QleverAnswering-PUCRS at SemEval-2025 Task 8: Exploring LLM agents, code generation and correction for Table Question Answering

André Bergmann Lisboa, Lucas Cardoso Azevedo, and **Lucas R. C. Pessutto**
School of Technology – PUCRS – Porto Alegre – Brazil
andre.bergmann@edu.pucrs.br, lucas.azevedo96@edu.pucrs.br,
lucas.pessutto@pucrs.br

Abstract

Table Question Answering (TQA) is a challenging task that requires reasoning over structured data to extract accurate answers. This paper presents QleverAnswering-PUCRS, our submission to SemEval-2025 Task 8: DataBench, Question-Answering over Tabular Data. QleverAnswering-PUCRS is a modular multi-agent system that employs a structured approach to TQA. The approach revolves around breaking down the task into specialized agents, each dedicated to handling a specific aspect of the problem. Our system was evaluated on benchmark datasets and achieved competitive results, ranking mid-to-top positions in the SemEval-2025 competition. Despite these promising results, we identify areas for improvement, particularly in handling complex queries and nested data structures.

1 Introduction

The rapid growth of structured data across various domains has increased the demand for automated systems capable of extracting and interpreting information from tabular data. TQA is a crucial Natural Language Processing (NLP) task that focuses on generating accurate answers to factual questions using structured information (Jin et al., 2022). Unlike traditional QA systems that rely on free-text corpora, TQA systems must directly retrieve relevant information from relational tables, spreadsheets, or structured databases. Additionally, the limited context window of Large Language Models (LLMs) constrains the amount of tabular data that can be processed in a single prompt, making metadata extraction a key component for precise query resolution.

Task 8 in SemEval 2025 – DataBench, Question-Answering over Tabular Data (Osés Grijalba et al., 2025), introduces a new benchmark for evaluating TQA systems. This benchmark enables the assessment of different question types spanning multiple information domains. The challenge lies in developing systems that can accurately answer queries over standardized datasets, where responses may take various forms, including boolean, categorical, numerical, or lists. Evaluation is based on the system's ability to provide precise answers given a (dataset, question) pair.

In this paper, we introduce QleverAnswering-PUCRS[1], a modular approach to Table Question Answering. Our system decomposes the TQA task into specialized agents, each responsible for a distinct function: metadata extraction, expression generation, error handling, and code execution. By structuring the workflow into interconnected components, QleverAnswering-PUCRS aims to improve observability, reduce execution failures, and enhance accuracy.

We evaluate QleverAnswering-PUCRS on benchmark datasets, demonstrating its effectiveness in handling complex queries over structured data. Our average results ranked us between 9^{th} and 19^{th} out of 49 participating systems across the four different rankings considered in the task. Nonetheless, we identify areas for further improvement and refinement in our approach.

2 Background and Related Work

TQA aims to answer questions referring to data stored in tables (Jin et al., 2022). Early methods had trouble generalizing across many data formats and depended on semantic parsing to translate natural language queries into structured languages like SQL. Modern approaches process tabular data using LLMs, avoiding the need for explicit query translation. Proprietary models tend to perform better, but all models have limits when performing sophisticated queries (Osés Grijalba et al., 2024).

[1]Our code is available at https://github.com/LucasAzeved/SemEval_2025_Task_8_QleverAnsweringPUCRS.

Proceedings of the The 19th International Workshop on Semantic Evaluation (SemEval-2025), pages 1342–1350

Current research has investigated retrieval-augmented generation (RAG), structured query generation, and simple neural network-based techniques to enhance query resolution. These approaches enable models to overcome context window restrictions by extracting pertinent metadata before query execution (Zhou et al., 2025). Program-based prompting is a further method in which models produce executable code to interpret tabular data rather than immediately responding to queries. This method is used for hybrid question answering by the HPROPRO framework (Shi et al., 2024), which shows good performance without the need for explicit data retrieval or transformation.

