TabaQA at SemEval-2025 Task 8: Column Augmented Generation for Question Answering over Tabular Data

Ekaterina Antropova[1] **Egor Kratkov**[1] **Roman Derunets**[4,5]
Margarita Trofimova[1] **Ivan Bondarenko**[4] **Alexander Panchenko**[3,2]
Vasily Konovalov[2,1] **Maksim Savkin**[1]
[1]Moscow Institute of Physics and Technology
[2]AIRI [3]Skoltech [4]Novosibirsk State University
[5]Siberian Neuronets LLC
{antropova.eg, savkin.mk, vasily.konovalov}@phystech.edu

Abstract

The DataBench shared task in the SemEval-2025 competition aims to tackle the problem of question answering (QA) from tabular data. Given the diversity of the structure of tables, there are different approaches to retrieving the answer. Although Retrieval-Augmented Generation is a viable solution, extracting relevant information from tables remains a significant challenge. In addition, the table can be prohibitively large for direct integration into the LLM context. In this paper, we address QA over tabular data first by identifying relevant columns that might contain the answers, then the LLM generates answers by providing the context of the relevant columns, and finally, the LLM refines its answers. This approach secured us 7th place in the DataBench lite category.

1 Introduction

Question Answering (QA) is a long-standing challenge in artificial intelligence, with numerous variations and applications. QA systems are extensively used in a variety of real-world scenarios, such as virtual assistants and chatbots for customer support. They serve as powerful tools for extracting relevant information from large and diverse datasets.

A crucial challenge arises when the required information is not just embedded in natural language but also stored in structured formats, such as tabular data. The primary challenge in tabular QA stems from the need to bridge the gap between structured data representation and natural language understanding. While substantial progress has been made in developing models that handle either natural language or structured data independently, effectively integrating both remains an open research problem.

Traditional retrieval methods struggle to select relevant table segments, and large language models (LLMs) can be inefficient when tasked with processing extensive tabular data directly. To advance research in this area, the DataBench shared task (Os'es Grijalba et al., 2025) was introduced, focusing on answering questions from tabular datasets. In particular, the shared task proposes to answer the question on the DataBench (Grijalba et al., 2024) datasets and a smaller DataBench lite version (a reduced version of each dataset from the original DataBench). In this paper, we focus primarily on a lite subtask. Our approach incorporates several key strategies inspired by existing methodologies. First, we observe that all the questions can be answered using information from only three relevant columns. We leverage column augmented generation (CAG), a technique that dynamically selects relevant columns based on LLM-generated hints and helps retain only the most relevant columns. CAG addresses multiple challenges simultaneously by filtering out irrelevant columns, reducing noise, and significantly shrinking the table size, which enhances model efficiency. We then explore the effect of several prompting techniques: 1) iterative self-refinement employs a feedback loop where the model iteratively refines its own responses; 2) self-consistency answer selection improves the reliability of stochastic LLM generations by aggregating multiple generated answers; 3) Chain-of-thought (CoT) allows the model to produce intermediate reasoning steps, which increases its reasoning capabilities.

By systematically analyzing the impact of each individual modification, we investigate how these techniques can be effectively combined to make our best performing solution. Our submission utilizes only open-source and still achieves competitive results: we rank 7th out of 35 teams on the DataBench lite leaderboard. Our findings highlight the potential of combining structured data processing techniques with robust QA methodologies to improve tabular QA capabilities.

Our main contributions are threefold:

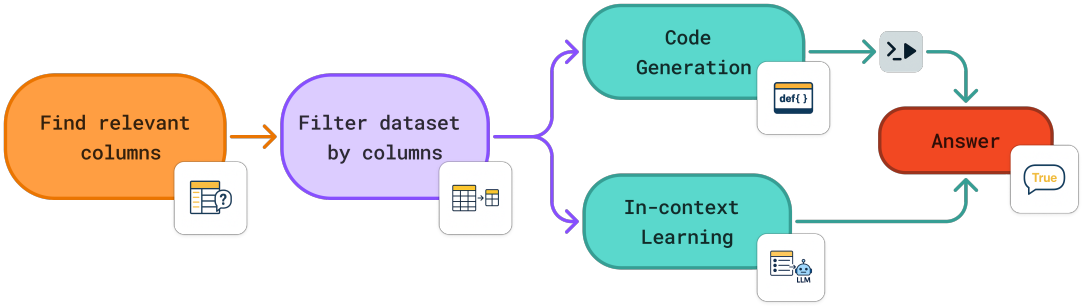

Figure 1: An illustration of CAG pipeline. Step 1) Extract column relevant to the question based on their names, see Appendix A. Step 2) Remove all irrelevant columns from the tables in dataset. Step 3) Code generation: Insert column names and descriptions, question in the prompt and generate code (see Appendix A), then execute it; In-context learning: Insert filtered tables, column descriptions and question in the prompt and generate an answer, see Appendix A. Reasoning prompting techniques, see Section 5.2, are applied along with the Step 3.

