict the type of answer from this list:
    [list[number], category, number, boolean , list[category]]
    return only the predicted type of the answer.
    """
```

Figure 9: Experiment 3 Hint Generation Prompt

```
# Implement a Python function only as a single code block. Don't write comments or
    docstrings.

def answer(df):
    '''
    Processes a DataFrame and returns the answer based on the given question and
        data types.

    Args:
        df (pd.DataFrame): The input DataFrame with specific dtypes: {meta}.
            It contains some categorical data in columns: {categorical_columns}.

    Returns: {return_type}
    Make sure to satisfy the following conditions:
            - Boolean: Returns either True or False.
            - Category: Returns a value from a cell (or a substring of a cell) in
                the dataset.
            - Number: Returns a numerical value from a cell in the dataset, which
                may represent a computed statistic (e.g., average, maximum, minimum)
                .
            - List[category]: Returns a list containing a fixed number of categories
                .
            - List[number]: Returns a list containing a fixed number of numerical
                values.

    The function returns the result for the following question: {question}.
    '''
```

Figure 10: Experiment 3 Code Generation Prompt

```
"""
You are a helpful assistant. Given a question you are requested to Follow this
    instructions strictly:

- Predict the type of answer to be literally one of this :
[list[number], category, number, boolean , list[category]]
    * Boolean: Returns either True or False.
    * Category: Returns a value from a cell (or a substring of a cell) in the
        dataset.
    * Number: Returns a numerical value from a cell in the dataset, which may
        represent a computed statistic (e.g., average, maximum, minimum).
    * List[category]: Returns a list containing a fixed number of categories.
    * List[number]: Returns a list containing a fixed number of numerical values
        .
Don't use any other type of answer.

- Predict the relevant column names neeeded to answer the question using the
    metadata of the table.

Return only a json in this format:

{
"type": <predicted type of answer>,
"columns_used": <list of columns to be used to answer>
}

"""
```

Figure 12: Exp4 - More Hints Generation Prompt