AlphaPro at SemEval-2025 Task 8: A Code Generation Approach for Question-Answering over Tabular Data

Anshuman Aryan **Laukik Wadhwa** **Kalki Eshwar D** **Aakarsh Sinha** **Durgesh Kumar**
School of Computer Science and Engineering (SCOPE)
Vellore Institute Of Technology, Vellore, Tamil Nadu, India - 632014
{anshuman.aryan24, laukikwadhwa, kalkieshward, aakarshsinha.in, durgesh.nlpai}@gmail.com

Abstract

This work outlines the AlphaPro team's solution to SemEval-2025 Task 8: Question Answering on Tabular Data. The task evaluates the question-answering capabilities of LLMs over tabular data. The proposed system introduces a three-stage pipeline: question transformation, intermediate Python code generation, and code execution. Utilizing the deepseek-v3 model produces overall 77.43% accuracy on the task dataset, demonstrating the feasibility of code generation for tabular question answering. A comparative analysis of deepseek-v3 with five current state-of-the-art LLMs tested has been presented. The strengths of the system are outlined and directions for further research are provided. The code and generated results for each tested model have been made available in a public code repository [1] to promote reproducibility and research in this area.

1 Introduction

Large Language Models (LLMs) have demontrated impressive abilities to understand, reason and interact with structured tabular data (Sui et al., 2024; Wu et al., 2025). By integrating Natural Language (NL) processing with advanced reasoning, LLMs offer a powerful and flexible way to extract valuable insights from tabular datasets (Liu et al., 2023).

Tabular data are prevalent in various fields, from financial reports to scientific findings and statistical analyses. SemEval-2025 Task 8 (Osés Grijalba et al., 2025) focuses on the importance and challenge of developing systems capable of answering questions (QA) over tabular data. The ability to query tables using natural language, instead of specialized languages like SQL, significantly reduces the barrier to accessing and interpreting data.

Earlier research on tabular datasets used a translation-based system to translate from natural language to SQL using semantic parsing techniques (Zhong et al., 2017; Li and Jagadish, 2014) and deep learning-based techniques (Xu et al., 2017). These methods lack schema awareness of the table and suffer from the ambiguity of natural languages. Recent advancements, such as RAT-SQL (Wang et al., 2021) and NL2SQL (Liu et al., 2024), address schema integration but still encounter limitations related to semantic context representation and query diversity (Liu et al., 2024; Zhang et al., 2024). Traditional query languages such as SQL are limited in this regard, as they primarily understand only the structural aspects of tables but struggle to capture their underlying semantics (Zhang et al., 2024).

To address the above limitations, this work proposes a schema-infused LLM prompting approach leveraging few-shot learning and intermediate Python code generation. Unlike traditional SQL-based methods, this approach capitalizes on schema-aware question paraphrasing and dynamically generated Python code, substantially improving semantic expressiveness and flexibility in handling NL queries. Figure 1 illustrates the three-stage pipeline of the proposed system to answer questions posed in natural language, employing schema-aware question paraphrasing and generating Python code as intermediaries. In the first stage, the answer type is predicted and the user's question is transformed into a schema-aware format, optimizing it for the subsequent stage. In the second stage, the model generates Python code utilizing the pandas framework [2]. The final stage involves extracting and executing this generated Python function, subsequently retrieving the results. Python was selected due to its powerful libraries for manipulating tabular datasets, such as pandas, and its minimalistic and dynamically typed nature, which facilitates effective analysis of the generated

[1] https://github.com/AnshumanAryan24/AlphaPro-SemEval2025-Task8

[2] pandas

code.

The key contributions of this work are summarized as follows:

- A novel three-stage LLM prompting framework leveraging few-shot learning is proposed for question answering over tabular data. This framework integrates schema-aware paraphrasing and intermediate Python code generation, explicitly utilizing table schema information such as column names and data types to enhance the reasoning capabilities of the system.