The MACT framework (Zhou et al., 2025) introduces a multi-agent approach, where distinct agents handle planning, query formulation, execution, and validation. This division of tasks enhances robustness, allowing for iterative refinement and error handling, leading to performance gains over fine-tuned LLMs in complex table-based reasoning tasks. Error correction mechanisms improve accuracy by identifying logical inconsistencies and refining queries dynamically (Shi et al., 2024).

Program-Aided Language Models (PAL) (Gao et al., 2023) propose generating intermediate Python code to be executed by an interpreter, which improves arithmetic and symbolic reasoning. This delegation enhances accuracy, especially for problems requiring precise calculation or logic that exceeds LLMs' intrinsic capabilities.

The Plan-of-SQLs (POS) framework (Nguyen et al., 2025) emphasizes the need for interpretability in Table QA. By decomposing complex queries into atomic SQL operations, POS enables transparent, step-by-step reasoning that can be verified by humans and LLMs alike. This work is particularly relevant in high-stakes domains, demonstrating that explainability can coexist with competitive performance.

While PAL and POS focus on symbolic execution and explainability, our approach targets the integration of structured program reasoning with lightweight retrieval to improve efficiency and adaptability in practical settings.

3 QleverAnswering-PUCRS

3.1 Task Description

The Question Answering over Tabular Data task generates accurate and contextually relevant responses to factoid questions based on structured data stored in tabular formats. Given a pair (d, q), where d is a dataset, and q is a question, we aim to produce an answer a corresponding to the response of q over d. The system needs to extract, interpret, and synthesize information directly from tables/datasets.

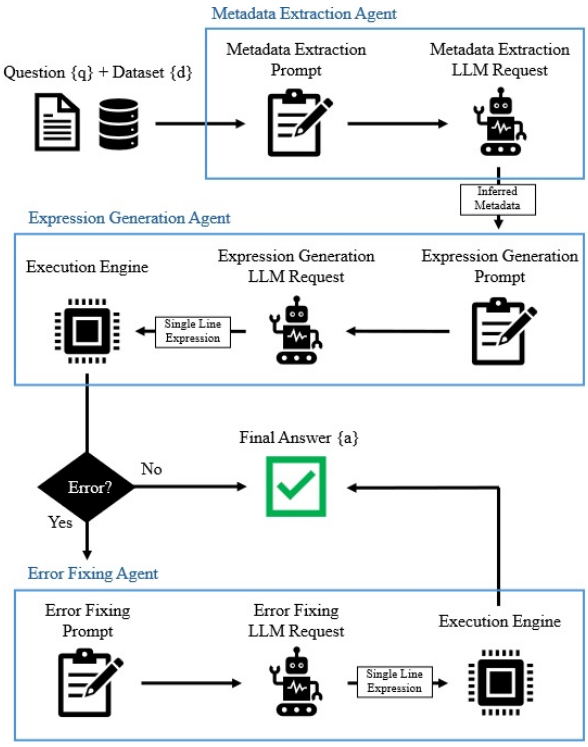

Figure 1: Overview of QleverAnswering-PUCRS

3.2 Solution Overview

QleverAnswering-PUCRS employs a modular agentic approach, where specialized agents handle specific subtasks such as metadata extraction, query interpretation, code generation, and validation. This pipeline aims to improve accuracy, facilitate debugging, and ensure adaptability to complex queries. The system enhances robustness by structuring the workflow into independent yet interconnected agents, allowing for targeted optimizations at each stage. An overview of QleverAnswering-PUCRS can be seen in Figure 1.

3.2.1 Metadata Extraction Agent

Extracting relevant information from large structured datasets is non-trivial, as that they may contain many columns. This fact, added to the limited context window of LLMs, makes it impractical to input an entire dataset in a prompt. Besides, we want to provide a concise prompt that only contains the information related to the question.

The first step of our approach is an *metadata extraction agent*, which pre-processes and structures metadata to guide later steps by filtering, cleaning, and organizing raw data into a prompt. We design the prompt presented in Appendix A.1.

