I2R-NLP at SemEval-2025 Task 8: Question Answering on Tabular Data

Yuze Gao, Bin Chen, Jian Su
A*STAR
{gaoy1,bchen,sujian}@i2r.a-star.edu.sg

Abstract

We present a Large Language Model (LLM) based system for question answering (QA) over tabular data that leverages multi-turn prompting to automatically generate executable Pandas functions. Our framework decomposes the problem into three key steps: (1)Answer Type Identification, where the system identifies the expected format of the response (e.g., boolean, number, category); (2) Pandas Function Generation, which generates a corresponding Pandas function using table metadata and in-context examples, and (3) Error Correction and Regeneration, where iteratively refining the function based on error feedback from executions. Evaluations on the SemEval-2025 Task 8 Tabular QA benchmark (Grijalba et al., 2024) demonstrate that our multi-turn approach significantly outperforms single-turn prompting models in exact match accuracy by 7.3%. The proposed system not only improves code generation robustness but also paves the way for enhanced and adaptability in table-QA reasoning tasks. Our implementation is available at `https://github.com/Gyyz/Question_Answering-over-Tabular-Data`.

1 Introduction

Answering natural language queries over tabular data requires a deep understanding of both linguistic nuances and structured data semantics. Traditional systems rely on rule-based approaches or parsing pipelines to translate questions into database queries (e.g., SQL). While effective in constrained domains, these approaches often demand significant manual engineering and domain expertise (Zelle and Mooney, 1996; Woods, 1977). These approaches can struggle with the complexities and ambiguities inherent in natural language. In contrast, recent advances in large language models (LLMs) have enabled prompt-based code generation, offering a promising alternative for complex reasoning tasks (Brown et al., 2020; Chen et al., 2021). LLMs have shown impressive capabilities in generating code from natural language descriptions, including for the task of Text-to-SQL (Yu et al., 2018; Sun et al., 2023), which focuses on translating natural language questions into SQL queries. However, single-turn prompts can fail to capture all necessary subtleties, leading to generated code that is either syntactically or semantically incorrect. This limitation highlights the need for more sophisticated prompting strategies.

Our work addresses these challenges by introducing a multi-turn prompting framework that engages the LLM in several refinement iterations. Unlike single-turn generation, our approach mirrors the iterative debugging process of human developers by correcting early mistakes and reinforcing the understanding of ambiguous queries. This stepwise process enables the system to produce executable Pandas functions that precisely match the intended output. This iterative refinement approach builds upon the concept of interactive code generation and human-in-the-loop AI for code, where human feedback and interaction are used to improve the quality and correctness of generated code. While interactive code generation has been explored, our specific application to generating Pandas functions for tabular data queries with multi-turn LLM interaction offers a novel contribution. The use of Pandas, a crucial library for data manipulation in Python, further motivates this work, as it enables the seamless integration of generated code into data science workflows.

2 Background

Over the past two years, the field of natural language processing (NLP) has undergone rapid evolution, largely driven by advances in large language models (LLMs). Early approaches were predominantly task-specific, demonstrating robust language understanding but frequently encountering limitations with respect to domain-specific tasks or complex, multi-step processes. The introduction of transformer-based architectures significantly improved scalability and generalization, paving the way for more sophisticated LLMs.

Recent innovations—exemplified by GPT-based models and open-source foundation models such as Llama (Touvron et al., 2023)—have further broadened the scope of potential applications by enhancing both data efficiency and the capacity to generate accurate, contextually appropriate responses to complex problems. However, it is important to note that our approach leverages human-prompted multi-step workflows rather than relying on purely model-driven multi-step reasoning. By incorporating iterative user input and guidance, we effectively harness the strong language comprehension of modern LLMs while maintaining precise oversight of the reasoning process.

(Brown et al., 2020) demonstrated the remarkable few-shot learning abilities of large models, revealing their potential to adapt to new tasks with minimal examples. In parallel, (Wei et al., 2022) introduced chain-of-thought prompting, a technique that decomposes complex reasoning tasks into intermediate steps, thereby improving the clarity and effectiveness of generated responses. Additionally, models such as (Zettlemoyer and Collins, 2005) and TAPAS(Herzig and Berant, 2020) have focused on bridging the gap between natural language queries and structured query languages, especially for tabular data.