- A systematic comparison and detailed performance analysis are conducted across six state-of-the-art open-source LLMs, ranging from 7B to 671B parameters, evaluating their effectiveness across questions of varying complexity.

- A comprehensive error analysis is provided, highlighting significant challenges in code-based tabular reasoning, particularly in the processing of categorical data and special characters.

- An end-to-end implementation with outputs generated by all evaluated models has been publicly released [3], promoting reproducibility and supporting future research endeavours in tabular reasoning.

The proposed system using the DeepSeek-v3 model (DeepSeek-AI, 2024), evaluated on the DataBench dataset (Grijalba et al., 2024), outperforms five contemporary state-of-the-art open-source LLMs, achieving an average accuracy of 77.43%.

2 Related Work

Recent advances in question answering (QA) over structured data have spurred research into neural architectures, semantic parsing, and program generation for tabular reasoning.

2.1 QA over Structured Data

Early work on structured data QA often employed semantic parsers to translate natural language questions into SQL or other formal query languages (Zhong et al., 2017). These approaches typically relied on handcrafted grammar rules and domain-specific features.

With the advent of pretrained language models, neural techniques such as TAPAS (Herzig et al., 2020) and TaBERT (Yin et al., 2020) introduced joint encoders for table-question pairs, enabling end-to-end reasoning without intermediate formal representations. These models directly perform operations like cell selection and aggregation. More recently, RHGN (Yang et al., 2023) proposed a two-stage strategy involving row selection followed by row-level comprehension, further improving QA accuracy over tables.

Despite these advances, many existing models require full-table encoding and extensive task-specific finetuning, which can be computationally expensive and less adaptable across domains.

2.2 Executable Code Generation for QA

A complementary direction involves generating executable programs as intermediate representations for QA. This paradigm enables complex reasoning using the expressiveness of programming languages. Chen et al. (Chen et al., 2021) showed that pretrained LLMs can generate Python code to answer multi-step questions, providing interpretability and improved control.

Rajkumar et al. (Rajkumar et al., 2022) extended this idea by proposing a framework to query tables using Python and the Pandas library (pandas development team, 2020). Their results demonstrated that Python-based generation can outperform SQL-based parsing for complex queries, especially those involving arithmetic, filtering, or nested logic.

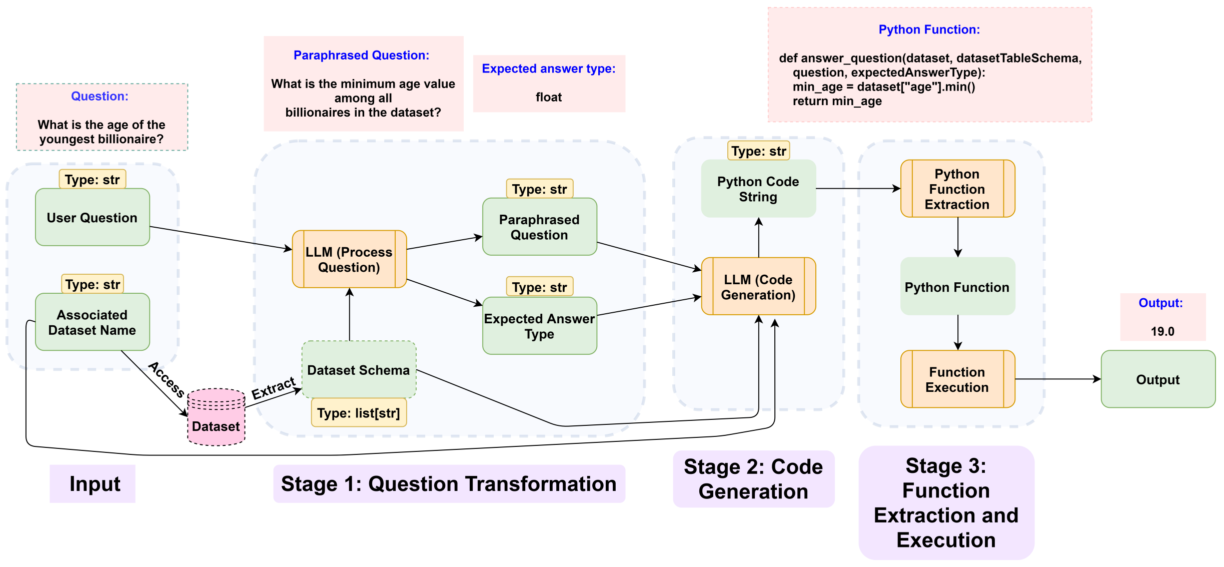
Building on these insights, our work adopts a prompt-based approach that leverages LLMs to generate Python code for tabular QA without requiring model finetuning. We focus specifically on the SemEval-2025 Task 8 (Osés Grijalba et al., 2025), introducing a schema-aware prompting strategy that bridges structured inputs and executable reasoning via interpretable code generation.