The three components that are inserted into this prompt are (i) the dataframe preview, replacing the field `{df_str}` in the prompt, providing a snapshot of the dataset that captures a limited set of initial rows. This preview aims to allow the LLM to infer potential value distributions, column relationships, and general patterns in the data, ensuring that subsequent steps leverage meaningful insights; (ii) the columns information, on the field `{columns_info_str}`, containing structural details about the dataset, which includes the column names and data types. By incorporating this information, the LLM should be able to determine if the dataset contains categorical or numerical attributes, recognize potential restrictions, and validate whether a query can be executed without encountering errors due to type mismatches; the final and most critical input is (iii) the user query, in the field `{query}`, which represents the natural language question that the system must interpret and translate into a structured response. It is crucial that the query's intent can be readily determined and ambiguity avoided; otherwise, all future generated information can be misleading.

The output of this step consists of four components. The first is a list of relevant columns required to answer the query, and the second is the data type of each relevant column (*e.g.,* integer, string, boolean). The third component is the expected response type for that question, which is one of the following set {`boolean`, `number`, `category`, `list[category]`, `list[number]`}, and the last one is a sample answer, which is a plausible response generated based on the query and dataset preview.

3.2.2 Expression Generation Agent

Once relevant metadata has been identified on the input dataset, the next step is to convert the query into a valid expression that can extract the answer to the question over the provided dataframe. This step of the pipeline receives as input the newly refined information from the *metadata extraction agent*. Based on this data, the agent translates the natural language query into a single-line executable expression. This stage takes the refined data from the *metadata extraction agent* and converts the natural language query into a single-line expression, which disallows loops, variable assignments, or multi-line code. We designed the prompt in Appendix A.2 to achieve this goal.

The prompt template for this agent receives as input (i) the dataframe preview, in the field `{df_str}`, which consists of a small preview of the dataset, (ii) the structured set of metadata inferred from the *metadata extraction agent*, represented as `{generated_information}`, crucial for guiding the generation of the correct expression. This field is responsible for encapsulating key aspects such as the relevant columns, their data types, and the expected response format, ensuring that the generated expression aligns with the given query; (iii) the query itself, in the field `{query}`, which is provided to define the specific operation that must be translated into a valid executable expression.

Given that an incorrect expression may lead to execution failures, this step is critical to the pipeline's success, and its output is directly connected with the *execution engine module*, which acts as the pipeline's execution layer. This module receives a valid expression, executes it over the dataset in a code interpreter, and returns the result, which is the answer to a factoid question.

3.2.3 Error Fixing Agent

Despite careful query construction, execution errors may still arise due to column mismatches, such as referencing a non-existent column or data type conflicts, where numerical operations are applied to categorical data. Logical inconsistencies can also occur if incorrect assumptions are made during metadata extraction, leading to unexpected failures when executing expressions.

To address these issues, the *error fixing agent* automatically detects execution failures and reattempts expression generation with an improved context. Our approach detects errors as they occur. The failing expression and error message are captured in those cases to enhance the correction prompt. Using this updated context, a corrected expression is generated and re-executed by the *execution engine module*.

We designed the prompt in Appendix A.3 to achieve this goal. The inserted information also includes (i) a preview of the DataFrame, replacing the field `{df_str}` in the prompt, which provides a small snapshot of the dataset to offer context for the correction process; (ii) the previously generated expression, in the field

{previous_expression}, which serves as a reference for the attempted solution that encountered an error; (iii) the corresponding error message, represented as {error_encountered}, providing crucial insights into the nature of the failure, facilitating the identification of necessary corrections; and (iv) the query, in the field {query}, which defines the intended operation that must be correctly translated into an executable expression.

This iterative approach strengthens system reliability by reducing manual work and ensuring that erroneous queries can be resolved dynamically.

4 Experimental Setup

4.1 Dataset Description

We worked with two datasets: one during the development phase and another during the competition phase. For development, we used the 65 datasets from DataBench[2] (Osés Grijalba et al., 2024), a benchmark designed to provide a realistic and diverse testing ground for question-answering models over tabular data. DataBench consists of structured CSV-style files with varying numbers of rows and columns, representing real-world datasets across multiple domains.