Moreover, recent advances in Large Language Models (LLMs) such as Llama 3.3 (AI@Meta, 2024) to handle more complex and specialized tasks. Our approach builds upon these foundations by integrating an iterative error correction mechanism into the generation process, thereby ensuring that the output is both syntactically correct and semantically aligned with the intended query.

3 System Overview

Figure 1 provides an overview of our multi-turn prompting system in an end-to-end method (question to answer). The process begins with a user query and table metadata, and proceeds through three main stages, as described below.

3.1 Step 1: Answer Type Identification

The first step determines the expected answer type for the question. Since our dataset supports five distinct answer types (`boolean`, `category`, `list[category]`, `list[number]`, and `number`), this information is critical for generating a function that produces output in the correct format.

To achieve this, we craft a prompt with in-context examples that demonstrate the mapping from natural language queries to their answer types. For example, given a question like `"Which company has the highest revenue?"`, the expected answer type is `category`. Code Block 2.1 in **Appendix** Section shows a snippet of the prompt template used in this stage.

The LLM outputs the answer type appended with a special delimiter (e.g., a sequence of # characters), which is subsequently extracted using simple string operations. This preprocessing step is essential, as it ensures that the generated Pandas code adheres to the required output format. In our experiments on the development set and prompt template environments, omitting the **answer type** guidance resulted in a decrease in accuracy from **87%** to approximately **82%**.

3.2 Step 2: Pandas Function Generation

In the second step, the system generates an initial Pandas function that can answer the query. The prompt for this step is carefully constructed to include:

1). The **User Question**.

2). The predicted **Answer Type** from **Step 1**.

3). **Table Metadata** such as column names, column types, and sample rows from the corresponding database.

4). A set of **Example Shots** demonstrating similar question-to-function mappings.

The process can be divided into two sub-steps:

(I) **Retrieving Similar Shots:** To improve the accuracy of our generated code, we curated a dataset consisting of pairs of **User Questions** and their corresponding gold-standard **Pandas Functions**, which produce correct outputs upon execution. For each new **User Question**, we first retrieve a subset of training examples that share the same answer type. From this subset, we select k samples with semantically similar meanings. These examples are then incorporated into the prompt, providing the model with additional context to generate an appropriate Pandas function for a similar question. The aggregation process for these examples is illustrated in Code Snippet 2.2.1 in the Appendix.