3 System Overview

This work proposes a three-stage framework powered by Large Language Models (LLMs) to generate executable Python code to answer questions on tabular data sets. The overall architecture, depicted in Figure 1, comprises the following sequential components:

(a) **Question Transformation**: The input ques-

[3] https://github.com/AnshumanAryan24/AlphaPro-SemEval2025-Task8


Question:

What is the age of the youngest billionaire?

Paraphrased Question:

What is the minimum age value among all billionaires in the dataset?

Expected answer type:

float

Python Function:

```
def answer_question(dataset, datasetTableSchema,
    question, expectedAnswerType):
    min_age = dataset["age"].min()
    return min_age
```

Output:

19.0

Figure 1: Architecture diagram for proposed system

tion is first transformed into a schema-aware representation. This step employs prompt engineering techniques that incorporate the structure of the dataset, including column names and data types. The LLM is guided to infer the expected answer type (e.g., scalar, list, boolean) and reformulate the question to enhance compatibility with code generation in subsequent stages.

(b) **Code Generation**: The reformulated question is passed to the LLM to generate Python code, typically using the Pandas library. The generated code is crafted to query the input table effectively and extract the relevant answer. This stage bridges natural language understanding with executable logic.

(c) **Execution and Result Formatting**: The generated code is executed on the tabular dataset, and the resulting output is post-processed to conform to the desired answer format. This may include standardizing numeric precision, formatting lists, or converting values into natural language responses, depending on task-specific requirements.

3.1 Question Transformation Prompt Design

This section elucidates the methodology through which the prompt template shown in Figure 3 of Appendix B was crafted. The objective is to paraphrase the natural language question into a form infused with keywords from the table schema, and which is in a format more suitable for code generation. The prompt consists of:

- **Precise instructions regarding the goal** The prompt defines two key tasks for the model: first, to predict the expected answer type of the question; and second, to paraphrase the question into a form that is more suitable for code generation while preserving its original semantic meaning.

- **Explicit marking of expected answer types** The set of possible data types for the answer (boolean, category, number, list[category], list[number]) is designated in the prompt.

- **Few-shot prompting** Two input-output examples are provided that demonstrate the dual task of predicting answer types and paraphrasing questions. This enables the LLM to infer the task structure and generalize to unseen queries within the same format.

3.2 Code Generation using Prompt Engineering

In the code generation step, the system approach employed another customized prompt that takes the paraphrased question and produces Python code that can be executed. The prompt consists of:

- **Information about the prompt structure** The prompt first informs that the model will be provided with four pieces of information - dataset name, schema, question, and expected answer type.

- **Information about main objective** The following requirements to generate the necessary Python code are mentioned in the prompt:

(a) The name of the dataset and schema description (column names along with data types)

(b) The paraphrased question from the prior stage of question transformation

(c) The anticipated answer type (also from the prior stage)

- **Guidelines on output formatting** To obtain a more precise output, the expected answer type obtained from the output of stage 1 as shown in Figure 1 is mentioned.

- **Expected output function definition** The model is explicitly instructed to generate only the function definition, assuming the presence of the required code base and pandas library.