During the competition phase, we used the 15 datasets provided by the task organizers for system evaluation. These datasets cover a diverse range of domains and vary in the number of rows and columns, allowing the models to be tested in an unseen environment with a wide range of structural complexities. Table 1 presents some statistics of these datasets, including the number of questions, columns, and rows of each test dataset. A Lite version of the datasets, containing the same questions and columns but limited to the first 20 rows of each dataset, was provided by the task organizers.

Dataset	#Ques	#Cols	#Rows
066_IBM_HR	39	35	1,470
067_TripAdvisor	29	10	20,000
068_WorldBank_Awards	34	20	239,461
069_Taxonomy	35	8	703
070_OpenFoodFacts	29	11	9,483
071_COL	36	8	121
072_Admissions	39	9	500
073_Med_Cost	32	7	1,338
074_Lift	35	5	3,000
075_Mortality	29	7	400
076_NBA	36	30	8,835
077_Gestational	31	7	1,012
078_Fires	39	15	517
079_Coffee	38	15	149,116
080_Books	41	13	40

Table 1: Statistics of Competition Datasets

4.2 Agents Description

Metadata Extraction LLM: We used Meta Llama 3 8B (Dubey et al., 2024) to extract key metadata from the dataset and the question. The model processes a structured prompt with a data set preview and a query. We first format the dataset to construct this prompt, ensuring that the column names are cleaned and represent the actual data. Given token limitations, only a subset of rows is included, focusing on covering different value types within the dataset.

Once the formatted prompt is sent to the model, it identifies the relevant columns required to answer the query and determines their data types. The model then classifies the expected response type, selecting from predefined categories to ensure consistency in subsequent processing steps. Additionally, it generates a sample answer based on the provided dataset fragment, serving as a reference. All LLM calls share the same request parameters: temperature = 0.0; max_tokens = 256. A low temperature tends to produce more deterministic responses. Due to API token constraints, where input tokens consume most of the available quota, we set max_tokens to avoid exceeding this limit.

Expression Generation LLM: For expression generation, we used Qwen2.5 Coder 32B Instruct (Hui et al., 2024). The model receives a prompt containing the dataset preview, user query, and the metadata extracted from the previous step. The prompt is designed to include all needed context while staying concise to fit token limits. The model is then instructed to generate a single-line Pandas[3] expression that directly answers the query without defining additional variables or performing unnecessary operations. Multi-line code, loops, and function definitions are explicitly disallowed to maintain compatibility with our execution framework. If the generated expression does not conform to the expected format or contains syntax errors, it is flagged for correction in the error-handling stage.

[2] https://huggingface.co/datasets/cardiffnlp/databench

[3] https://pandas.pydata.org/

System	Databench	Databench Lite
QleverAnswering-PUCRS	**76.82 (81.03)**	**79.50 (80.27)**
Ranking General	19/49	15/49
Ranking Open	13/35	9/35
Baseline	**26.00**	**27.00**
Ranking General	45/49	44/49
Ranking Open	31/35	31/35

Table 2: Comparison between our system's performance and the task baseline. Bold values indicate the accuracy (%) scores, with the official scores on DataBench after human review shown in parentheses.

Without Error Fixing				
	Acc	Q^+	Q^-	# Errors
DataBench	75,86	396	126	30
DataBench Lite	78,73	411	111	27
With Error Fixing				
	Acc	Q^+	Q^-	# Errors
DataBench	76,82	401	121	25
DataBench Lite	79,50	415	107	23

Table 3: Evaluation results for the DataBench and DataBench Lite on both setups, reporting % accuracy scores (Acc), correct predictions (Q^+), incorrect predictions (Q^-), and total number of errors.