(II) **Composing the Prompt:** With the simi-

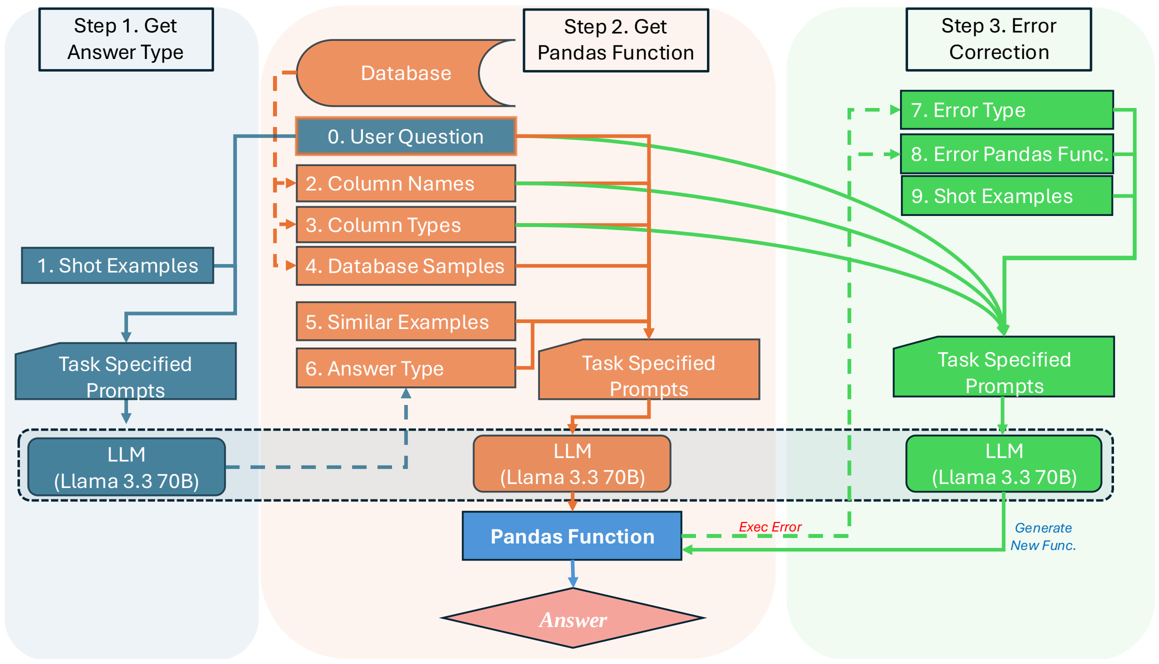

Figure 1: Overview of the multi-turn prompting system for tabular question answering. The figure illustrates the step-by-step process from initial query analysis to final answer extraction, highlighting the iterative error correction loop.

lar shots integrated, we construct a comprehensive prompt that incorporates additional context. Specifically, the prompt includes (1) general shot examples, (2) the user question, (3) column names and types, and (4) row samples from the relevant database. Code Snippet 2.2.2 in the Appendix provides an excerpt of the prompt composition.

Once the LLM generates the function (again ending with a special delimiter), we extract and execute the code. If the generated code produces the correct output, it is returned as the answer. This stage emphasizes the importance of precise prompt engineering in eliciting correct and executable code from the LLM.

3.3 Step 3: Error Correction and Regeneration

In real-world scenarios, generated code may occasionally fail during execution. Our system includes an error-handling loop that:

- Captures the **Error Message** from the failed execution.
- Combines this message with the original query, table metadata, and a concise description of the intended functionality.
- Retrieves additional example shots that illustrate proper error correction.

The error-correction prompt is designed to guide the LLM in revising the faulty function, below is an example.

Category	**Content**
Pandas Fn.	`df['Item'].dt.date.nunique()`
Error Msg.	`Can only use .dt accessor with datetimelike values`
Correction	`df['date_time'].dt.date.nunique()`

An example template is also shown in Code Snippet for **Step 3**, where all the **placeholder** fields are formatted with the information from **Step 2** and the relevant **Metadata**.

This iterative error correction mechanism not only enhances overall accuracy but also improves the system's resilience to minor syntactic and logical errors. The process emulates a human developer's workflow: debugging, refining, and retesting until a robust solution is achieved. In this study, we set the loop depth to 3, and the experimental results demonstrate a reduction in the execution error rate from **12%** to approximately **3%**.

4 Experimental Setup

4.1 Dataset and Evaluation Metrics

We evaluate our system on SemEval 2025 Task 8 Benchmark on a tabular QA. The dataset comprises tables from various domains along with corresponding natural language questions. We adhere to the official data splits and measure performance using: **Exact Match Accuracy:** The percentage of system-generated outputs that exactly match the gold-standard answers. Additionally, we report execution success rates to account for cases where minor formatting issues might otherwise obscure correct reasoning.

4.2 Implementation Details

Our experiments are conducted using the Llama 3.3 70B Instruct model, accessed via the `transformers` Python package. The model is deployed on a GPU server equipped with 6 NVIDIA A6000 GPUs. 4 of the GPUs are employed for the inferring process. To optimize for speed and memory efficiency, we load the Llama 70B model using quantization methods during the referring process. This quantization significantly reduces computational overhead while preserving the model's performance for generating and refining Pandas functions.

4.3 Baselines and Models

Baseline:
Single-Turn Prompting The LLM (quantized 70B) is prompted once to generate a Pandas function without subsequent error correction. Additionally, the Golden **Answer Type** information is provided instead of relying on the prediction from (**Step 1**), emphasizing the advantages of our multi-turn approach.

5 Results

Table 1 summarizes the performance of our multi-turn prompting system in comparison to the baseline on both Development and Testing Sets.

The multi-turn prompting framework exhibits a marked improvement over single-turn prompting, achieving higher accuracy and more robust handling of execution errors.

5.1 Discussion

The experimental results confirm that our multi-turn approach substantially improves the accuracy and reliability of automatically generated Pandas functions for tabular QA. Although the iterative refinement process introduces additional computational overhead, the increased robustness and error-correction capability justify the trade-off. Several key observations emerge from our study:

Error Sensitivity: Our approach incorporates an error-correction loop that leverages targeted feedback to systematically address both syntactic and semantic errors. This mechanism is highly effective, as demonstrated by a reduction in the execution error rate from **12%** to approximately **3%**. Such a significant improvement underscores the robustness of our method in refining the outputs generated by the language model.