- **Column Augmented Generation** We enhance context retrieval by dynamically selecting relevant table segments using LLM-generated column hints, improving the quality of retrieved evidence.
- **Reasoning Prompts** We systematically experiment with different reasoning strategies to enhance LLM reasoning capabilities and identify the most effective prompting techniques.
- **Reasoning Scaling** We analyze how different levels of reasoning complexity influence performance, demonstrating that increased reasoning depth can significantly improve tabular QA accuracy.

2 Related Work

Prior work in table-based question answering can be broadly categorized into two main paradigms: **Text-to-Code** and **Retrieval-Augmented Generation** .

Text-to-Code methods translate natural language queries into executable code, typically SQL. Recent advances in this area have been driven by two key factors. First, improvements in pre-trained language models have led to richer semantic representations of queries and table content (Konovalov and Tumunbayarova, 2018). Second, a deeper understanding of the role that table structure plays in query interpretation has informed more accurate model architectures (Li et al., 2023).

Within this paradigm, several sub-approaches have emerged. Schema-aware parsing models, such as TAPAS (Herzig et al., 2020), T5-SQL (Arcadinho et al., 2022), TAPEX (Liu et al., 2022), and OmniTab (Jiang et al., 2022), explicitly encode schema information (e.g., column names and data types) to guide SQL generation. Another line of work uses intermediate logical forms, as in DIN-SQL (Pourreza and Rafiei, 2023), which first parse questions into structured operations before converting them to SQL, increasing the robustness of the model. Extensions to text-to-code approaches include systems like text2pandas (Venturi, 2023), which generate executable `pandas` code. These methods offer more flexible data manipulation and are especially useful for non-standard or heterogeneous table formats.

Retrieval-Augmented Generation approaches condition the model output on evidence retrieved from the knowledge base (table) (Belikova et al., 2024). These methods enhance the grounding of generated answers and improve performance on complex queries.

Several retrieval strategies have been proposed within the RAG framework. Aushev et al. (2025) proposed to combine retrieval with techniques to enhance LLM attention on the retrieved context rather than on its own knowledge. Table segment retrieval, as used in TableRAG (Chen et al., 2024), identifies relevant portions of the table for conditioning, helping models focus on the most informative regions. Row-column retrieval methods, including ITR (Lin et al., 2023) and TAP4LLM (Sui et al., 2024), further improve scalability by selectively encoding and retrieving key rows and columns.

In TableRAG (Chen et al., 2024), schema retrieval is used to identify relevant table columns by constructing structured representations of each column (including name, type, min/max values, and most frequent values). Given a query, a lan-

guage model generates a list of candidate columns, and retrieval is performed using vector search (e.g., FAISS), keyword search (BM25), or hybrid search strategies combining both. Embeddings from the *BAAI/bge-large-en-v1.5* model (Xiao et al., 2023) are used for vectorization. Similarly, TAP4LLM (Sui et al., 2024) introduces Table Sampling, where each column is embedded and the top-k columns are retrieved based on their semantic proximity to the query, again using FAISS. To enhance retrieval, keyword-based and hybrid search strategies are also explored.

ReasTAP (Zhao et al., 2022) demonstrates that high-level reasoning over tables can be incorporated during pre-training without requiring task-specific architectures. Although these methods reduce input length and improve focus, they often incur additional computational costs and may struggle with degraded embedding quality on longer sequences.

Finally, hybrid approaches combine retrieval with execution or synthesis mechanisms. For example, ReAcTable (Zhang et al., 2024) integrates context retrieval with program synthesis and iterative refinement, bridging the text-to-SQL and RAG paradigms. H-STAR (Abhyankar et al., 2024) further extends this idea by employing dynamic hybrid prompting, adapting between structured and unstructured strategies depending on the complexity of the query. These systems highlight the potential of combining multiple paradigms to balance efficiency, scalability, and accuracy.

Recent progress, particularly inspired by TableRAG (Chen et al., 2024), underscores the importance of selectively retrieving and conditioning on relevant table segments. Such methods have shown strong performance gains, especially on complex or noisy tables. However, their effectiveness remains closely tied to the precision of the retrieval component — inaccuracies at this stage can significantly degrade downstream performance.

The approaches to develop a QA system based on the tabular data can be further enhanced to be based on knowledge graphs (Sakhovskiy et al., 2024).

3 Dataset

The DataBench dataset consists of 65 publicly available datasets, with 3 269 975 rows and 1615 columns in total, and 1300 questions in 5 domains. The dataset is split into three sections: training (comprising 988 questions), testing (featuring 522 questions and 517 questions after changes), and development (consisting of 320 questions) (Grijalba et al., 2024). DataBench Lite is obtained by shortening the table for each question in the DataBench dataset so that 20 columns and 20 rows remain. Examples of multi-column questions (requiring more than two columns to generate a correct answer) are provided in the Appendix C.

