Team Anotheroption at SemEval-2025 Task 8: Bridging the Gap Between Open-Source and Proprietary LLMs in Table QA

Nikolas Evkarpidi
HSE University
nik.evkarpidi@gmail.com

Elena Tutubalina
AIRI
Sber AI
Kazan Federal University
tutubalinaev@gmail.com

Abstract

This paper presents a system developed for SemEval 2025 Task 8: Question Answering (QA) over tabular data. Our approach integrates several key components: text-to-SQL and text-to-code generation modules, a self-correction mechanism, and a retrieval-augmented generation (RAG). Additionally, it includes an end-to-end (E2E) module, all orchestrated by a large language model (LLM). Through ablation studies, we analyzed the effects of different parts of our pipeline and identified the challenges that are still present in this field. During the evaluation phase of the competition, our solution achieved an accuracy of 80%, resulting in a top-13 ranking among the 38 participating teams. Our pipeline demonstrates a significant improvement in accuracy for open-source models and achieves a performance comparable to proprietary LLMs in QA tasks over tables. The code is available at this GitHub repository.

1 Introduction

Accessing structured data through natural language (NL) queries is crucial in various fields. However, converting NL into operations that retrieve outputs such as strings, numbers, booleans, or lists continues to be a significant challenge.

This paper outlines our team's participation in SemEval 2025 Task 8: DataBench (Osés Grijalba et al., 2024). This competition assesses question-answering (QA) systems working with tabular data, taking into account various formats, data quality issues, and complex question types. Our aim is to develop a system that accurately retrieves answers from tables, despite challenges such as missing values, inconsistencies, and ambiguous queries.

We focus on improving Large Language Models (LLMs) using Chain-of-Thought (CoT) reasoning (Wang et al., 2023; Cui et al., 2024). By integrating reasoning-inducing prompts, we enhance LLM-based code generation and decision-making. Additionally, our approach includes an end-to-end (E2E) pipeline and an LLM orchestrator to improve accuracy. To further refine performance, we implement several techniques aimed at reducing model forgetfulness. We introduce structured checklists to help the model verify each step, reducing errors in multi-hop reasoning. By combining prompt engineering, structured reasoning, and workflow optimizations, our system aims to achieve high exact-match accuracy. This paper details our system's architecture and key techniques (Sec. 3), dataset overview (Sec. 4), and experimental results (Sec. 5).

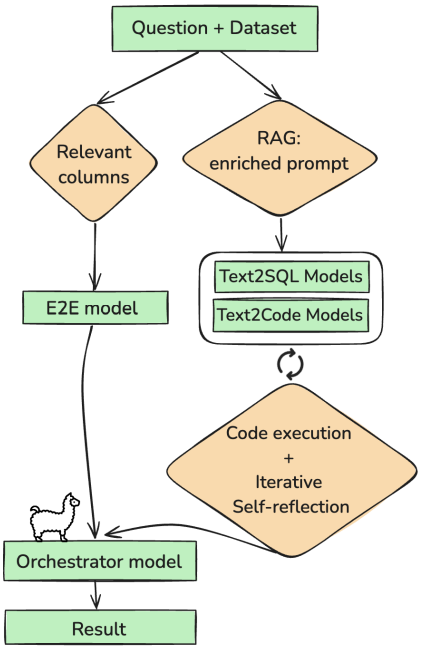

Figure 1: Overview of our system, featuring two solutions: end-to-end (E2E) and code-based. The code-based solution utilizes a self-correction mechanism and retrieval-augmented generation (RAG), with the final decision made by the orchestrator model.

Our findings offer key insights for enhancing LLM-driven QA for structured data, bridging the gap between open-source and proprietary models. Notably, our development set results showed that open-source LLMs achieved an accuracy of 88%, surpassing GPT-4o's 74%.

Proceedings of the The 19th International Workshop on Semantic Evaluation (SemEval-2025), pages 1870–1884
July 31 - August 1, 2025 ©2025 Association for Computational Linguistics

2 Related work

Question Answering (QA) over tabular data has gained significant attention due to the growing need for structured information retrieval (Sui et al., 2024; Liu et al., 2023; Singh and Bedathur, 2023; Ruan et al., 2024; R. et al., 2024). Research in this field has progressed with key datasets, such as FeTaQA (Nan et al., 2021) and ChartQA (Masry et al., 2022), as well as a large Wikipedia-based dataset, OpenWikiTable (Kweon et al., 2023). Various methodologies have been explored in recent surveys (Fang et al., 2024; Jin et al., 2022), including reinforcement learning and selective classification for text-to-SQL (Zhong et al., 2017; Somov et al., 2024; Somov and Tutubalina, 2025), pre-trained deep learning models (Abraham et al., 2022; Mouravieff et al., 2024), and few-shot prompting techniques (Guan et al., 2024). Building on previous work, we introduce a hybrid LLM-based pipeline that combines multiple techniques to improve performance.

3 System Description

Our system, as shown in Fig. 3, leverages LLMs and consists of the following key elements:

1. Text-to-SQL and Text-to-Code models to translate NL questions into code executable against tabular data (Sec. 3.2; Sec. 3.3)

2. RAG used to enrich prompts with relevant rows and delete irrelevant columns (Sec. 3.4);

3. Self-correction mechanism used to correct potential errors during execution (Sec. 3.5);

4. An E2E answering model to answer questions that target semantic understanding (Sec. 3.7);

5. An orchestrator model to make the final decision between provided solutions (Sec. 3.8).

3.1 Models

We used state-of-the-art instruction-tuned models featuring various model families: Llama (Grattafiori et al., 2024) (version 3.3 with 70b parameters as an orchestrator and 3.2 version with 3b for retrieval), Codestral (20.51 version) (Mistral AI Team, 2025) and Qwen Coder Instruct (2.5 version with 32b parameters) (Hui et al., 2024) for SQL and Code generation, also MiniMax-01 (MiniMax et al., 2025) for E2E solution. The selection of the models was driven by their outstanding performance in various benchmarks and the fact that they are open-source.

3.2 SQL code generation

Here the system resorts to generating SQL queries while using a carefully crafted prompt with a few relevant rows injected in it (Sec. 3.4) and with a suggested list of relevant columns to use (Sec. 5.5). The query is executed against the in-memory database (SQLite database via the SQLAlchemy package in our case), and the result is formatted and returned as a potential solution.

3.3 Pandas code generation

Here we prompt LLM to generate Python code with the use of Pandas library. The code is then executed against a Pandas Dataframe within a sandboxed environment with a timeout to prevent indefinite loops. The result of the execution is recorded as a potential solution. Along with the result, we record query success status and error text (if present) for possible future error correction (Sec. 3.5).