In-Context Learning: By integrating similar examples directly into the input, we enhance the large language model's ability to generalize across a wide range of table schemas and query patterns. This in-context learning strategy contributes to a performance improvement of around **2%** in our experimental evaluations. The results indicate that providing contextual examples not only aids in comprehension but also improves the overall reliability of the model's predictions. We provide a table in the appendix for the detailed information.

Scalability: Although our current experiments have focused on relatively small tables, we recognize the importance of validating our approach on larger, real-world datasets. Future work will be directed towards extending the system's scalability while ensuring that its accuracy and efficiency are maintained in more complex environments. This exploration will be critical for adapting the system to practical applications where data size and variability are significantly higher.

Information Utilization: Our model architecture strategically leverages various types of information across different processing stages, with each element playing a distinct role in enhancing performance. For instance, the inclusion of **'Column_Types'** does not adversely affect performance during the initial processing stage (**Step 1**); however, it significantly contributes to performance in the later stage (**Step 3**). Moreover, the **'Answer_Type'** is particularly valuable in guiding the language model to generate the correct function corresponding to the user's query.

5.2 Performance Gap on Dev and Test Sets

Our system achieves an approximate exact match accuracy of **85%** on the development set but only

Model	Databench (Acc. %)	Databench(lite) (Acc. %)
Baseline (on Dev)	69.25	67.38
Steps (Ours) on Dev	**87.19**	**83.44**
Steps (Ours) on Test	**80.65**	**77.25**

Table 1: Performance comparison on the tabular QA task. The multi-turn framework achieves notable gains on both the development and test sets, demonstrating the effectiveness of iterative refinement.

around **77%** on the test set. Since our approach leverages a pre-trained LLM and does not involve traditional fine-tuning, overfitting is unlikely to be the primary cause of this discrepancy. A preliminary analysis suggests that the test set includes a higher frequency of queries requiring ‘List‘ type answers, which may expose limitations in our current postprocessing strategy. The potential issues include:

Format Sensitivity: The exact match metric depends on strict string-level comparisons. Even if the LLM generates semantically correct answers, slight variations in formatting such as whitespace, punctuation, or line breaks can lead to mismatches. Our postprocessing pipeline has not fully normalized these variations, causing correct answers to be marked as incorrect.

Output Format Mismatch: In some cases, the generated Pandas function is executable and returns the correct data, but the output format does not align with the expected answer format. For example, the correct answer might be returned within a list or nested structure, whereas our evaluation expects a simple scalar or a specific string representation. Such discrepancies directly impact the exact match accuracy.

Imprecision in Postprocessing: The current postprocessing procedures are not fully robust against the variability of LLM outputs. Minor inconsistencies in parsing or converting the returned output can lead to errors. This imprecision means that even correct executions may not be reflected accurately in the final evaluation metric.

5.3 Limitations

One limitation of our model is its reliance on rudimentary, unoptimized postprocessing. Like the baseline, it converts execution output into a string without advanced techniques. The Appendix provides details (see Code Snippet). Future work will enhance normalization and adopt flexible matching to better capture correct answers despite format variations. Additionally, using a general LLM may limit task-specific performance; replacing it with fine-tuned LLMs at different stages could yield better results.

6 Conclusion

We have presented a multi-turn prompting framework for the automated generation of Pandas functions in tabular question answering. By decomposing the problem into answer type extraction, initial function generation, and error-driven refinement, our system achieves substantial improvements over single-turn prompting and fine-tuned models. Our experiments demonstrate that multi-turn prompting enhances both accuracy and robustness, offering a promising direction for future research in table reasoning and prompt-based code generation.

7 Future Work

Looking ahead, we plan to explore several avenues for further improvement. In particular, we intend to:

1). Develop more sophisticated postprocessing techniques to handle format variations and improve exact match accuracy.

2). Extend our approach to support larger and more complex tables and domain-specific datasets.

3). Investigate the integration of additional feedback loops that can adaptively adjust prompt parameters based on real-time execution performance.