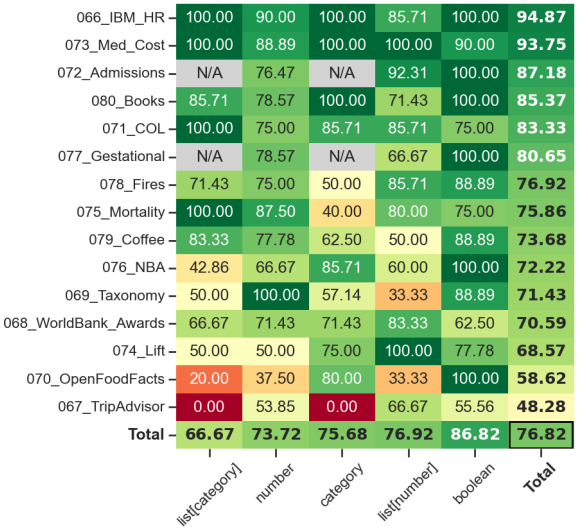

Figure 2: Accuracy heatmap on DataBench Dataset

Error Fixing LLM: We also utilized the Qwen2.5 Coder 32B Instruct (Hui et al., 2024) for error correction. The dataset preview, the previously created expression, the discovered error message, and the original query are all included in the structured prompt sent to the model. This prompt is thoughtfully structured to give all the required debugging context while being brief enough to adhere to token restrictions. The model is specifically told to examine the error, determine its possible causes, and provide a corrected one-line Pandas expression that fixes the problem. The system analyzes the model's response and runs the corrected expression generated against the dataframe. Regardless of the outcome, the execution response is saved as the final solution.

Execution Engine Module: To execute the generated expression, we utilized the `PandasInstructionParser` from LlamaIndex[4], which acts as an execution engine for Pandas-based operations over tabular data. The execution is orchestrated within a controlled pipeline using the `QueryPipeline` (QP) module, ensuring structured processing and response parsing. The pipeline consists of an input component that receives the expression and forwards it to the `PandasInstructionParser`, which interprets and executes the operation over the dataset in a Python 3.10 environment.

5 Results

Table 2 presents the official results of our system. In the open-source model ranking, our scores were 76.82 on Databench and 79.50 on Databench Lite, positioning us 13^{th} and 9^{th}, respectively, of 35 participants. In the general ranking, our scores remained the same, but we ranked 19^{th} and 15^{th} out of 49 participants. These results show strong mid-to-top performance, with our system achieving scores nearly three times higher than the task baseline.

Table 3 presents an ablation study evaluating the impact of the *error fixing agent* on the performance of the system. The results show an accuracy improvement of 0.96% for DataBench and 0.77% for DataBench Lite when the error fixing mechanism is applied. This is an improvement of 16.67% and 14.81% on the number of errors resolved in DataBench and DatabenchLite, respectively.

Figures 2 and 3 present the accuracy scores per semantic `category` (data type) and dataset. One notable observation is the consistency of our results across Databench and Databench Lite, suggesting that our model generalizes well even when restricted to only 20 rows per dataset.

Regarding different semantic types, our system

[4] https://docs.llamaindex.ai/en/stable/

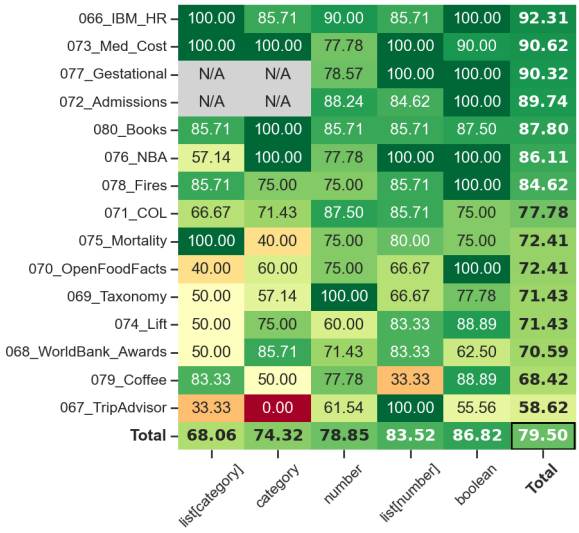

Figure 3: Accuracy heatmap on DataBench Lite Dataset

achieved its highest accuracy in the `boolean` type, which aligns with the expectation that the restricted domain in boolean values is easier to infer. In contrast, the lowest accuracy was observed in the `list[category]` type, which requires handling multiple categorical elements in a single response. The difficulty in correctly parsing and structuring multiple values is likely a contributing factor to this result, highlighting an area where additional refinement in the expression generation process might be beneficial.