3.4 Retrieval

The Databench dataset contains data similar to what you might find in the real world, which presents certain challenges. One of these challenges is accurately filtering data based on specific properties, which often requires contextual knowledge. For example, to answer the question "How many customers are from Japan?", the model needs to know that "japan" is spelled in lowercase in the dataset. To tackle this challenge, we implemented a retrieval step (Gao et al., 2024). We first created sentence embeddings for each relevant column (previously identified in Sec. 5.5) and stored them for efficient searching. When a question was asked, we searched these embeddings to find the top three rows that were most semantically similar. The retrieved data was then used to enrich the LLM's context, enabling the model to answer such questions more accurately and efficiently.

3.5 Self-correction

The system incorporates a self-correction mechanism (Deng et al., 2025) that attempts to refine solutions that have failed execution attempts. After the failure of the Pandas solution, the system passes meta-info (schema, error message), along with the question back to LLM. Then new solu-

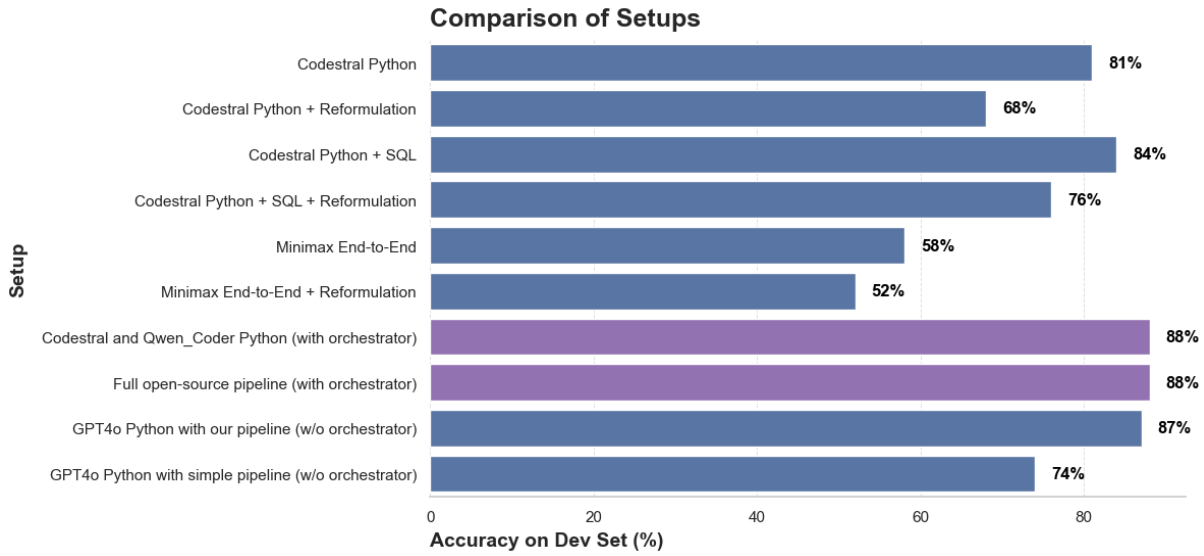

Figure 2: System performance on the dev set. Llama3.3-70b-instruct is used as an orchestrator. As a full pipeline, we're using: Codestral and Qwen Coder for Python and SQL, with Minimax E2E, managed through an orchestrator.

tion is generated. The same rules apply to SQL solutions.

3.6 Reasoning step

Recent advancements in prompting techniques, such as chain-of-thought prompting (Wei et al., 2022), have demonstrated that large language models (LLMs) can be guided to perform complex reasoning by structuring prompts to include intermediate reasoning steps. In our approach, we leverage this technique by explicitly instructing the LLM to reason extensively before providing its final answer. To extract only the relevant answer from the model's response, we employ fuzzy matching, which allows us to identify and isolate the desired output even when the response contains additional explanatory text or reasoning steps.

3.7 E2E answer generation

This method completely skips the code generation. The dataset is converted into human-readable text (markdown), then given to the LLM along with the underlying question. The model generates a direct answer. Model must put solution into one of the following data formats: Boolean, List, Number or String. The method is used to take advantage of LLM's ability to understand text and, therefore, answer questions about text data from the dataset.

Unlike code-generation, an E2E solution may only work well in a limited context: that is why we use the Retrieval step (see Sec. 3.4) and combine E2E with code-generation approaches to further increase performance.

3.8 Orchestrator

We use Llama (3.3 instruct version with 70b parameters) to choose the most probable solution among all presented. The model is provided with several solutions that were successfully executed. Each solution has code and a text-formatted result. Prompt (See Appendix prompt) has specific recommendations on how to choose the most probable solution. The model is incapable of generating new solution on the fly and only chooses between the presented options. Further orchestrator's performance analysis is in Sec. 5.3.

4 Dataset

The dataset comprises 65 publicly available tables across five domains: Health, Business, Social Networks & Surveys, Sports & Entertainment, and Travel & Locations. It retains real-world noise to enhance robustness and includes **1300** manually curated QA pairs **in English**, with 500 used for the test set across five answer types: boolean, category, number, list[category], and list[number]. DataBench is provided in two versions: the full dataset and DataBench lite, a smaller subset containing the first 20 rows per dataset. Key dataset statistics are summarized in Tab. 4. To illustrate dataset diversity, Fig. 5 presents five representative question types. Our **dev set** consists of the first 100 QA pairs, designated for hypothesis testing.

A notable challenge was handling emojis in col-

umn names and textual data, as exact answer matching was required per competition rules, but LLMs struggle with emojis (Qiu et al., 2024). They often insert spaces or omit them, leading to inaccuracies. We mitigated this issue by:

a) Replacing emojis in column names with unique symbols (hashes) for easier query generation.

b) Restricting the orchestrator to selecting answers from SQL or Python outputs rather than generating responses, ensuring accuracy.

4.1 Accuracy Calculation

The evaluation was conducted using the framework provided in the repository. The evaluation metric was calculated by the rules presented in Fig. 4.1. The approach is flexible and provides a fair metric calculation for different pipelines.

Comparison Rules

Numbers: Truncated to two decimals.

Categories: Compared directly as-is.

Lists: Order is ignored.

5 Experiments and Evaluation

This section details the experiments conducted to evaluate our system's performance in Table QA. More experiments are in Appx. A.1 and A.2.