Column types presented in DataBench:

- Boolean: Valid answers include True/False, Y/N, Yes/No (all case insensitive).
- Category: A value of a cell (or a substring of a cell) in the dataset.
- Number: A numerical value from a cell in the dataset that may represent a computed statistic (e.g., average, maximum, minimum).
- List[category]: A list containing a fixed number of categories. The expected format is: "['cat', 'dog']". Pay attention to the wording of the question to determine if uniqueness is required or if repeated values are allowed.
- List[number]: Similar to List[category], but with numbers as its elements.

4 Evaluation

Participants must answer questions using only the provided dataset, including development and training sets, without external data. The accuracy scores for DataBench and DataBench lite are ranked separately. An automated evaluation framework[1] streamlines the process, allowing customization of prompt building, model calling, and result evaluation, while supporting batch processing and standardized response handling.

The default evaluation function compares submissions against the truth set, featuring:

- Null: Treats null-like values as equivalent.
- Boolean: Normalizes inputs and checks against true/false lists.
- Category/String: Strips and compares strings, with date parsing for format variations.
- Number: The values are rounded to two decimal places for tolerance.

[1] https://github.com/jorses/databench_eval

Model	Precision	Recall	F1
CAG			
Llama-3.3-70B-Instruct	84.60	98.33	90.95
Llama-3.1-8B-Instruct	61.80	96.90	75.46
Table Sampling (k=3)			
bge-large-en-v1.5	26.10	52.27	34.82
BM25	14.64	27.92	19.21
bge-large-en-v1.5 + BM25	20.15	62.29	30.46
TableRAG			
Llama-3.3-70B-Instruct			
bge-large-en-v1.5	16.41	32.70	21.85
BM25	26.96	27.92	27.43
bge-large-en-v1.5 + BM25	22.28	50.84	30.98

Table 1: The micro-averaged results for detecting the required columns to form the answer to the question (on the dev dataset). For the Table Sampling method, we consider k=3 based on the considerations given in the section D.

- List[Category]: Normalizes and checks set equivalence (order-agnostic).
- List[Number]: Normalizes, rounds, and ensures order-agnostic equivalence.

In addition, the organizers decided to manually review the results of the top scores to ensure fairness and accuracy in determining the winner.

5 System Overview

5.1 Column Augmented Generation

The Column Augmented Generation (CAG) approach is a two-step method inspired by Retrieval-Augmented Generation techniques. Since large tables cannot be easily incorporated into an LLM's context due to size constraints, the first step in our pipeline involves identifying the most relevant columns needed to answer a given question (see Appendix A for the relevant prompt). This column selection process is treated as a multi-label classification task, evaluated using micro-averaged precision, recall, and F1 score (see Table 1). The selected column names are then validated for format compliance, and if necessary, the generated response is refined or corrected.

Once the relevant columns are identified, the second step involves prompting the LLM again, this time with the extracted column data included as context. This ensures that the model focuses only on the most relevant information when generating an answer. The CAG approach aligns closely with techniques used in TableRAG (Chen et al., 2024), improving both passage quality and model efficiency by dynamically narrowing the input scope.

5.2 Reasoning Prompting Techniques

In addition to the CAG technique, we incorporated advanced prompt engineering methods to enhance reasoning abilities, specifically Chain-of-Thought (CoT) (Wei et al., 2022), Self-Consistency (Wang et al., 2023), and Self-Refine (Madaan et al., 2023), see all prompts in Appendix B. These techniques can be applied independently or in combination with CAG to improve its effectiveness.

The Self-Consistency method involves calling the model multiple times and aggregating the results using a majority vote to determine the most frequent answer. We experimented with different numbers of model calls, such as 3, 5, and 10, to assess its impact, see Figure 3.

The Chain-of-Thought technique encourages step-by-step reasoning by instructing the model to break down its thought process. This approach improves the focus of the model on intermediate reasoning steps while also increasing token generation. To implement this, we used few-shot prompting with explicit CoT examples. Additionally, we explored the combination of CoT and Self-Consistency to further refine the reasoning process.

Finally, we tried applying Self-Refine as a post-generation technique. The model was instructed to evaluate its own responses, providing feedback on correctness and answer format. This feedback was then used to guide subsequent answers, leading to iterative improvements.

In our code generation method, we generated Python code (which yielded better results than SQL) and then executed it. If the code generated an error, the self-refine method was applied with the exception given to the prompt. Chain-of-Thought and self-consistency were also integrated to improve the effectiveness. However, we achieved better results on DataBench lite using the Table-to-text method with the Chain-of-Thought and self-consistency.