One of the most significant drops in accuracy occurred in the `067_TripAdvisor` dataset, where our system obtained the lowest performance among all datasets. A deeper inspection reveals that this dataset contains JSON-formatted strings within queried columns, introducing additional complexity in extracting relevant values. Since our system relies on Pandas expressions to directly parse and filter data, handling embedded JSON structures within textual fields requires additional pre-processing, which was not included in our current pipeline. Similarly, the dataset `070_OpenFoodFacts` exhibited similar problems, but in this case, due to list-formatted strings rather than JSON, requiring specific parsing strategies that were not accounted for. This issue impacted both `list[category]` and `category` type questions, where correct identification and parsing of values were hindered, as well as `number` type questions, particularly those involving "How many..." questions, where quantities needed to be extracted from structured text fields.

Further analyzing the per-dataset performance, we identified patterns that explain specific drops in accuracy.

In the `075_Mortality` dataset, focused on `category` questions, all incorrect predictions occurred in queries requiring the identification of the entity with the highest or lowest average rate. The system consistently inverted the aggregation logic, incorrectly assuming that higher "Rate" values indicated better outcomes, whereas lower rates were preferable. This reveals a limitation in inferring the semantic orientation (*i.e.,* whether minimizing or maximizing is desirable) based solely on dataset structure.

In the `068_WorldBank_Awards` dataset, primarily involving `boolean` questions, errors arose from dataset ambiguities. Even manual inspection revealed that answering required domain knowledge beyond the available data preview, as column names and sample rows were insufficient to disambiguate the intended meaning without additional context.

6 Conclusion

In this work, we presented QleverAnswering-PUCRS, a modular system for Table Question Answering that relies on code generation over structured data. Our approach generates single-line Pandas expressions to extract answers, demonstrating strong performance across multiple semantic categories and significantly surpassing the task baseline in both Databench and Databench Lite evaluations.

Despite these promising results, our analysis revealed specific limitations. The system struggled to correctly infer the semantic orientation of metrics, particularly in tasks requiring judgment on whether lower or higher values are preferable, and showed difficulties when answering questions that demanded external domain knowledge beyond the provided data structure.

For future work, we plan to explore multi-step query execution strategies to improve reasoning over complex tabular data. We also intend to investigate the impact of model size on performance, assess the system's sensitivity to different prompt components, and evaluate iterative error correction methods to determine if multiple repair attempts can further improve accuracy. Additionally, incorporating external dataset descriptions or semantic enrichment mechanisms could mitigate context-related ambiguities.

References

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. *Preprint*, arXiv:2211.10435.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Nengzheng Jin, Joanna Siebert, Dongfang Li, and Qingcai Chen. 2022. A survey on table question answering: Recent advances. In *Knowledge Graph and Semantic Computing: Knowledge Graph Empowers the Digital Economy*, pages 174–186, Singapore. Springer Nature Singapore.

Giang Nguyen, Ivan Brugere, Shubham Sharma, Sanjay Kariyappa, Anh Totti Nguyen, and Freddy Lecue. 2025. Interpretable llm-based table question answering. *Preprint*, arXiv:2412.12386.

Jorge Osés Grijalba, L. 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 the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 13471–13488, Torino, Italia. ELRA and ICCL.

Jorge Osés Grijalba, Luis 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)*, Vienna, Austria. Association for Computational Linguistics.

Qi Shi, Han Cui, Haofeng Wang, Qingfu Zhu, Wanxiang Che, and Ting Liu. 2024. Exploring hybrid question answering via program-based prompting. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11035–11046, Bangkok, Thailand. Association for Computational Linguistics.

Wei Zhou, Mohsen Mesgar, Annemarie Friedrich, and Heike Adel. 2025. Efficient multi-agent collaboration with tool use for online planning in complex table question answering. In *Efficient Multi-Agent Collaboration with Tool Use for Online Planning in Complex Table Question Answering*.

A Prompt Templates

A.1 Metadata Extraction Prompt

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>You are an expert in Python and Pandas
    (version 2.2.2). Analyze a DataFrame and a query to infer key metadata required to process the
    query.