- Owing to some of the larger models, such as Llama and DeepSeek, there was a need for specific instructions to prevent additional Markdown and explanatory content in the output.

Following this, manually crafted few-shot examples have also been provided. An illustrative example of the prompt designs has been presented in Appendix B.

3.3 Execution and Function Extraction

In the last stage, the system runs the code produced with the table as input, providing all global and local scope variables to the execution. We used Python's built-in interpreter function `exec()`. The execution environment consists of essential libraries, such as the pandas library, for handling data, and the result is presented according to the requirements of the task. The result, or error, produced by running this code is then formatted suitably. This is expected to allow:

- Ensuring proper answer data type (boolean, category, number, or lists)

- Correct list formatting with proper brackets and separators

- Handling exceptions and errors in system output

4 Experimental Setup

This paper evaluates proposed framework using the dataset from SemEval-2025 Task 8 (Osés Grijalba et al., 2025), made available through the DataBench platform (Grijalba et al., 2024). The dataset includes both training and development splits, and spans diverse domains and table schemas. This diversity enables robust testing of the system's generalization ability across different question types and data formats.

LLMs Used: This paper evaluates six LLMs with sizes ranging from 7B to 671B parameters, which includes general LLMs and opensource models : gemma-3-12b-it (Team, 2025), llama-3.3-70b-instruct-turbo, qwen2.5-7b-instruct-1m (Yang et al., 2025), qwen2.5-coder-14b-q5, c4ai-command-r-plus-08-2024 (Cohere Labs, 2024), deepseek-v3 (DeepSeek-AI, 2024).

System Components: The experimental pipeline incorporates the following core components:

- **Prompt Engineering:** Carefully crafted prompt templates with few-shot examples are used for both question paraphrasing and code generation, ensuring schema-awareness and context preservation.

- **Answer Type Prediction:** The system identifies the expected answer type—boolean, category, number, list[category], or list[number]—to guide the formatting and structure of the generated output (Codabench, 2025).

- **Code Execution:** Python code generated by the LLM is executed using a controlled environment powered by `exec()`. Execution is sandboxed with scoped global and local variables, and includes exception handling mechanisms to ensure safe and consistent evaluation.

Inference Configuration: All model inferences were performed through the Together AI API, with models such as `gemma`, `deepseek`, and others accessed via its hosted endpoints. For API-based calls, the `stream` parameter was set to `False` to ensure consistent output handling. Code generation tasks were configured with a maximum token limit of 1000 and a temperature range between 0.5 and 0.7 to balance creativity and determinism.

Evaluation Protocol: The proposed model is evaluated using the **DataBench** *eval* function. Accuracy is measured as the ratio of correctly answered questions to the total number of questions in the development set. In addition, output formatting is evaluated to ensure alignment with the expected data type and structure. To assess robustness, this work analyse model performance across varying levels of question complexity.

5 Results and Observations

The comprehensive analysis conducted demonstrates the system's feasibility and robustness, as detailed below.

5.1 Dataset Overview

Table 1 summarizes the characteristics of the DataBench dataset (Grijalba et al., 2024), comprising 65 tables across domains such as Business, Health, Social, Sports, and Travel. Additionally, a condensed version, **DataBench Lite**, includes all tables with only the first 20 rows, facilitating rapid prototyping. The dataset's questions are categorized into five answer types: `boolean`, `category`, `number`, `list[category]`, and `list[number]`.

Domain	Datasets	Rows	Columns
Business	26	1,156,538	534
Health	7	98,032	123
Social	16	1,189,476	508
Sports	6	398,778	177
Travel	10	427,151	273
Total	65	3,269,975	1615

Table 1: Domains of Tables in Dataset (Grijalba et al., 2024)

5.2 Question Complexity Analysis

To assess the complexity of the question, we used spaCy[4], a Python NLP library, analysing syntactic and lexical features. This analysis enabled categorization of questions into complexity levels based on factors such as sentence length, syntactic depth, and lexical diversity. Figure 2 illustrates the distribution of questions across complexity levels and the corresponding performance of the `deepseek-v3` model, as contrasted to others. The complexity-wise distribution of the questions and the performance of deepseek-v3 for each are presented in Table 2.