6 Experimental Setup

For our experimental setup we used two open LMs: (1) `Llama-3.1-8B-Instruct`[2] is a language

[2] https://hf.co/meta-llama/Llama-3.1-8B-Instruct

Model	EM
Python Code Generation	
CodeLlama-7B*	30.3
CodeLlama-13B*	33.1
Llama-3.1-8B-Instruct	45.98
+ CAG	51.15 (+5.17)
Llama-3.3-70B-Instruct	70.5
+ CAG	65.51 (-4.99)
In-Context Learning	
Llama-2-7B*	14.8
Llama-2-13B*	20.7
Llama-3.1-8B-Instruct	27.78
+ CAG	37.93 (+10.15)
Llama-3.3-70B-Instruct	49.81
+ CAG	64.37 (+14.56)
DeepSeek-R1-32B	77.78
+ CAG	83.52 (+5.74)

Table 2: Results on the DataBench Lite test set. EM denotes the Exact Match (see Section 4); "*" denotes baselines provided by the competition organizers, see (Grijalba et al., 2024). +CAG indicates the use of our proposed CAG method. Our approach consistently outperforms baselines.

Model	EM
Llama-3.3-70B-Instruct	49.81
+CAG	64.37 (+14.56)
+CAG+Ref	69.36 (+19.55)
+CAG+CoT	81.03 (+31.22)
+CAG+CoT+Ref(1)	77.59 (+27.78)
+CAG+CoT+Cons(10)	**81.99** (+32.18)

Table 3: Results on the DataBench Lite test set. This table demonstrates the impact of various prompting techniques on the overall performance, applied in in-context learning settings. The techniques include: CAG (Column-Augmented Generation), CoT (Chain-of-Thought instruction included in the prompt), Ref (Self-Refine prompt), and Cons (Self-Consistency answer selection). For further details on how these prompting methods were implemented, refer to Section 5.2 and Appendix B.

model with 8 billion parameters and is optimized for conversational applications, supporting a contextual length of up to 128 thousand tokens; (2) `Llama-3.3-70B-Instruct`[3] is a large language model with 70 billion parameters and is optimized for conversational applications, supporting a contextual length of up to 128 thousand tokens (Touvron et al., 2023). In addition, we evaluate our CAG approach on DeepSeek-R1-32B[4] that has shown advanced reasoning abilities over the LMs of a comparable number of parameters (DeepSeek-AI, 2025). All LMs were evaluated for `temperature = 0.7`. The `BAAI/bge-large-en-v1.5`[5] was also used as an embedding model.

7 Results and Discussion

Our main results on the DataBench lite test split are summarized in Table 2.

7.1 Required Columns Generation

The findings confirm that the 70B model consistently outperforms the smaller 8B model across all settings. This is evident in the first step of the CAG pipeline, where LLMs identify relevant columns, as shown in Table 1. The 70B Llama achieves a micro-average F1 score of 90.95%, significantly outperforming the 8B models.

We also evaluated retrieval-based baselines. In TableRAG, the best results were achieved with the bge-large-en-v1.5 embeddings combined with BM25 retrieval, reaching an F1 score of 30.98%. In Table Sampling (from TAP4LLM), the best result was obtained with the bge-large-en-v1.5 model, achieving an F1 score of 34.82%. On the DataBench Lite dataset, Table Sampling outperformed TableRAG but still showed lower performance compared to CAG. This could be due to the fact that TableRAG and Table Sampling approaches typically perform better on large tables, while DataBench Lite consists of relatively small tables (20 rows and 20 columns) (Table 1). For Table Sampling, we found that the optimal number of retrieved columns is $k = 3$, as detailed in Appendix D.

7.2 Column Augmented Generation

Table 2 further demonstrates that the CAG technique significantly improves performance across all prompting strategies that do not require code execution, producing an absolute percentage increase of 10.1 and 12.6 for the 8B and 70B models, respectively. When combined with Chain-of-Thought, performance improves even further, with

[3] https://hf.co/meta-llama/Llama-3.3-70B-Instruct

[4] https://hf.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-32B

[5] https://hf.co/BAAI/bge-large-en-v1.5

percentage gains of 26.8 and 19.5 over the baseline prompt.

7.3 Reasoning Prompting Techniques

Reasoning-based prompting techniques, including CoT, self-refinement, and self-consistency, contribute to additional accuracy gains when combined with CAG, see Table 3.

By aggregating multiple responses through self-consistency, we aimed to mitigate the randomization effect of suboptimal prompts. This approach proved effective, boosting EM scores by 0.96 absolute percent when compared to the CAG+CoT approach. Experiments with different numbers of reasoning paths indicate a stable increase in EM scores up to 10 paths, see Figure 3. Beyond this, setting up to 40 paths for the 8B model resulted in only a marginal gain, so due to the high computational cost, we opted to settle on 10 paths.

Self-refinement enables the model to verify and correct its responses, improving its reliability, particularly. However, while refinement techniques improved accuracy through self-correction, pure CoT performed better. CoT may have confused the model by already including iterative refinements, making added steps redundant.