As shown in Fig. 2, reformulating the question generally decreases accuracy (we discuss why in Sec. 5.4), as seen in setups like "Codetral Python + Reformulation" (68%) and "Minimax End-to-End + Reformulation" (52%), both of which perform worse than their non-reformulated counterparts. Whereas, adding SQL capabilities tends to increase accuracy, with "Codetral Python + SQL" reaching 84%. The highest accuracy is achieved when multiple models are combined and orchestrated, such as "Codetral and Qwen Coder Python (with orchestrator)" and "Full open-source pipeline (with orchestrator)", both achieving 88%, surpassing single-model approaches.

5.1 Performance of Code-Generation

Text-to-SQL and text-to-code generation performed well on structured queries but struggled with ambiguous questions that lack explicit context. The self-correction mechanism improved accuracy by refining failed queries. However, unclear queries, such as "Provide the median number of claims for B2 and S1 kinds" could lead to misinterpretations, whether computing a single median or separate medians, resulting in incorrect outputs.

5.2 Effectiveness of E2E Processing

The E2E approach, which skips code generation and directly answers questions using a textual representation of the table, performed well on questions requiring semantic understanding. For instance, it excelled at answering non-exact questions like "Is there a patent related to 'communication' in the title?". Furthermore, models were given the task of answering the question: "How many distinct male participants took part in the competition?" based solely on participants' names. This required the models to infer the participants' sex from their names that is a task that LLMs typically excel at. However, solving this problem using SQL or Python alone would be quite challenging. In such cases, both systems complemented each other, leveraging the strengths of LLMs for context understanding and the structured data processing power of SQL and Python.

5.3 Orchestrator Performance

The orchestrator model, which selects the most probable solution from multiple candidates, generally performed well. However, its accuracy depended heavily on the quality of the candidate solutions. If all candidates were incorrect, the model couldn't generate a correct answer on its own. Additionally, when the majority of candidate answers were incorrect, the orchestrator sometimes failed to select the correct solution, instead favoring the most frequent or popular response.

We propose, for future research, exploring automatic methods to determine whether a given question is better suited for SQL or Python-based querying. This could help the orchestrator make more informed decisions, leading to improved accuracy and efficiency in selecting the correct answer.

5.4 Question Reformulation

Handling ambiguous or under-specified queries is a key challenge in structured data QA with LLMs (Zhao et al., 2024). We tested LLM-based question reformulation to make queries more explicit, but it proved counterproductive (as seen in Fig. 2). Errors in reformulation at the pipeline's start led to failures without any recovery mechanism. Conversely, without reformulation, some models inferred intent correctly, enabling the orchestrator to choose the

Original Name	Renamed Column
DMC	Duff Moisture Code
DC	Drought Code
ISI	Fire Spread Index
RH	Relative Humidity

Table 1: Renaming ambiguous column names for clarity using context and LLM insights.

right response even if others failed. While reformulation may be less effective with multiple models (e.g., in our case: two SQL, two Python, one E2E), further research is needed to confirm this.

5.5 Predicting Useful Columns

E2E models often encounter challenges when working with long-context data, a difficulty sometimes referred to as "The Needle In a Haystack problem" (Laban et al., 2024). To address this problem, we introduced LLM-driven column selection. The separate model is given a description of the query and asked to select the most relevant columns before attempting to generate an answer. This method ensures that only the useful parts of the dataset are provided to the E2E model, reducing context length and minimizing the risk of hallucinations.

5.6 Column name explanation

Structured datasets often have ambiguous or abbreviated column names, making LLM comprehension challenging. To address this, we introduced column reformulation. For example, given the table "078 Fires" and initial data rows, LLMs effectively generated clearer column names.

Tab. 1 highlights the ambiguity of original column names, which were clarified through renaming for better usability. However, this poses challenges, as users may refer to original names, necessitating entity recognition for query adjustments, which is beyond the scope of our study. Misinterpretation is also a risk: abbreviations like DC and DMC have multiple meanings, and even strong models can generate incorrect names (e.g., GPT-4o renamed ISI as Fire Spread Index instead of Initial Spread Index). Further research is needed to refine this strategy for effective QA pipeline integration.

6 Comparison with proprietary models

The integration of multiple components showcased the potential of open-source LLMs for solving QA tasks over tabular data. As Fig. 4 illustrates, we compared Codestral against GPT-4o (OpenAI et al., 2024), both utilizing our pipeline. While GPT-4o outperformed Codestral, the performance gap remained within a reasonable range. However, the best results were achieved by a two-model system, which combined Codestral and Qwen Coder for Python code generation, managed by an orchestrator. This setup reached 88% accuracy, surpassing GPT-4o. By leveraging an orchestrator to optimize the strengths of multiple models, our approach demonstrates that open-source solutions can achieve accuracy levels comparable to proprietary models. Our pipeline applied to GPT4o (w/o orchestrator) also performs well (87%), resulting in a noticeable improvement over a simpler pipeline, showing the effectiveness of such an approach even for already strong proprietary models.

Rank	Codabench ID	Team	Score
1	xiongsishi	TeleAI	95.02
2	pbujno	SRPOL AIS	89.66
13	**anotheroption**	**anotheroption**	**80.08**
	baseline	stable-code-3b-GGUF	26.00

Table 2: Official results among open source models

7 Official results

We ranked in the top 13 out of 38 teams in the competition's OpenSource-models-only section (Osés-Grijalba et al., 2025), achieving an accuracy score of 80% on the Databench evaluation as well as on a lite part of the benchmark. Our official results on Databench part of the task are presented in Tab. 2, showing that we significantly outperformed the baseline by 54 points. The best solution achieved a score of 95.20. In the global ranking presented in Tab. 3, which includes proprietary models, we placed in the top 20 out of 53 teams while exclusively using open-source models.

8 Error Analysis

8.1 Orchestrator decisions

In Fig. 3, the distribution of orchestrator decision types is shown. Most cases (63.4%) involved simple confirmation of consensus among identical outputs ('Agreement'). However, in 36.6% of scenarios, the orchestrator took a more active role: filtering out logically flawed responses (14.6%), rejecting answers with mismatched data formats (12.2%), or resolving conflicts between divergent yet seemingly valid outputs (9.8%).

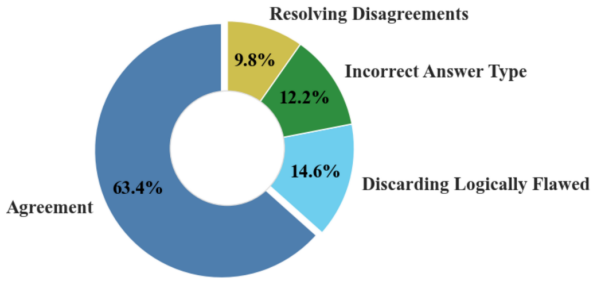

Figure 3: Distribution of LLM orchestrator decision scenarios (based on 41 questions from the dev set).