5.3 Model Performance Across Complexity Levels

Figure 2 presents the accuracy of various models across different complexity levels of the question. All models exhibited a decline in performance with increasing complexity. In particular, `qwen2.5-7b-instruct-1m` consistently performed beyond complexity level 0. In contrast, `deepseek-v3` achieved the highest accuracy at all levels of complexity. This discussion is expanded in the Appendix A.

[4]https://spacy.io/

	Question		Accuracy
S.No	Complexity	Count	Accuracy (%)
1	0	2	100.00
2	1	194	83.51
3	2	292	81.85
4	3	275	73.45
5	4	174	70.11
6	5	51	74.51

Table 2: Systematic Performance Analysis of question complexity vs accuracy

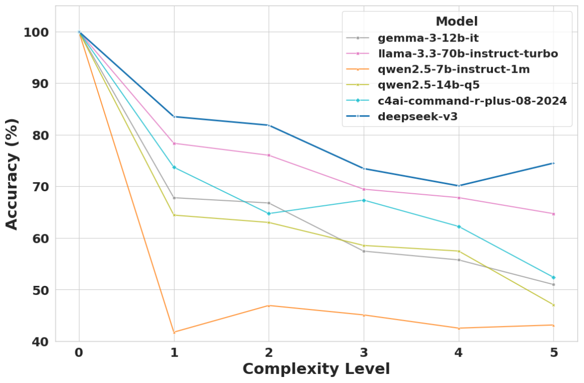


Figure 2: Complexity Wise Model Performance

5.4 Evaluation Using DataBench Eval Function

We utilized the `databench_eval` package (Grijalba et al., 2025) to compare generated answers with ground truth. This evaluation metric accommodates minor formatting discrepancies, such as item order in lists and numerical representations (e.g., integer vs. float), focusing on semantic correctness.

5.5 Error Analysis

A careful inspection of the generated code errors revealed several recurring issues. Table 3 summarizes the total number of errors produced by each model.

- **Special Character Handling:** Smaller models struggled to process inputs containing emojis or special characters (e.g., the euro symbol

'€'), often resulting in incomplete or failed code execution.

- **Data Type Mismatches for Categorical Columns:** The most frequent error across all models involved misinterpretation of categorical columns as string types. For example, in response to the question *"List the 2 most common host verification methods"*, the model incorrectly treated the column `host_verifications` (of type `list[category]`) as a string, leading to incorrect logic in the generated code.

- **Schema Ignorance and Improper Type Assumptions:** Despite having access to the column schema, models sometimes assumed incorrect data types and applied incompatible functions. For instance, for the question *"Are there any players who joined their current club before they were 18 years old?"*, the model misinterpreted columns `Joined<gx:date>` and `Age<gx:number>` as strings, resulting in erroneous function calls such as `str()` on numeric values.

- **Use of Deprecated Functions:** Some models generated outdated code involving deprecated functions or parameters, resulting in warnings or compatibility issues during execution. For example, when answering *"What are the bottom five number of replies?"*, the generated code included the deprecated Pandas parameter `observed=False`, triggering a `FutureWarning` due to changes in the default behavior of the library.

- **Correlation with Model Size:** We observed a negative correlation between model size and the number of errors. The smallest model, `qwen2.5-7b-instruct-1m`, generated 371 errors, whereas the largest model, `deepseek-v3` (671B parameters), produced only 44. This trend suggests that larger models better generalize schema understanding and generate more reliable code.