DeepSeek-R1-32B, which performs extended reasoning before generating an answer, emerged as a strong alternative to traditional prompting techniques. It achieved an EM of 77.78%, which increased to 83.52 with CAG—the highest result obtained. Increasing test-time compute, whether through the progressive combination of prompting techniques (CAG $\rightarrow$ CAG+CoT $\rightarrow$ CAG+CoT+self-consistency) or through the use of reasoning models, consistently led to improved results. These findings highlight the benefits of test-time compute scaling for tabular QA.

Conclusion

In this work we applied the CAG approach for SemEval-2025 Task 8: TabularQA competition. The CAG is a two-step approach in which we first use an LLM to identify columns relevant to the question. The LLM then generates an answer based on the content of these columns. Using the CAG approach, we outperformed all of the baselines mentioned. In addition, we applied reasoning prompt engineering techniques to further improve the generated answers. Moreover, we revealed that CAG based on `DeepSeek-R1-32B` outperformed all sizes of Llamas, which confirms the superiority of the reasoning language models for TabularQA.

The CAG approach can be used independently or integrated within an NLP framework such as DeepPavlov (Savkin et al., 2024).

Limitations

Our experiments were limited by time and computational resources, which prevented us from fully optimizing the suggested prompting techniques. Wang et al. (2023) noted that increasing the number of reasoning paths could further improve performance. Additionally, we did not compute scores for the combination of self-refine and self-consistency prompting, as the computational cost of such an approach would have been prohibitively high. The `DeepSeek-R1-32B` model also presented challenges, as it generated extensive reasoning traces, often exceeding 16 000 tokens for even simple prompts, making it impractical to test additional prompting techniques within our resource constraints. Furthermore, we were unable to evaluate all potentially valuable open-source LLMs and did not test any models. Since we primarily focused on no-code solutions, we did not explore code-generation-specific optimizations in detail, though all the described approaches can be easily transferred to code generation.

References

Nikhil Abhyankar, Vivek Gupta, Dan Roth, and Chandan K. Reddy. 2024. H-STAR: llm-driven hybrid sql-text adaptive reasoning on tables. *CoRR*, abs/2407.05952.

Samuel Arcadinho, David Aparício, Hugo Veiga, and António Alegria. 2022. T5ql: Taming language models for sql generation. *Preprint*, arXiv:2209.10254.

Islam Aushev, Egor Kratkov, Evgenii Nikolaev, Andrei Glinskii, Vasilii Krikunov, Alexander Panchenko, Vasily Konovalov, and Julia Belikova. 2025. RAGulator: Effective RAG for regulatory question answering. In *Proceedings of the 1st Regulatory NLP Workshop (RegNLP 2025)*, pages 114–120, Abu Dhabi, UAE. Association for Computational Linguistics.

Julia Belikova, Evegeniy Beliakin, and Vasily Konovalov. 2024. JellyBell at TextGraphs-17 shared task: Fusing large language models with external knowledge for enhanced question answering. In *Proceedings of TextGraphs-17: Graph-based Methods for Natural Language Processing*, pages 154–160, Bangkok, Thailand. Association for Computational Linguistics.

Si-An Chen, Lesly Miculicich, Julian Eisenschlos, Zifeng Wang, Zilong Wang, Yanfei Chen, Yasuhisa Fujii, Hsuan-Tien Lin, Chen-Yu Lee, and Tomas Pfister. 2024. Tablerag: Million-token table understanding with language models. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Jorge Osés Grijalba, Luis Alfonso Ureña López, Eugenio Martínez Cámara, and José 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, 20-25 May, 2024, Torino, Italy*, pages 13471–13488. ELRA and ICCL.

Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Eisenschlos. 2020. Tapas: Weakly supervised table parsing via pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Zhengbao Jiang, Yi Mao, Pengcheng He, Graham Neubig, and Weizhu Chen. 2022. Omnitab: Pretraining with natural and synthetic data for few-shot table-based question answering. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*, pages 932–942. Association for Computational Linguistics.

Vasily Konovalov and Zhargal Tumunbayarova. 2018. Learning word embeddings for low resourse languages: The case of buryat. In *Komp'juternaja Lingvistika i Intellektual'nye Tehnologiithis*, pages 331–341.

Shuqin Li, Kaibin Zhou, Zeyang Zhuang, Haofen Wang, and Jun Ma. 2023. Towards text-to-sql over aggregate tables. *Data Intell.*, 5(2):457–474.

Weizhe Lin, Rexhina Blloshmi, Bill Byrne, Adrià de Gispert, and Gonzalo Iglesias. 2023. An inner table retriever for robust table question answering. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 9909–9926. Association for Computational Linguistics.