8.2 Code-based solution failure analysis

Some code-based solutions had incorrect syntax. There are several common patterns in which this occurred.

- Incorrect aggregation: queries with broken logical chains, incorrect applications of aggregation functions or "group by" operation.
- Type unaware operations: the system would often make syntax errors due to incorrect handling, such as trying to retrieve properties over int objects.
- Flawed code understanding: errors included attempts to call Pandas methods with incorrect or omitted arguments.

And when the code syntactically was correct, there were several common failure patterns.

- Subtle logical errors: this often manifests syntactically correct code that nonetheless employs incorrect aggregation, filtering, or sorting logic for the specific dataset. *Example: Incorrect identification of the most retweeted author due to flawed aggregation.*
- Query misinterpretation: in these cases, the generated code fails to capture the full intent of the query. *Example: Returning pokemon name instead of total stats when asked for the "lowest total stats of pokemon".*
- Data-specific edge cases: generated code struggles with particular data characteristics, such as incorrectly handling null values, emojis, timestamps, or failing to provide a robust approach to tied rankings in sorting or max/min operations. *Example: Failure to correctly identify authors of shortest posts due to inaccurate word count.*

Identifying these distinct failure types is crucial for improving the overall reliability of the Q&A system.

8.3 Self-correction

The self-correction mechanism was largely ineffective due to the system design involving multiple LLM agents: two for Python and two for SQL. In the vast majority of cases, at least one Python and one SQL agent a runnable solution. As a result, the orchestrator could select a valid answer without having to rely on self-correction.

9 Conclusion

We introduced a comprehensive system for QA over tables, showcasing that well-orchestrated open-source models can rival proprietary solutions. We tested various methods: some risked errors, while others improved accuracy and reliability of the system. Our system ranked among the top 13 teams with 80% accuracy. Future work could explore dynamic pipeline selection — automatically determining whether a question requires code-based execution, semantic analysis, or hybrid approaches — to optimize efficiency and accuracy. Additionally, enhancing the orchestrator's capacity to detect and correct logical inconsistencies in candidate answers could further improve robustness.

10 Limitations

The performance of the system exhibits significant variability across different model sizes. Additionally, retrieval systems often encounter challenges when the terms in a query do not align well with the tabular data being searched, and embedding models do not completely address this issue. A notable limitation lies in the generation of candidates for orchestration, where it is possible for all generated responses to be incorrect. This represents a well-known challenge, as identified in prior work (Bradley, 2024), highlighting how certain tasks can prove difficult even for comprehensive groups of large language models (LLMs). In these instances, the system is inherently designed without a mechanism to independently generate a correct answer. Future research could explore potential strategies to address such scenarios effectively.

11 Ethics Statement

All models and datasets used are publicly available. We honor and support the ACL Code of Ethics.

Acknowledgments

We acknowledge the computational resources of HPC facilities at the HSE University. The work has been supported by the Russian Science Foundation grant #23-11-00358.

References

Abhijith Neil Abraham, Fariz Rahman, and Damanpreet Kaur. 2022. Tablequery: Querying tabular data with natural language. *Preprint*, arXiv:2202.00454.

William F. Bradley. 2024. Llms and the madness of crowds. *Preprint*, arXiv:2411.01539.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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.

Jonathan Cook, Tim Rocktäschel, Jakob Foerster, Dennis Aumiller, and Alex Wang. 2024. Ticking all the boxes: Generated checklists improve llm evaluation and generation. *Preprint*, arXiv:2410.03608.

Yingqian Cui, Pengfei He, Xianfeng Tang, Qi He, Chen Luo, Jiliang Tang, and Yue Xing. 2024. A theoretical understanding of chain-of-thought: Coherent reasoning and error-aware demonstration. *Preprint*, arXiv:2410.16540.

Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. 2025. Reforce: A text-to-sql agent with self-refinement, format restriction, and column exploration. *Preprint*, arXiv:2502.00675.

Xi Fang, Weijie Xu, Fiona Anting Tan, Ziqing Hu, Jiani Zhang, Yanjun Qi, Srinivasan H. Sengamedu, and Christos Faloutsos. 2024. Large language models (LLMs) on tabular data: Prediction, generation, and understanding - a survey. *Transactions on Machine Learning Research*.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-augmented generation for large language models: A survey. *Preprint*, arXiv:2312.10997.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, et al. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Che Guan, Mengyu Huang, and Peng Zhang. 2024. Mfort-qa: Multi-hop few-shot open rich table question answering. In *International Conference on Computing and Artificial Intelligence*.

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. *arXiv preprint arXiv:2409.12186*.

Nengzheng Jin, Joanna Siebert, Dongfang Li, and Qingcai Chen. 2022. A survey on table question answering: Recent advances. *ArXiv*, abs/2207.05270.

Sunjun Kweon, Yeonsu Kwon, Seonhee Cho, Yohan Jo, and Edward Choi. 2023. Open-wikitable: Dataset for open domain question answering with complex reasoning over table. *Preprint*, arXiv:2305.07288.

Philippe Laban, Alexander R. Fabbri, Caiming Xiong, and Chien-Sheng Wu. 2024. Summary of a haystack: A challenge to long-context llms and rag systems. *Preprint*, arXiv:2407.01370.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2022. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7716–7730.

Tianyang Liu, Fei Wang, and Muhao Chen. 2023. Rethinking tabular data understanding with large language models. *Preprint*, arXiv:2312.16702.

Ahmed Masry, Do Xuan Long, Jia Qing Tan, Shafiq R. Joty, and Enamul Hoque. 2022. Chartqa: A benchmark for question answering about charts with visual and logical reasoning. *ArXiv*, abs/2203.10244.