These directions aim to enhance both the scalability and the reliability of our system in real-world applications.

8 Acknowledgments

We thank our colleagues and the anonymous reviewers for their insightful comments. This work was partially supported by the programme DesCartes funded by the National Research Foundation, Prime Minister’s Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

References

AI@Meta. 2024. Llama 3 model card. Accessed: 2024-09-16.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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.

Ronan Herzig and Jonathan Berant. 2020. Tapas: Weakly supervised table parsing via pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2252–2261.

Shuo Sun, Yuchen Zhang, Jiahuan Yan, Yuze Gao, Donovan Ong, Bin Chen, and Jian Su. 2023. Battle of the large language models: Dolly vs llama vs vicuna vs guanaco vs bard vs chatgpt–a text-to-sql parsing comparison. *arXiv preprint arXiv:2310.10190*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models.

Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*.

William A Woods. 1977. Lunar rocks in natural english: Explorations in natural language question answering.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

John M Zelle and Raymond J Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pages 1050–1055.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence*, pages 658–666.

A Appendix

Code Snippet for Step 1

```
# Instruction: You are proficient in database and can easily tell the type of the
↪  retrieving answer for the question. Please complete the following function in
↪  one line. End your answer with ############
answer_types = ['boolean', 'category', 'list[category]', 'list[number]', 'number']
# TODO: complete the following function in one line. It should give the answer
↪  type for: List the 3 patents (by ID) with the most number of claims.
def get_answer_type() -> str:
    answer_types = ['boolean', 'category', 'list[category]', 'list[number]',
    ↪  'number']
    question = "List the 3 patents (by ID) with the most number of claims."
    return 'list[number]' ############

# TODO: complete the following function in one line. It should give the answer
↪  type for: Which graphext cluster is the most common among the patents?
def get_answer_type() -> str:
    answer_types = ['boolean', 'category', 'list[category]', 'list[number]',
    ↪  'number']
    question = "Which graphext cluster is the most common among the patents?"
    return 'category' ############

# TODO: complete the following function in one line. It should give the answer
↪  type for: List the 2 most common types of patents in the dataset.
def get_answer_type() -> str:
    answer_types = ['boolean', 'category', 'list[category]', 'list[number]',
    ↪  'number']
    question = "List the 2 most common types of patents in the dataset."
    return 'list[category]' ############

# TODO: complete the following function in one line. It should give the answer
↪  type for: Is the most favorited author mainly communicating in Spanish?.
def get_answer_type() -> str:
    answer_types = ['boolean', 'category', 'list[category]', 'list[number]',
    ↪  'number']
    question = "Is the most favorited author mainly communicating in Spanish?"
    return 'list[category]' ############

# TODO: complete the following function in one line. It should give the answer
↪  type for: {question (placeholder)}
def answer() -> str:
    answer_types = ['boolean', 'category', 'list[category]', 'list[number]',
    ↪  'number']
    question = {question (placeholder)}
    return
```

Code Snippet for Step 2.2.1

```
similar_shot_content = ""
for sid, shot in enumerate(shots):
    similar_shot_content += f"""
# example {5+sid}, similar case
# TODO: complete the following function in one line. The response type is one of
# ['boolean', 'category', 'list[category]', 'list[number]', 'number'].
def answer(df: pd.DataFrame) -> category:
    df.columns = {shot['columns']}
    df.column_types = {str(shot['column_types'])}
    return {shot['df_func']} ############
"""
```