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2022. TAPEX: table pre-training via learning a neural SQL executor. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Jorge Os'es Grijalba, Luis Alfonso Ure na-L'opez, Eugenio Mart'inez C'amara, 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.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Andrey Sakhovskiy, Mikhail Salnikov, Irina Nikishina, Aida Usmanova, Angelie Kraft, Cedric Möller, Debayan Banerjee, Junbo Huang, Longquan Jiang, Rana Abdullah, Xi Yan, Dmitry Ustalov, Elena Tutubalina, Ricardo Usbeck, and Alexander Panchenko. 2024. TextGraphs 2024 shared task on text-graph representations for knowledge graph question answering. In *Proceedings of TextGraphs-17: Graph-based Methods for Natural Language Processing*, pages 116–125, Bangkok, Thailand. Association for Computational Linguistics.

Maksim Savkin, Anastasia Voznyuk, Fedor Ignatov, Anna Korzanova, Dmitry Karpov, Alexander Popov, and Vasily Konovalov. 2024. DeepPavlov 1.0: Your gateway to advanced NLP models backed by transformers and transfer learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 465–474, Miami, Florida, USA. Association for Computational Linguistics.

Yuan Sui, Jiaru Zou, Mengyu Zhou, Xinyi He, Lun Du, Shi Han, and Dongmei Zhang. 2024. TAP4LLM: table provider on sampling, augmenting, and packing semi-structured data for large language model reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, pages 10306–10323. Association for Computational Linguistics.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971.

Gabriele Venturi. 2023. PandaAI: the conversational data analysis framework.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-pack: Packaged resources to advance general chinese embedding. *CoRR*, abs/2309.07597.

Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. 2024. Reactable: Enhancing react for table question answering. *Proc. VLDB Endow.*, 17(8):1981–1994.

Yilun Zhao, Linyong Nan, Zhenting Qi, Rui Zhang, and Dragomir Radev. 2022. Reastap: Injecting table reasoning skills during pre-training via synthetic reasoning examples. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 9006–9018. Association for Computational Linguistics.

A CAG Baseline Prompts

Prompt for extracting relevant columns

Given a table contains columns with names <*list of column names*>, I want to answer a question: <*question*>.

Please select a few column names from the list <*list of column names*>, the values of which will help answer the question.

Please provide a list of column names in the list format without any additional explanations.

Example of output: ["column name1", "column name2", "column name3"]

Baseline in-context learning prompt

You are a data analysis assistant. Given a table with the following **columns**: <*list of retrieved column names*>, and the **table data**: <*table as string*>, answer the following **question**: <*question*>. Your task is to analyze the table and provide a concise answer to the question. The answer must strictly adhere to one of the following formats, depending on the question:

1. **Boolean**: True or False
2. **Category**: A single category as a Python string (e.g., "category_name")
3. **Number**: A single number (e.g., 42)
4. **List[Category]**: A list of strings (e.g., ['category_1', 'category_2'])
5. **List[Number]**: A list of numbers (e.g., [1, 2, 3])

Rules:
- Do not include any additional text, comments, or explanations.

Examples:
Question 1: <*question*>
Answer 1: <*answer*>
. . .
Now, provide the answer to the following question: <*question*>

Baseline code generation prompt

 # Examples:
Example 1:
Question: <*question*>
Answer: <*answer*>
Example 2: . . .
—

 # Instructions:
Implement the following function in one line.
Answer with the function only with no additional explanations.
The function must return one of these types: **bool, int, str, list[int], or list[str].**

 # Formatting rules:
1) Answer with the function only. **No comments, no additional explanations.**

```
# 2) Your answer should start with: def answer
# 3) The function should be implemented in one line

# Your task: complete the following function in one line. It should give the answer to: {question}
def answer(df: pd.DataFrame):
    df.columns = {list_columns}
    return
```

B Reasoning Prompting Techniques

In-context learning prompt with Chain-of-Thought

Example 1:
Table columns: <*list of column names*>
Question: <*question*>
Answer: <*answer*>

Example 2: . . .
—

You are a data analysis assistant. You are given a table and a question. Answer the question based only on the information in the table. The answer must be in one of the 5 following formats:
- **Boolean** (True or False) if the question requires a yes/no answer.
Example: [ANSWER] True
- **Number** if the question requires a numerical response.
Example: [ANSWER] 15
- **Category** (a string) if the question requires a categorical response.
Example: [ANSWER] London
- **List[Category]**: A list of strings.
Example: [ANSWER] ['San Francisco', 'New York', 'Wuhan', 'Bangalore']
- **List[Number]**: A list of numbers.
Example: [ANSWER] [1, 2, 3, 4, 5]

Rules:
- Start the answer with: "Let's think step by step". Then give your reasoning to your answer.
- Finish your answer with [ANSWER] and give the final answer.
- [ANSWER] part should contain only one of the following types: boolean, number, category, list[category], list[number]
- Don't give any additional text, comments or explanation in the [ANSWER] part.

Table: <*table as string*>
Table columns: <*list of column names*>
Question: <*question*>
Answer:
Let's think step by step.