MiniMax, Aonian Li, Bangwei Gong, Bo Yang, Boji Shan, Chang Liu, Cheng Zhu, Chunhao Zhang, Congchao Guo, Da Chen, Dong Li, Enwei Jiao, Gengxin Li, Guojun Zhang, Haohai Sun, Houze Dong, Jiadai Zhu, Jiaqi Zhuang, Jiayuan Song, Jin Zhu, Jingtao Han, Jingyang Li, Junbin Xie, Junhao Xu, Junjie Yan, Kaishun Zhang, Kecheng Xiao, Kexi Kang, Le Han, Leyang Wang, Lianfei Yu, Liheng Feng, Lin Zheng, Linbo Chai, Long Xing, Meizhi Ju, Mingyuan Chi, Mozhi Zhang, Peikai Huang, Pengcheng Niu, Pengfei Li, Pengyu Zhao, Qi Yang, Qidi Xu, Qiexiang Wang, Qin Wang, Qiuhui Li, Ruitao Leng, Shengmin Shi, Shuqi Yu, Sichen Li, Songquan Zhu, Tao Huang, Tianrun Liang, Weigao Sun, Weixuan Sun, Weiyu Cheng, Wenkai Li, Xiangjun Song, Xiao Su, Xiaodong Han, Xinjie Zhang, Xinzhu Hou, Xu Min, Xun Zou, Xuyang Shen, Yan Gong, Yingjie Zhu, Yipeng Zhou, Yiran Zhong, Yongyi Hu, Yuanxiang Fan, Yue Yu, Yufeng Yang, Yuhao Li, Yunan Huang, Yunji Li, Yunpeng Huang, Yunzhi Xu, Yuxin Mao, Zehan Li, Zekang Li, Zewei Tao, Zewen Ying, Zhaoyang Cong, Zhen Qin, Zhenhua Fan, Zhihang Yu, Zhuo Jiang, and Zijia Wu. 2025. Minimax-01: Scaling foundation models with lightning attention. *Preprint*, arXiv:2501.08313.

Mistral AI Team. 2025. Codestral 25.01. [Online; accessed 20-February-2025].

Raphael Mouravieff, Benjamin Piwowarski, and Sylvain Lamprier. 2024. Training table question answering via sql query decomposition. *ArXiv*, abs/2402.13288.

Linyong Nan, Chia-Hsuan Hsieh, Ziming Mao, Xi Victoria Lin, Neha Verma, Rui Zhang, Wojciech Kryscinski, Nick Schoelkopf, Riley Kong, Xiangru Tang, Murori Mutuma, Benjamin Rosand, Isabel Trindade, Renusree Bandaru, Jacob Cunningham, Caiming Xiong, and Dragomir R. Radev. 2021. Fetaqa: Free-form table question answering. *Transactions of the Association for Computational Linguistics*, 10:35–49.

OpenAI, :, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Mądry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, Alex Nichol, Alex Paino, Alex Renzin, Alex Tachard Passos, Alexander Kirillov, Alexi Christakis, Alexis Conneau, Ali Kamali, Allan Jabri, Allison Moyer, Allison Tam, Amadou Crookes, Amin Tootoochian, Amin Tootoonchian, Ananya Kumar, Andrea Vallone, Andrej Karpathy, Andrew Braunstein, Andrew Cann, Andrew Codispoti, Andrew Galu, Andrew Kondrich, Andrew Tulloch, Andrey Mishchenko, Angela Baek, Angela Jiang, Antoine Pelisse, Antonia Woodford, Anuj Gosalia, Arka Dhar, Ashley Pantuliano, Avi Nayak, Avital Oliver, Barret Zoph, Behrooz Ghorbani, Ben Leimberger, Ben Rossen, Ben Sokolowsky, Ben Wang, Benjamin Zweig, Beth Hoover, Blake Samic, Bob McGrew, Bobby Spero, Bogo Giertler, Bowen Cheng, Brad Lightcap, Brandon Walkin, Brendan Quinn, Brian Guarraci, Brian Hsu, Bright Kellogg, Brydon Eastman, Camillo Lugaresi, Carroll Wainwright, Cary Bassin, Cary Hudson, Casey Chu, Chad Nelson, Chak Li, Chan Jun Shern, Channing Conger, Charlotte Barette, Chelsea Voss, Chen Ding, Cheng Lu, Chong Zhang, Chris Beaumont, Chris Hallacy, Chris Koch, Christian Gibson, Christina Kim, Christine Choi, Christine McLeavey, Christopher Hesse, Claudia Fischer, Clemens Winter, Coley Czarnecki, Colin Jarvis, Colin Wei, Constantin Koumouzelis, Dane Sherburn, Daniel Kappler, Daniel Levin, Daniel Levy, David Carr, David Farhi, David Mely, David Robinson, David Sasaki, Denny Jin, Dev Valladares, Dimitris Tsipras, Doug Li, Duc Phong Nguyen, Duncan Findlay, Edede Oiwoh, Edmund Wong, Ehsan Asdar, Elizabeth Proehl, Elizabeth Yang, Eric Antonow, Eric Kramer, Eric Peterson, Eric Sigler, Eric Wallace, Eugene Brevdo, Evan Mays, Farzad Khorasani, Felipe Petroski Such, Filippo Raso, Francis Zhang, Fred von Lohmann, Freddie Sulit, Gabriel Goh, Gene Oden, Geoff Salmon, Giulio Starace, Greg Brockman, Hadi Salman, Haiming Bao, Haitang Hu, Hannah Wong, Haoyu Wang, Heather Schmidt, Heather Whitney, Heewoo Jun, Hendrik Kirchner, Henrique Ponde de Oliveira Pinto, Hongyu Ren, Huiwen Chang, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian O'Connell, Ian Osband, Ian Silber, Ian Sohl, Ibrahim Okuyucu, Ikai Lan, Ilya Kostrikov, Ilya Sutskever, Ingmar Kanitscheider, Ishaan Gulrajani, Jacob Coxon, Jacob Menick, Jakub Pachocki, James Aung, James Betker, James Crooks, James Lennon, Jamie Kiros, Jan Leike, Jane Park, Jason Kwon, Jason Phang, Jason Teplitz, Jason Wei, Jason Wolfe, Jay Chen, Jeff Harris, Jenia Varavva, Jessica Gan Lee, Jessica Shieh, Ji Lin, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joanne Jang, Joaquin Quinonero Candela, Joe Beutler, Joe Landers, Joel Parish, Johannes Heidecke, John Schulman, Jonathan Lachman, Jonathan McKay, Jonathan Uesato, Jonathan Ward, Jong Wook Kim, Joost Huizinga, Jordan Sitkin, Jos Kraaijeveld, Josh Gross, Josh Kaplan, Josh Snyder, Joshua Achiam, Joy Jiao, Joyce Lee, Juntang Zhuang, Justyn Harriman, Kai Fricke, Kai Hayashi, Karan Singhal, Katy Shi, Kavin Karthik, Kayla Wood, Kendra Rimbach, Kenny Hsu, Kenny Nguyen, Keren Gu-Lemberg, Kevin Button, Kevin Liu, Kiel Howe, Krithika Muthukumar, Kyle Luther, Lama Ahmad, Larry Kai, Lauren Itow, Lauren Workman, Leher Pathak, Leo Chen, Li Jing, Lia Guy, Liam Fedus, Liang Zhou, Lien Mamitsuka, Lilian Weng, Lindsay McCallum, Lindsey Held, Long Ouyang, Louis Feuvrier, Lu Zhang, Lukas Kondraciuk, Lukasz Kaiser, Luke Hewitt, Luke Metz, Lyric Doshi, Mada Aflak, Maddie Simens, Madelaine Boyd, Madeleine Thompson, Marat Dukhan, Mark Chen, Mark Gray, Mark Hudnall, Marvin Zhang, Marwan Aljubeh, Mateusz Litwin, Matthew Zeng, Max Johnson, Maya Shetty, Mayank Gupta, Meghan Shah, Mehmet Yatbaz, Meng Jia Yang, Mengchao Zhong, Mia Glaese, Mianna Chen, Michael Janner, Michael Lampe, Michael Petrov, Michael Wu, Michele Wang, Michelle Fradin, Michelle Pokrass, Miguel Castro, Miguel Oom Temudo de Castro, Mikhail Pavlov, Miles Brundage, Miles Wang, Minal Khan, Mira Murati, Mo Bavarian, Molly Lin, Murat Yesildal, Nacho Soto, Natalia Gimelshein, Natalie Cone, Natalie Staudacher, Natalie Summers, Natan LaFontaine, Neil Chowdhury, Nick Ryder, Nick Stathas, Nick Turley, Nik Tezak, Niko Felix, Nithanth Kudige, Nitish Keskar, Noah Deutsch, Noel Bundick, Nora Puckett, Ofir Nachum, Ola Okelola, Oleg Boiko, Oleg Murk, Oliver Jaffe, Olivia Watkins, Olivier Godement, Owen Campbell-Moore, Patrick Chao, Paul McMillan, Pavel Belov, Peng Su, Peter Bak, Peter Bakkum, Peter Deng, Peter Dolan, Peter Hoeschele, Peter Welinder, Phil Tillet, Philip Pronin, Philippe Tillet, Prafulla Dhariwal, Qiming Yuan, Rachel Dias, Rachel Lim, Rahul Arora, Rajan Troll, Randall Lin, Rapha Gontijo Lopes, Raul Puri, Reah Miyara, Reimar Leike, Renaud Gaubert, Reza Zamani, Ricky Wang, Rob Donnelly, Rob Honsby, Rocky Smith, Rohan Sahai, Rohit Ramchandani, Romain Huet, Rory Carmichael, Rowan Zellers, Roy Chen, Ruby Chen, Ruslan Nigmatullin, Ryan Cheu, Saachi Jain, Sam Altman, Sam Schoenholz, Sam Toizer, Samuel Miserendino, Sandhini Agarwal, Sara Culver, Scott Ethersmith, Scott Gray, Sean Grove, Sean Metzger, Shamez Hermani, Shantanu Jain, Shengjia Zhao, Sherwin Wu, Shino Jomoto, Shirong Wu, Shuaiqi, Xia, Sonia Phene, Spencer Papay, Srinivas Narayanan, Steve Coffey, Steve Lee, Stewart Hall, Suchir Balaji, Tal Broda, Tal Stramer, Tao Xu, Tarun Gogineni, Taya Christianson, Ted Sanders, Tejal Patwardhan, Thomas Cunninghman, Thomas