Model Name	Errors Generated
gemma-3-12b-it	183
llama-3.3-70b-instruct-turbo	78
qwen2.5-7b-instruct-1m	371
qwen2.5-coder-14b-q5	97
c4ai-command-r-plus-08-2024	127
deepsseek-v3	44

Table 3: Count of erroneous codes generated by different models

6 Conclusion and Future Work

This paper presents a three-stage LLM-based framework for SemEval-2025 Task 8: Question Answering over Tabular Data. The proposed three-stage framework comprises question transformation and answer type prediction, followed by code generation and code execution utilizing an LLM-based code prompting and a few-shot learning based approach to tabular reasoning. The system achieves an overall accuracy of 77.43% using the deepseek-v3 model.In addition, this study also presents a comparative study of different models and analyses the structure of the task data set. The prompt design is analysed and an illustrative example of the system's working is presented. The experimental results show the potential of using prompt engineering and code generation as an intermediate step in tabular question answering.

The scope of the system can be expanded further, through investigation on the following topics:

- Enhancing the Question Transformation prompt by testing with additional context.
- More effectively incorporating table structure and metadata into the question interpretation.
- Improving code generation with an error recovery step for automated handling of encountered errors.
- Expanding the scope of the Code Generation stage to generate code for different related tasks, and using different libraries available for Python.

Acknowledgements

We would like to express our gratitude to Jorge Osés Grijalba for designing this task as challenging and significant. We also appreciate the helpful comments from anonymous reviewers, peers, and the open-source community that helped in the development of our method.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg

Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Codabench. 2025. Codabench — codabench.org. `https://www.codabench.org/competitions/3360/#/pages-tab`. [Accessed 28-02-2025].

Cohere Labs. 2024. c4ai-command-r-plus-08-2024.

DeepSeek-AI. 2024. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Jorge Osés Grijalba, L Alfonso Urena Lopez, 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.

Jorge Osés Grijalba, Nikolas Evkarpidi, and Aditya Bangar. 2025. GitHub — jorses/databench_eval. `https://github.com/jorses/databench_eval`. [Accessed 28-02-2025].

Jonathan Herzig, Paweł Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. Tapas: Weakly supervised table parsing via pre-training. *arXiv preprint arXiv:2004.02349*.

Fei Li and Hosagrahar V Jagadish. 2014. Nalir: an interactive natural language interface for querying relational databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 709–712, New York, NY, USA. Association for Computing Machinery.

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

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuyu Luo, Yuxin Zhang, Ju Fan, Guoliang Li, and Nan Tang. 2024. A survey of nl2sql with large language models: Where are we, and where are we going? *ArXiv*, abs/2408.05109.

Jorge Osés Grijalba, L. 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)*, pages 1015–1022, Vienna, Austria. Association for Computational Linguistics.

The pandas development team. 2020. pandas-dev/pandas: Pandas.

Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. 2022. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498*.

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. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*, pages 645–654.

Gemma Team. 2025. Gemma 3. *Unknown*.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2021. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *Preprint*, arXiv:1911.04942.

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xeron Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, and 1 others. 2025. Tablebench: A comprehensive and complex benchmark for table question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25497–25506.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *Preprint*, arXiv:1711.04436.

An Yang, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoyan Huang, Jiandong Jiang, Jianhong Tu, Jianwei Zhang, Jingren Zhou, Junyang Lin, Kai Dang, Kexin Yang, Le Yu, Mei Li, Minmin Sun, Qin Zhu, Rui Men, Tao He, and 9 others. 2025. Qwen2.5-1m technical report. *arXiv preprint arXiv:2501.15383*.

Peng Yang, Wenjun Li, Guangzhen Zhao, and Xianyu Zha. 2023. Row-based hierarchical graph network for multi-hop question answering over textual and tabular data. *The Journal of Supercomputing*, 79(9):9795–9818.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. Tabert: Pretraining for joint understanding of textual and tabular data. *arXiv preprint arXiv:2005.08314*.

Xiang Zhang, Khatoon Khedri, and Reza Rawassizadeh. 2024. Can llms substitute sql? comparing resource utilization of querying llms versus traditional relational databases. *Preprint*, arXiv:2404.08727.