Self-Refine Step 1: Prompt for extracting relevant columns

You are an AI assistant specialized in data analysis.
Given a table with the following columns: <*list of column names*>
Task: Select the most relevant columns that are necessary to answer the following question:**"question"**

Step-by-step reasoning:

1. Identify the key concepts in the question.
2. Match these concepts to the relevant columns in the table.
3. If multiple columns could provide similar information, select the **most informative** ones.
4. Ensure that the chosen columns are **minimal yet sufficient**.

Example:
Question: "What is the average salary of employees in the IT department?"

- **Step 1:** Key concepts → "average salary", "IT department"
- **Step 2:** Relevant columns → ["Salary", "Department"]
- **Step 3:** "Salary" contains numerical salary data; "Department" helps filter IT employees
- **Step 4:** Selected columns → ["Salary", "Department"]

IMPORTANT:

- Your step-by-step reasoning **MUST NOT** exceed 5 sentences.
- Your final JSON **MUST** be included within the response.
- If you fail to comply, your response will be discarded as invalid.

Your Task:

Perform the same step-by-step reasoning for the given question and return the final selected columns in **strict JSON format**.

Example Outputs:
Example 1 (if the question is about employee age):
json
["Age"]

Example 2 (if the question is about employee salary growth):
json
["Salary", "YearsAtCompany", "PercentSalaryHike"]

Example 3 (if the question is about employee job satisfaction and department):
json
["Department", "JobSatisfaction"]

CRITICAL REQUIREMENTS:

- You **MUST** return your response in two parts:
    1. **Your step-by-step reasoning.**

2. **The final validated response in STRICT JSON FORMAT.**

- The JSON response **must be the last thing in your answer** and formatted correctly.

Now, provide your response in the exact same JSON format.

Self-Refine Step2: Prompt for validating the retrieved columns

You have selected the following columns to answer the question: <*question*>

Selected columns: <*list of retrieved column names*>
Your previous reasoning was: <*list of column names*>
Full list of available columns in the dataset: <*list of column names*>

Validation Task

- If the selection is **correct and minimal**, return **"VALID"**.
- If the selection is **incorrect, incomplete, or redundant**, return the corrected column list in **strict JSON format**.
- If additional columns are **necessary**, add them.
- If some columns are **not needed**, remove them.

New Validation Rules

- You **must** justify any change in column selection.
- If the selection is **"VALID"**, explain **why**.
- If changes are needed, return **only** the corrected column list in JSON format.

IMPORTANT:

- Your step-by-step reasoning **MUST NOT** exceed 5 sentences.
- Your final JSON **MUST** be included within the response.
- If you fail to comply, your response will be discarded as invalid.

Example Outputs (JSON format)

If the selected columns are correct and minimal:
json
"VALID"

If some columns are incorrect or missing:
json
["Department", "EmployeeCount"]

If unnecessary columns are included:
json
["Age"]

CRITICAL REQUIREMENTS

- You **MUST** return your response in two parts:
  1. **Your step-by-step reasoning.**
  2. **The final validated response in STRICT JSON FORMAT.**
- The JSON response **must be the last thing in your answer** and formatted correctly.

Now, validate the selected columns and return your response in the correct JSON format.

Self-Refine Step 3: Prompt for generating an answer

You are a data analysis assistant. Given the following table: <*retrieved columns*>

Your previous reasoning was: <*response*>

Step-by-step reasoning before answering:

1. Identify the key data points in the table that are needed to answer the question.
2. Analyze how these data points interact with each other.
3. Perform any necessary calculations or logical deductions.
4. Formulate the final answer based on these steps.

IMPORTANT:

- Your step-by-step reasoning **MUST NOT** exceed 5 sentences.
- Your final JSON **MUST** be included within the response.
- If you fail to comply, your response will be discarded as invalid.

<*question*>

Answer format (strict JSON):

- Boolean: {"answer": true}
- Category: {"answer": "category_name"}
- Number: {"answer": 42}
- List of Categories: {"answer": ["category_1", "category_2"]}
- List of Numbers: {"answer": [1, 2, 3]}

CRITICAL REQUIREMENTS:

- You **MUST** return your response in two parts:
  1. **Your step-by-step reasoning.**
  2. **The final validated response in STRICT JSON FORMAT.**
- The JSON response **must be the last thing in your answer** and formatted correctly.

Now, provide the answer.

Self-Refine Step 4: Prompt for validating a generated answer

You have the following **table**: <*table as string*>
Given question: <*question*>
Proposed answer: <*answer*>
Your previous reasoning was: <*response*>

Validation Task

1. Check whether the provided answer is logically correct based on the table data.
2. If it is incorrect or incomplete, identify the error.
3. Provide the corrected answer if needed, using **strict JSON format**.
4. If the original answer is correct, return **"VALID"** (as a JSON string).