Task:
Based on the information provided:
1. Columns Used: Identify the relevant column(s) for answering the query.
2. Column Types: Provide the data types of the relevant columns.
3. Response Type: Choose one of: `boolean`, `number`, `category`, `list[category]`, or
    `list[number]`. No other formats are allowed.
4. Sample Answer: Generate a plausible sample answer based on the query and DataFrame preview,
    aligned with the Response Type.

Requirements:
- Pay close attention to what the query is asking for, it can be tricky.
- Only include columns explicitly needed for the query.
- Base the sample answer on plausible values from the provided DataFrame preview.
- Ensure the response is concise and well-structured.
- Do not provide any extra explanation or context beyond the requested metadata.

Output Format:
Columns Used: (columns_used)
Column Types: (column_types)
Response Type: (response_type)
Sample Answer: (sample_answer)
<|eot_id|>
<|start_header_id|>user<|end_header_id|>
DataFrame Preview:
`{df_str}`

Columns Information:
`{columns_info_str}`

Query:`{query}`
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|> Response:
```

A.2 Expression Generation Prompt

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>You are working with a pandas DataFrame
    in Python (version 2.2.2), named `df`.
Result of `print(df.head(2))`:
`{df_str}`

The following information was inferred from the DataFrame and query, you MUST use it to generate the
    expression:
`{generated_information}`

Instructions:
1. You are tasked to convert the query into **a SINGLE expression** using Pandas (version 2.2.2).
2. The result of the expression MUST be one of the following types: `boolean`, `number`,
    `category`, `list[category]`, or `list[number]`.
3. You MUST NOT return a DataFrame or any type not listed above.
4. **STRICTLY FORBIDDEN**: Writing multi-line code, defining variables, or using statements like
    `import`, `print`, or assignments (e.g., `x = ...`).
5. The Python expression MUST have only ONE line of code that can be executed directly using the
    `eval()` function.
6. **DO NOT USE NEWLINES** in the expression. Only return the single expression directly.
7. **DO NOT QUOTE THE EXPRESSION**. The output must ONLY be the raw code of the single expression.

*** Pay CLOSE attention to what the query is asking for, it can be tricky. ***
<|eot_id|>
<|start_header_id|>user<|end_header_id|>
Query: `{query}`
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>Expression:
```

A.3 Error Fixing Prompt

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>You are working with a Pandas DataFrame
    in Python (version 2.2.2) named `df`. Your task is to fix a failed Pandas expression by
    generating a new one that avoids the same error.
Below is a preview of the DataFrame (result of `print(df.head())`):
`{df_str}`

Task:
1. Analyze the provided DataFrame, query, expected expression reponse information, previous
    expression, and encountered error.
2. Generate a new expression that solves the query and prevents the error.

Previous Attempt:
Expression: `{previous_expression}`
Error: `{error_encountered}`

Expected Expression Reponse Information:
`{generated_information}`

Instructions:
1. The new expression MUST resolve the query and fix the error encountered previously.
2. The result of the expression MUST be one of the following types: `boolean`, `number`,
    `category`, `list[category]`, or `list[number]`. No other types are allowed.
3. You MUST NOT return a DataFrame, dictionary, or any type not explicitly listed above.
4. The Python expression MUST consist of ONLY ONE line of code and MUST be directly executable using
    the `eval()` function.
5. **STRICTLY FORBIDDEN**: Writing multi-line code, defining variables, or using statements like
    `import`, `print`, or assignments (e.g., `x = ...`).
6. **DO NOT INCLUDE NEWLINES** in the expression. Only provide a SINGLE LINE of code.
7. **PRINT ONLY THE RAW EXPRESSION**: Do not include explanations, comments, or quote the
    expression. The output must be directly evaluable.
8. Ensure the new expression fixes the error while strictly adhering to these rules. Failure to
    comply will result in failure of execution.

*** Pay CLOSE attention to what the query is asking for, it can be tricky. ***
<|eot_id|>
<|start_header_id|>user<|end_header_id|>
Query: `{query}`
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>Expression:
```