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

A Comparative Analysis of Different LLMs

The accuracy results obtained for each model have been mentioned in the Figure 2. The `qwen2.5-7b-instruct-1m` model showed the worst performance. The models `gemma-3-12b-it` and `qwen2.5-coder-14b-q5` showed similar decline in performance. Further discussion based on

Model	Accuracy (%)	Adjusted Accuracy (%)	Decline Rate (%)	Std. Deviation
gemma-3-12b-it	61.76	59.75	3.36	7.29
llama-3.3-70b-instruct-turbo	72.67	71.27	2.73	5.72
qwen2.5-7b-instruct-1m	44.53	43.89	-0.28	2.1
qwen2.5-coder-14b-q5	60.32	58.1	3.47	6.83
c4ai-command-r-plus-08-2024	66.09	64.47	3.37	6.69
deepseek-v3	77.43	76.69	1.8	5.74

Table 4: Comparison of model performance metrics

adjusted accuracy scores for different complexity levels is presented below.

The complexity type 0 had only 2 questions, compared to the vastly more number of questions in other types, and all models got correct answers for these. This resulted in skewed data for accuracy scores. Hence, while finding the adjusted accuracy and standard deviation, these have not been considered, as discussed below.

To account for the difference in the number of questions in each category, we follow the following macro-average principle for calculating accuracies:

$$A'_{model} = \frac{1}{N} \sum A_{model,i}$$

where $A_{model,i}$ is the accuracy for complexity level i, A'_{model} is the model's new complexity-averaged accuracy score, and $N = 5$ is the number of levels averaged over.

Along with these scores, the performance decline rate and the standard deviation based on the complexity-wise performance in Figure 2 are also computed in Table 4.

Decline rate is computed as follows:

$$D_{model} = \frac{A_{model,5} - A_{model,1}}{4}$$

where D_{model} is model's decline rate, and $A_{model,5}$ and $A_{model,1}$ are the accuracies at complexity levels 5 and 1 respectively. From Table 4, the deepseek-v3 and llama-3.3-70b-instruct-turbo models show a smooth, gradual performance decline with increasing question complexity, reflecting stability. The qwen2.5-7b-instruct-1m shows a negative decline rate, implying that performance improved slightly, but overall performance is the worst among all models.

B Prompt Design

Two prompts utilized in the implementation are illustrated here - query transformation prompt and code generation prompt. Both the prompts have an initial template which contains specific instructions along with few-shot examples. These prompt initials are then completed by adding specific information for the given query. For illustration the question given in Table 5 has been taken from the Databench dataset (Grijalba et al., 2024). The prompts are as follows:

- **Query Transformation Prompt** The prompt given in Figure 3 shows the complete prompt for the Question Transformation stage. The lines 1 to 23 are the prompt initials, and the line 24 to 26 are filled for each specific question.

- **Code Generation Prompt** The prompt given in Figure 5 shows the completed prompt for the Code Generation stage. The lines 1 to 36 are the prompt initials and 37 onward are filled for the specific transformed question.

The transformed question from the output of the first stage given in Table 5 show that, based on the few-shot examples, the model was able to add the specification of 'minimum' age, instead of asking about the 'youngest' billionaire. The generated code shown in Figure 4 first selects the minimum age using the `.min()` method of the pandas library, and returns the result.

Question	Paraphrased Question	Expected Answer Type
What is the age of the youngest billionaire?	What is the minimum age value among all billionaires in the dataset?	float