CRITICAL REQUIREMENTS

- The response **MUST BE STRICTLY IN JSON FORMAT** with **NO explanations, no additional text, and no reasoning**.
- **DO NOT** include **"Reasoning"** or **"Validation"** steps in the output.
- **ONLY return one of the following:**
  - **"VALID"** (as a JSON string)
  - A corrected JSON answer in the exact same format as the proposed answer.

IMPORTANT:

- Your step-by-step reasoning **MUST NOT** exceed 5 sentences.
- Your final JSON **MUST** be included within the response.
- If you fail to comply, your response will be discarded as invalid.

Example Outputs

If the answer is correct: "VALID"
If the answer needs correction: {"answer": 42}

DO NOT RETURN outputs like:

- "The answer is correct. Here is my reasoning..."
- "After analysis, I found a mistake. The correct answer is: ..."

Now, validate the answer and provide the response in the correct format.

C Multi-hop Questions

Examples of questions from the Data Bench dataset that strictly require more than two columns to answer (for our dataset, three columns).

Multi-hop questions

Q: Which 6 Pokémon from the second generation have the highest attack stats?
Columns used: ['generation', 'attack', 'name']

Q: Is the profession with the highest Openness the same as the profession with the highest Conscientousness?
Columns used: ['Profession', 'Openness', 'Conscientousness']

Q: Does the profession with the lowest Emotional_Range also have the lowest level of Conversation?
Columns used: ['Profession', 'Emotional_Range', 'Conversation']

Q: What is the average Extraversion level for the profession with the highest number of records (n)?
Columns used: ['Profession', 'Extraversion', 'n']

Q: Has the author with the highest number of followers ever been verified?
Columns used: ['author_id<gx:category>', 'user_followers_count<gx:number>', 'user_verified<gx:boolean>']

Q: Is the author who has the most favourites also the one with the most retweets?
Columns used: ['author_id<gx:category>', 'user_favourites_count<gx:number>', 'retweets<gx:number>']

Q: Is the most mentioned user also the most retweeted mentioned user?
Columns used: ['author_id<gx:category>', 'mention_names<gx:list[category]>', 'retweets<gx:number>']

Q: Does the author with the most retweets also have the most replies?
Columns used: ['author_id<gx:category>', 'retweets<gx:number>', 'replies<gx:number>']

D Table Sampling: Optimal k

The Figure 2 shows the F1 metric for the three approaches. It can be seen that for our table of 20 columns, it will be optimal to search for the top 3 most suitable columns. The results are also presented in the Table 1.

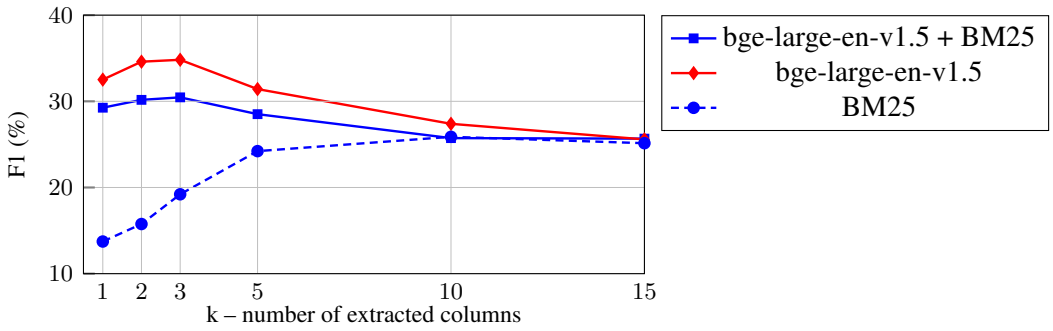

Figure 2: F1 scores for the "Table Sampling" column extraction method using different vector representations, shown for various values of the hyperparameter k (number of columns extracted).

E Analysis of Prompting Techniques

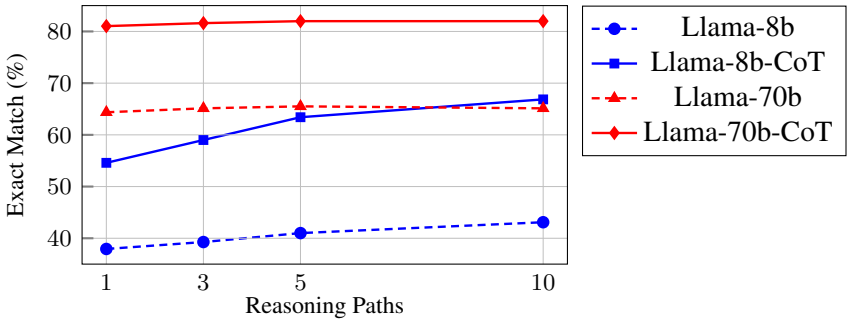

Figure 3: Self-Consistency results for DataBench lite test set. Exact Match is a metric with custom comparison function presented by competitions authors, see Section 4. Reasoning Paths means the number of times the model was called to generate an answer, see Section 5.2. CoT denotes Chain-of-Thought.