Degry, Thomas Dimson, Thomas Raoux, Thomas Shadwell, Tianhao Zheng, Todd Underwood, Todor Markov, Toki Sherbakov, Tom Rubin, Tom Stasi, Tomer Kaftan, Tristan Heywood, Troy Peterson, Tyce Walters, Tyna Eloundou, Valerie Qi, Veit Moeller, Vinnie Monaco, Vishal Kuo, Vlad Fomenko, Wayne Chang, Weiyi Zheng, Wenda Zhou, Wesam Manassra, Will Sheu, Wojciech Zaremba, Yash Patil, Yilei Qian, Yongjik Kim, Youlong Cheng, Yu Zhang, Yuchen He, Yuchen Zhang, Yujia Jin, Yunxing Dai, and Yury Malkov. 2024. Gpt-4o system card. *Preprint*, arXiv:2410.21276.

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.

Zhongyi Qiu, Kangyi Qiu, Hanjia Lyu, Wei Xiong, and Jiebo Luo. 2024. Semantics preserving emoji recommendation with large language models. *Preprint*, arXiv:2409.10760.

Kamesh R. 2024. Think beyond size: Adaptive prompting for more effective reasoning. *Preprint*, arXiv:2410.08130.

Maria F. Davila R., Sven Groen, Fabian Panse, and Wolfram Wingerath. 2024. Navigating tabular data synthesis research: Understanding user needs and tool capabilities. *Preprint*, arXiv:2405.20959.

Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–7.

Yucheng Ruan, Xiang Lan, Jingying Ma, Yizhi Dong, Kai He, and Mengling Feng. 2024. Language modeling on tabular data: A survey of foundations, techniques and evolution. *Preprint*, arXiv:2408.10548.

Rajat Singh and Srikanta Bedathur. 2023. Embeddings for tabular data: A survey. *Preprint*, arXiv:2302.11777.

Oleg Somov, Alexey Dontsov, and Elena Tutubalina. 2024. AIRI NLP team at EHRSQL 2024 shared task: T5 and logistic regression to the rescue. In *Proceedings of the 6th Clinical Natural Language Processing Workshop*, pages 431–438, Mexico City, Mexico. Association for Computational Linguistics.

Oleg Somov and Elena Tutubalina. 2025. Confidence estimation for error detection in text-to-sql systems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(23):25137–25145.

Yuan Sui, Mengyu Zhou, Mingjie Zhou, Shi Han, and Dongmei Zhang. 2024. Table meets llm: Can large language models understand structured table data? a benchmark and empirical study. *Preprint*, arXiv:2305.13062.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *Preprint*, arXiv:2305.04091.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

Wenting Zhao, Ge Gao, Claire Cardie, and Alexander M. Rush. 2024. I could've asked that: Reformulating unanswerable questions. *Preprint*, arXiv:2407.17469.

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

A Appendix

A.1 Checklists and Dialogue-Inducing Prompts

During testing, models often skipped crucial instructions, leading to incorrect code generation. To enhance reliability, we implemented checklist-based prompts (Cook et al., 2024), enforcing constraints like type matching, entity verification, and logical consistency for more accurate outputs. We also tested dialogue-inducing prompts, where the model simulated a specialist discussion to clarify queries, but this proved superficial, as the model did not actively use the dialogue to correct mistakes.