Code Snippet for Step 2.2.2

```
prompt = """
# Instruction: You are proficient in pandas and its functions to retrieve data
↪  from a dataframe. Please complete the following function in one line. Be
↪  careful with the case, whitespaces and special characters in the column name.
↪  End your answer with ############

# example 1
# TODO: complete the following function in one line, response type in ['boolean',
↪  'category', 'list[category]', 'list[number]', 'number']. It should give the
↪  answer to: How many rows are there in this dataframe?
def answer(df: pd.DataFrame) -> number:
    df.columns=["A"]
    return df.shape[0] ############
# example 2
# TODO: complete the following function in one line, response type in ['boolean',
↪  'category', 'list[category]', 'list[number]', 'number']. It should give the
↪  answer to: What are the column names of this dataframe?
def answer(df: pd.DataFrame) -> list[category]:
    return df.columns.tolist() ############

# example 3, complex level
# TODO: complete the following function in one line, response type in ['boolean',
↪  'category', 'list[category]', 'list[number]', 'number']. It should give the
↪  answer to: List the top 5 ranks of billionaires who are not self-made.
def answer(df: pd.DataFrame) -> list[number]:
    df.columns = 'rank', 'personName', 'age', 'finalWorth', 'category', 'source',
    ↪  'country', 'state', 'city', 'organization', 'selfMade', 'gender',
    ↪  'birthDate', 'title', 'philanthropyScore', 'bio', 'about']
    return df.loc[df['selfMade'] == False].head(5)['rank'].tolist() ############

# example 4, complex level
# TODO: complete the following function in one line, response type in ['boolean',
↪  'category', 'list[category]', 'list[number]', 'number']. It should give the
↪  answer to: Which category does the richest billionaire belong to?
def answer(df: pd.DataFrame) -> category:
    df.columns = ['rank', 'personName', 'age', 'finalWorth', 'category', 'source',
    ↪  'country', 'state', 'city', 'organization', 'selfMade', 'gender',
    ↪  'birthDate', 'title', 'philanthropyScore', 'bio', 'about']
    return df.loc[df['finalWorth'].idxmax()]['category'] ############
"""
if ('similiar_shots' in global_config.features):
    prompt += similiar_shot_content

prompt += f"""
```

```
# TODO: complete the following function in one line, response type in ['boolean',
↪  'category', 'list[category]', 'list[number]', 'number']. It should give the
↪  answer to: {question}
def answer(df: pd.DataFrame) -> {row["type"]}:
    df.columns = {list(df.columns)} # column names
"""
if 'col_types' in global_config.features:
    prompt += f"""
    df.column_types = {str([itm.name for itm in df.dtypes])} # column types
"""

if 'row_samples' in global_config.features:
    prompt += f"""
    first{global_config.database_sample_number}_row_samples =
    ↪  {df.head(global_config.database_sample_number).to_dict(orient='records')}
    """
prompt += """
    return"""
"""
```