Table 5: Illustrative Question and Output of Question Transformation Stage

```
You will be provided with two pieces of information. The first being a question and the second being the column names along with data types of a dataset. Your objective is twofold, the first to predict the datatype of the answer and second to paraphrase the question aptly such that the next person could generate the python code to required to answer the question while keeping the  answer type the same as the given question. You are provided a two examples below.
Remember to not change what the original question is actually asking.

Notes:
Do not use markdown
Do not leave additional line spacing

Few Shot Examples:
Question: Is the person with the highest net worth self-made?
Dataset Name: 001_Forbes
Dataset Table Schema: selfMade (bool), finalWorth (int64), city (string), title (string), gender ( string), age (float64), rank (int64), philanthropyScore (float64), category (string), source ( string), country (string)
Answer Type: bool
Paraphrased Question: Does the billionaire with the maximum final worth have self made attribute set to True?

Question: Did any children below the age of 18 survive?
Dataset Name: 002_Titanic
Dataset Table Schema: Age (float64), Siblings_Spouses Aboard (int64), Sex (string), Name (string), Pclass (int64), Fare (float64), Survived (bool)
Answer Type: bool
Paraphrased Question: Were there any survivors aged under 18?

The answers types are only of type: [bool, float64, int64, string, list of (type)]

Instruction for you to perform:
Question: What is the age of the youngest billionaire?
  Dataset: 001_Forbes
  Dataset Table Schema: 'rank (uint16)', 'personName (category)', 'age (float64)', 'finalWorth ( uint32)', 'category (category)', 'source (category)', 'country (category)', 'state (category) ', 'city (category)', 'organization (category)', 'selfMade (bool)', 'gender (category)', ' birthDate (datetime64[us, UTC])', 'title (category)', 'philanthropyScore (float64)', 'bio ( string)', 'about (string)'
```

Figure 3: Query Transformation Prompt Example

```python
def answer_question(dataset, datasetTableSchema, question, expectedAnswerType):
    min_age = dataset["age"].min()
    return min_age
```

Figure 4: Generated code for given question from the dataset

```
You will be provided four pieces of information all of which are provided in the means of strings.
1. Dataset name:
2. Dataset Table Schema:
3. Question:
4. Expected Answer Type:

Your objective is to create a python code to answer the question given the dataset schema. Here is the function you will be needing to complete:
def answer_question(db:, datasetTableSchema, question, expectedAnswerType):
        answer = (Here you generate the code which is needed to find the answer)
        return answer

Assume that the pandas library has been imported as pd.
Your answer should only contain the function definition. Assume that the dataset schema (containing column names and their datatypes in paranthesis) given is correct. The generated code should be correct. Do not attempt to change the dataset.
Your final answer data type should be one of the following categories:
1. Boolean: One of True or False.
2. Category: A string. For example - CEO, hello, drugstores.
3. Number: A numerical value. For example - 20, 23.3223, 414901.0.
4. list[category]: A list of strings. For example - ['India', 'Japan', 'China'], ['Ram', 'Shyam', 'Mohan']. Here, each entry should be enclosed within single quotes.
5. list[number]: A list of numbers. For example - [20.0, 30.4, 42.1], [171000, 129000, 111000, 107000, 106000, 91400].
When the question requests more than value, the expected answer type might be a list of strings or numbers. Ensure that lists are enclosed within square brackets.

Few Shot Examples:
Example 1:
1. Dataset name: 001_Forbes
2. Dataset Table Schema: selfMade (bool), finalWorth (int64), city (string), title (string), gender (string), age (float64), rank (int64), philanthropyScore (float64), category (string), source (string), country (string)
3. Question: Does the individual with the highest final worth value have the selfMade attribute set to True?
4. Expected Answer Type: bool

Answer:
def answer_question(dataset, datasetTableSchema, question, expectedAnswerType):
        max_worth_individual = dataset.loc[dataset["finalWorth"] == dataset["finalWorth"].max()]
        is_self_made = max_worth_individual["selfMade"].bool()

        return is_self_made

Now, complete the following:
    1. Dataset name: 001_Forbes
    2. Dataset Table Schema: 'rank (uint16)', 'personName (category)', 'age (float64)', 'finalWorth (uint32)', 'category (category)', 'source (category)', 'country (category)', 'state (category)', 'city (category)', 'organization (category)', 'selfMade (bool)', 'gender (category)', 'birthDate (datetime64[us', 'UTC])', 'title (category)', 'philanthropyScore (float64)', 'bio (string)', 'about (string)'
    3. Question: What is the minimum age value among all billionaires in the dataset?
    4. Expected Answer Type: float
```

Figure 5: Code Generation Prompt Example