A.2 Few-shot Prompting

LLMs perform better with contextual examples (Liu et al., 2022), a phenomenon often referred to as few-shot prompting (Reynolds and McDonell, 2021; Brown et al., 2020). This approach involves providing the model with a small number of task-specific examples before asking it to perform the desired task. We also experimented with dynamic few-shot prompting (R, 2024), where the model selects relevant examples based on their similarity

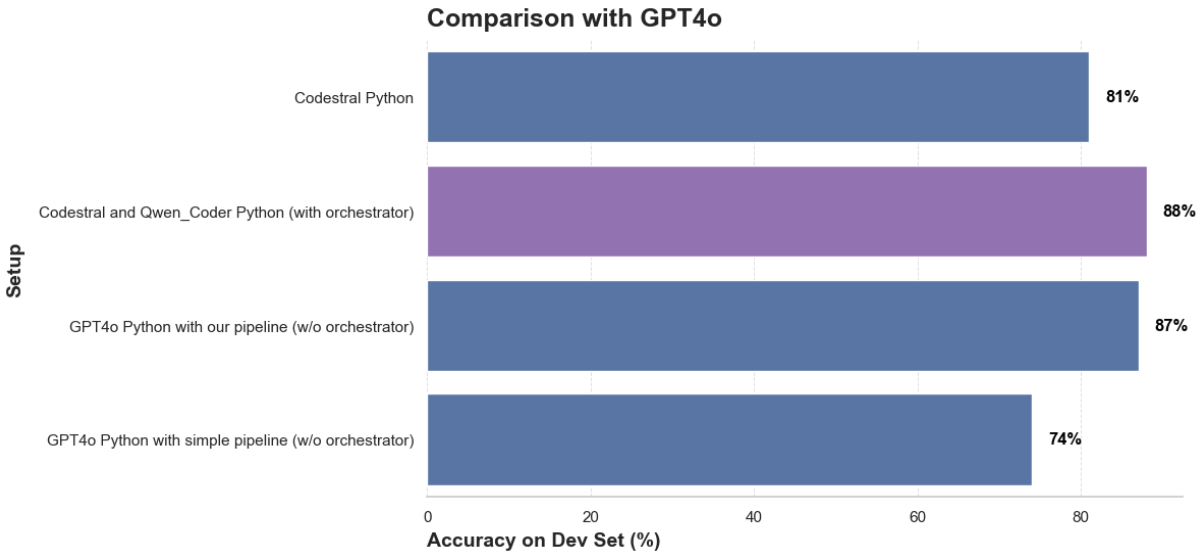

Figure 4: GPT4o comparison with and w/o our pipeline. Llama3.3-70b-instruct used as orchestrator.

to the given question. However, this approach requires generating a large number of high-quality examples for each question type, which is both labor-intensive and time-consuming. Additionally, scaling this method could be challenging, as the number of question types may become too large to manage effectively. Due to these limitations, we consider it beyond the scope of our current work.

Rank	Codabench ID	Team	Accuracy
1	xiongsishi	TeleAI	95.01
2	andreasevag	AILS-NTUA	89.85
20	**anotheroption**	**anotheroption**	**80.08**
	baseline	stable-code-3b-GGUF	26.00

Table 3: Results among both open and closed source models

Statistic	Value
Unique datasets	49
Avg. questions per dataset	20
Boolean answers (T/F)	65% / 35%
Avg. columns per question	2.47
Most common answer types	
Category / Boolean	199 / 198
List[Num] / List[Cat]	198 / 197
Number	196
Columns per dataset (avg./std)	25.98 / 22.74
Question length (avg./std)	61.36 / 18.01

Table 4: Core statistics illustrating the distribution of questions and answers in DataBench.

Data Questions and Formats

Data Type: Number
Q: What is the average age of our employees?
Format: Single numerical value (e.g., 35.2).

Data Type: List[Category]
Q: Unique classifications for employees' education fields?
Format: List of categories (e.g., ["Life Sciences", "Marketing"]).

Data Type: List[Number]
Q: Lowest 5 monthly incomes?
Format: List of numbers (e.g., [2000, 2100, 2200]).

Data Type: Category
Q: Most common role?
Format: Single category (e.g., "Manager").

Data Type: Boolean
Q: Is the highest DailyRate 1499?
Format: `True` or `False`.

Figure 5: Structured data questions and formats.

prompt for python generation (dialogue)

```
1. You are two of the most esteemed Pandas DataScientists engaged in a heated
     and truth-seeking debate. You are presented with a dataframe and a
    question. Begin dialogue by rigorously discussing your reasoning step by
    step, ensuring to address all aspects of the checklist. In your discourse,
     meticulously articulate the variable type necessary to derive the answer
     and confirm that each column referenced is indeed present in the
    dataframe. Conclude your debate by providing the code to answer the
    question, ensuring that the variable result is explicitly assigned to the
     answer. Remember, all code must be presented in a single line, with
    statements separated by semicolons.
2. Refrain from importing any additional libraries beyond pandas and numpy.
3. The dataframe, df, is already populated with data for your analysis; do
    not initialize it, but focus solely on manipulating df to arrive at the
    answer.
4. If the question requires multiple entries, always utilize .tolist() to
    present the results.
5. If the question seeks a single entry, ensure that only one value is output
    , even if multiple entries meet the criteria.

You MUST FOLLOW THE CHECKLIST, ANSWER EACH OF ITS QUESTIONS (REASONING STEP),
     AND ONLY THEN OUTPUT THE FINAL ANSWER BASED ON THOSE ANSWERS:
1) How many values should be in the output?
2) Values (or one value) from which column (only one!) should the answer
    consist of?
3) What should be the type of value in the answer?

Example of a task:
Question: Identify the top 3 departments with the most employees.
<Columns> = ['department', 'employee_id']
<First_row> = ('department': 'HR', 'employee_id': 101)
Reasoning: Count the number of employees in each department, sort, and get
    the top 3. The result should be a list of department names.
Checklist:
1) The output should consist of 3 values.
2) The values should come from the 'department' column.
3) The type of value in the answer should be a list of strings.
Code: result = df['department'].value_counts().nlargest(3).index.tolist()

Your data to process:
<question> = {question}

- Make absolute sure that all columns used in query are present in the table.
<columns_in_the_table> = {[col for col in df.columns]}
<first_rows_of_table> = {df.head(3).to_string()}
YOUR Reasoning through dialogue and Code (Start final code part by "Code:"):
```