Code Snippet for Step 3

```
#Instruction: You are proficient in pandas and its functions to retrieve data from
↪  a dataframe. Please complete the following function in one line. End your
↪  answer with ############
# example 1
# Todo: Rewrite the pandas function based on the columns, the old function and the
↪  error message. It should give the right pandas function to: What is the
↪  average unit price?
def check_and_fix_function(question: str, columns: List[str], error_function: str,
↪  error_message: str) -> str:
    question = "What is the average unit price?"
    columns = ['InvoiceNo', 'Country', 'StockCode', 'Description', 'Quantity',
    ↪  'CustomerID', 'UnitPrice']
    error_function =  df[' UnitPrice'].mean()
    error_message = ' UnitPrice' # unexpected whitespace
    return  df['UnitPrice'].mean()  ############

# example 2
# Todo: Rewrite the pandas function based on the columns, the old function and the
↪  error message. It should give the right pandas function to:  What is the most
↪  commonly achieved educational level among the respondents?
def check_and_fix_function(question: str, columns: List[str], error_function: str,
↪  error_message: str) -> str:
    question = " What is the most commonly achieved educational level among the
    ↪  respondents?"
    columns = ['Are you registered to vote?', 'Which of the following best
    ↪  describes your ethnic heritage?', 'Who are you most likely to vote for on
    ↪  election day?', 'Division', 'Did you vote in the 2016 Presidential
    ↪  election? (Four years ago)', 'Weight', 'How likely are you to vote in the
    ↪  forthcoming US Presidential election? Early Voting Open', 'State', 'County
    ↪  FIPS', 'Who did you vote for in the 2016 Presidential election? (Four
    ↪  years ago)', 'What is the highest degree or level of school you have
    ↪  *completed* ?', 'NCHS Urban/rural', 'likelihood', 'Which of these best
    ↪  describes the kind of work you do?', 'How old are you?']
    error_function = df['What is the highest degree or level of school you have
    ↪  *completed*?'].value_counts().idxmax()
    error_message = "What is the highest degree or level of school you have
    ↪  *completed*?" # missed a whitespace
    return df['What is the highest degree or level of school you have *completed*
    ↪  ?'].value_counts().idxmax() ############

# example 3
# Todo: Rewrite the pandas function based on the columns, the old function and the
↪   error message. It should give the right pandas function to: Who are the top 2
↪   authors of the tweets with the most retweets?
def check_and_fix_function(question: str, columns: List[str], error_function: str,
↪   error_message: str) -> str:
    question = "Who are the top 2 authors of the tweets with the most retweets?"
    columns = ['id<gx:category>', 'author_id<gx:category>',
    ↪  'author_name<gx:category>', 'author_handler<gx:category>',
    ↪  'author_avatar<gx:url>', 'user_created_at<gx:date>',
    ↪  'user_description<gx:text>', 'user_favourites_count<gx:number>',
    ↪  'user_followers_count<gx:number>', 'user_following_count<gx:number>',
    ↪  'user_listed_count<gx:number>', 'user_tweets_count<gx:number>',
    ↪  'user_verified<gx:boolean>', 'user_location<gx:text>',
    ↪  'lang<gx:category>', 'type<gx:category>', 'text<gx:text>',
    ↪  'date<gx:date>', 'mention_ids<gx:list[category]>',
    ↪  'mention_names<gx:list[category]>', 'retweets<gx:number>',
    ↪  'favorites<gx:number>', 'replies<gx:number>', 'quotes<gx:number>',
    ↪  'links<gx:list[url]>', 'links_first<gx:url>', 'image_links<gx:list[url]>',
    ↪  'image_links_first<gx:url>', 'rp_user_id<gx:category>',
    ↪  'rp_user_name<gx:category>', 'location<gx:text>', 'tweet_link<gx:url>',
    ↪  'source<gx:text>', 'search<gx:category>']
    error_function =
    ↪  df.nlargest(2,'retweets')['author_name<gx:category>'].tolist()
    error_message = 'retweets'
    return df.nlargest(2,
    ↪  'retweets<gx:number>')['author_name<gx:category>'].tolist() ############

# example 4
# Todo: Rewrite the pandas function based on the columns, the old function and the
↪   error message. It should give the right pandas function to: Is there a patent
↪   abstract that mentions 'software'?
def check_and_fix_function(question: str, columns: List[str], error_function: str,
↪   error_message: str) -> str:
    question = "Is there a patent abstract that mentions 'software'?"
    columns = ['kind', 'num_claims', 'title', 'date', 'lang', 'id', 'abstract',
    ↪  'type', 'target', 'graphext_cluster', 'organization']
    error_function = ('software' in df['abstract'].values).any()
    error_message = "'bool' object has no attribute 'any'"
    return ('software' in df['abstract'].values) ############

# Todo: Rewrite the pandas function based on the columns, the old function and the
↪  error message. It should give the right pandas function to:
↪  {question(placeholder)}
def check_and_fix_function(question: str, columns: List[str], error_function: str,
↪  error_message: str) -> str:
    question =  {question(placeholder)}
    column_names =   {column_names(placeholder)}
    columns_types = {column_types(placeholder)}
    error_function =  {error_function(placeholder)}
    error_message =  {error_message(placeholder)}
    return
```

Code Snippet for Post-Processing

```
def post_process_ans_return(response):
    """
    Post-process the model's answer into a string representation.
    Handles lists, scalars, pandas Series/DataFrame, and categorical data.

    - Lists are converted to their string representation.
    - Scalars are converted to strings.
    - Pandas Series and DataFrames are converted by turning their elements into
    ↪  strings
      and then using `.to_string()` to produce a readable result.
    - Categorical data is handled by converting each element to a string.
    """

    # If response is None or already a string/scalar (int, float, bool, etc.),
    ↪  just return str
    if response is None or isinstance(response, (int, float, bool, str)):
        return str(response)

    # If response is a list, convert it to string
    if isinstance(response, list):
        return str(response)
    # If it's a Pandas Series
    if isinstance(response, pd.Series):
        # Convert categorical or object dtype elements to string individually
        # response = response.
        response = response.to_list()
        return str(response)
    # If it's a Pandas DataFrame
    if isinstance(response, pd.DataFrame):
        # Convert all elements to string before using to_string
        df_str = response.values.ravel().tolist()
        return str(df_str)
    # If it's some other type (e.g., numpy array or other objects), fallback to str
    return str(response)
```