Figure 6: Prompt for python generation (dialogue)

prompt for python generation

```
1. You are a best in the field Pandas DataScientist. You are given a
    dataframe and a question. You should spell out your reasoning step by
    step and only then provide code to answer the question. In the reasoning
    state it is essentianl to spell out the answers' variable type that
    should be sufficient to answer the question. Also spell out that each
    column used is indeed presented in the table. In the end of your code the
     variable result must be assigned to the answer to the question. One
    trick: all code should be in one line separated by ; (semi-columns) but
    it is no problem for you.
2. Avoid importing any additional libraries than pandas and numpy.
3. All data is already loaded into df dataframe for you, you MUST NOT
    initialise it, rather present only manipulations on df to calculate the
    answer.
4. If the question ask for several entries alsways use .tolist().
5. If the question ask for one entry, make sure to output only one, even if
    multiple qualify.


<...> (same as previous prompt)
```

Figure 7: Prompt for python generation (without dialogue)

prompt for self-correction

```
"The following solutions failed for the task: \"{question}\"\n\n"
        + '\n'.join([f'Solution {i+1} Error:\n{traceback}\n' for i, traceback
     in enumerate(tracebacks)])
        + "\nDF info: \n"
        + "<columns_to_use> = " + str([(col, str(df[col].dtype)) for col in
    df.columns]) + "\n"
        + "<first_row_of_table> = " + str(df.head(1).to_dict(orient='records
    ')[0]) + "\n"
        + "YOUR answer in a single line of pandas code:\n"
        + "Please craft a new solution considering these tracebacks. Output
    only fixed solution in one line:\n"
```

Figure 8: Prompt for self-correction

prompt for orchestrator

```
Examples of deducing answer types:
1. If the question is "Do we have respondents who have shifted their voting
    preference?" the answer type is **Boolean** because the response should
    be True/False.
2. If the question is "How many respondents participated in the survey?" the
    answer type is **Integer**
3. If the question is "List the respondents who preferred candidate X?" the
    answer type is **List** because the response requires a collection of
    values.
4. If the question is "What is the average age of respondents?" the answer
    type is **Number** because the response should be a decimal value.
5. If the question is "What is the name of the candidate with the highest
    votes?" the answer type is **String** because the response is a single
    textual value.

Given the following solutions and their results for the task: "{question}"

{'   '.join([f'Solution Number {i+1}:  Code: {r["code"]} Answer: {str(r["
    result"])[:50]} (may be truncated) ' for i, r in enumerate(solutions)])}

Instructions:
- Deduce the most probable and logical result to answer the given question.
    Then output the number of the chosen answer.
- If you are presented with end-to-end solution, it should not be trusted for
     numerical questions, but it is okay for other questions.
- Make absolute sure that all columns used in solutions are present in the
    table. SQL query may use additional double quotes around column names, it'
    s okay, always put them. Real Tables columns are: {df.columns}
- If the column name contain emoji or unicode character make sure to also
    include it in the column names in the query.
- If several solutions are correct, return the lowest number of the correct
    solution.
- Otherwise, return the solution number that is most likely correct.
- If the question ask for one entry, make sure to output only one, even if
    multiple qualify.

You should spell out your reasoning step by step and only then provide code
    to answer the question. In the reasoning state it is essentianl to spell
    out the answers' variable type that should be sufficient to answer the
    question. Also spell out that each column used is indeed presented in the
     table. The most important part in your reasoning should be dedicated to
    comparing answers(results) from models and deducing which result is the
    most likely to be correct, then choose the model having this answer.
First, predict the answer type for the question. Then give your answer which
    is just number of correct answer with predicted variable type.  Start
    reasoning part with "REASONING:" and final answer with "ANSWER:".
```

Figure 9: Prompt for orchestrator

prompt for SQL generation

```
The task is: {question}
Here are some examples of SQL queries for similar tasks:
Example 1:
Task: Is there any entry where age is greater than 30?
REASONING:
1. Identify the column of interest, which is 'age'.
2. Determine the condition to check, which is 'age > 30'.
3. Use the SELECT statement to retrieve a boolean result indicating the
    presence of such entries.
4. Apply the WHERE clause to filter rows based on the condition 'age > 30'.
5. Use the EXISTS clause to ensure the query outputs 'True' if any row
    matches the condition, otherwise 'False'.
6. Verify that the query outputs 'True' or 'False' when presented with a yes
    or no question.
CODE: ```SELECT CASE WHEN EXISTS(SELECT 1 FROM temp_table WHERE "age" > 30)
    THEN 'True' ELSE 'False' END;```
Example 2:
Task: Count the number of entries with a salary above 50000.
REASONING:
1. Identify the column of interest, which is 'salary'.
2. Determine the condition to filter the data, which is 'salary > 50000'.
3. Use the SELECT COUNT(*) statement to count the number of rows that meet
    the condition.
4. Apply the WHERE clause to filter rows based on the condition 'salary >
    50000'.
5. Ensure the table name is 'temp_table' and the column name is enclosed in
    double quotes to handle any spaces or special characters.
CODE: ```SELECT COUNT(*) FROM temp_table WHERE [salary] > 50000;```
Write a correct fault-proof SQL SELECT query that solves this precise task.
Rules:
- Your SQL query should be simple with just SELECT statement, without WITH
    clauses.
- Your SQL query should output the answer, without a need to make any
    intermediate calculations after its finish
- Make sure not to use "TOP" operation as it is not presented in SQLite
- If present with YES or NO question, Query MUST return 'True' or 'False'
- If the question asks about several values, your query should return a list
- Equip each string literal into double quotes
- Use COALESCE( ..., 0) to answer with 0 if no rows are found and the
    question asks for the number of something.
Table name is 'temp_table'.
Available columns and types: {', '.join([f"{col}: {str(type(df[col].iloc[0]))
    }" for col in column_names])}
Top 3 rows with highest cosine similarity: {
    get_relevant_rows_by_cosine_similarity(df, question, ai_client).head(3).
    to_markdown()}
YOUR RESPONSE:
```

Figure 10: Prompt for SQL-generation

prompt for E2E model

```
Question: {question}
    Dataset: {dataset_text}

    Analyze the data. Provide your final answer to the question based on the
    data.
    If the question assumes several answers, use a list. Your answer should
    be in the form of one of the following:
    1. Boolean (True/False)
    2. List (e.g., ['Tree', 'Stone'])
    3. Number (e.g., 5)
    4. String (e.g., 'Spanish')

    Give extensive reasoning and then fianlly provide the answer starting
    with string "Final Answer:" in one of the four formats presented above (
    Boolean, List, Number, String). Your response should then be finished.
```

Figure 11: Prompt for E2E model