L-TQA: A Modular Multi-Agent Architecture for Context-Constrained Tabular Question Answering

Helen B. Alves [1], Diego D. Fernandes[1], Gustavo A. G. S. Dias[1], João A. M. B. Cavalcanti[1], Antônio Í. M. Pinto[1], Lara S. Silva, Rafael S. Santos[1], Thalyson G. N. Silva [1,3], Cleilton L. Rocha[2], Leonardo F. da Costa[1], Gustavo A. L. Campos[1], Ana Luiza B. P. Barros[1]

[1]State University of Ceara (UECE)
Fortaleza, Ceará, Brazil

[2]Atlantic Institute
Fortaleza, Ceará, Brazil

[3]Federal Institute of Education, Science, and Technology of Ceará (IFCE)
Fortaleza, Ceará, Brazil

{helen.alves, diego.dias, gustavo.alexandre, joao.cavalcanti, antonio.icaro, lara.silvestre, raf.silva, thalyson.nepomuceno}@aluno.uece.br, cleilton_rocha@atlantico.com.br, {leon.costa, gustavo.campos, analuiza.barros}@uece.br

Abstract. *Question Answering over Tabular Data requires accurate numerical reasoning and robust schema interpretation, both of which remain challenging for restricted-parameter language models. This paper presents Laura Tabular Question Answer (L-TQA), a multi-agent architecture designed to mitigate context limitations by decoupling semantic interpretation, schema filtering, and executable code generation. The proposed system improves query synthesis through targeted contextual constraints, combining a response-type classifier to enforce output formats with a dynamic column selector to reduce schema noise. Evaluated on the SemEval-2025 Task 8 DataBench benchmark, L-TQA establishes a new state-of-the-art for models with up to 9 billion parameters, achieving 77.01% accuracy on the test split. In addition, the proposed modular design generalizes effectively to larger proprietary models, highlighting its structural robustness and practical value for privacy-preserving, data-driven applications.*

1. Introduction

Large Language Models (LLM) have substantially advanced natural language interfaces across a wide range of domains [Touvron et al. 2023]. Nevertheless, querying structured tabular data remains a challenging problem. Unlike plain-text question answering, Question Answering over Tabular Data (Table QA) requires models to interpret two-dimensional structures, perform deterministic numerical and symbolic reasoning, and bridge the semantic gap between natural language queries and table schemas [Chen 2023, Nan et al. 2022].

Although large-scale proprietary models have achieved strong performance on Table QA tasks, their practical adoption is often constrained by inference cost, latency, and data privacy requirements [Wang et al. 2025]. In this context, Small Language Models

(SLMs) with fewer than 9B parameters constitute an attractive alternative for local and privacy-preserving applications. However, recent evaluations, such as SemEval 2025 Task 8 on Tabular QA [Grijalba et al. 2025], reveal a substantial performance gap between such models and state-of-the-art proprietary systems. In particular, SLMs tend to struggle with the long contexts induced by large tables, are more vulnerable to cascading reasoning errors, and exhibit lower reliability when generating executable code for data manipulation [Grijalba et al. 2024].

To address these limitations, we propose L-TQA, a Multi-Agent System (MAS) for Table QA designed specifically for models with up to 9B parameters. Instead of relying on a single general-purpose orchestrator, L-TQA decomposes the question-answering process into a set of specialized sub-tasks handled by dedicated agents. This modular design alleviates the context and reasoning bottlenecks typically observed in individual SLMs. More specifically, the framework incorporates specialized components for schema understanding, column filtering, response-type prediction, and code generation. In addition, a self-correction loop grounded in execution feedback enables the system to iteratively revise generated queries, thereby reducing hallucinations, execution failures, and logical inconsistencies.

We evaluate L-TQA on the DataBench benchmark [Grijalba et al. 2024], a comprehensive testbed for tabular reasoning. Our experimental protocol focuses on the capabilities of small open-source models and compares the proposed architecture against competitive systems reported in SemEval 2025 Task 8, including the DataBench QA and Lite QA subtasks [Grijalba et al. 2025]. We further position our method against recent approaches that also rely on restricted-parameter models, assessing the extent to which a structured multi-agent design can improve the tabular reasoning performance of SLM-based pipelines. In summary, this work makes three main contributions. First, we introduce L-TQA, a novel multi-agent architecture for Table QA that mitigates the context-window and reasoning limitations of SLMs through the explicit decomposition of semantic interpretation, schema filtering, code generation, and execution. Second, we propose a contextual constraint strategy that combines expected-response classification and dynamic column selection to reduce schema noise and improve the precision of executable query synthesis. Third, experimental results show that L-TQA establishes a new state of the art among restricted-parameter pipelines ($\leq$9B), achieving 77.01% accuracy on SemEval-2025 Task 8 DataBench. This result surpasses the best-ranked submissions in the small-model setting and demonstrates that carefully structured multi-agent orchestration can substantially narrow the gap between SLMs and larger proprietary models.

2. Background

LLM agents are systems centered on a language model that can autonomously reason, plan, and execute tasks through iterative interaction with their environment. Unlike conventional prompt-response use of LLMs, agentic systems incorporate additional architectural components, such as memory, planning modules, and tool-use mechanisms, which enable multi-step reasoning and more complex decision-making processes [Luo et al. 2025, Wang et al. 2024]. In recent literature, these components, particularly memory, planning, and action execution, are commonly regarded as the core dimensions for characterizing LLM-agent architectures.

Contemporary taxonomies describe LLM agents along several architectural dimensions, including memory, action execution, planning depth, and tool integration. Memory mechanisms allow agents to maintain contextual continuity across reasoning steps, whereas planning modules support task decomposition and long-horizon reasoning. Tool layers extend the capabilities of language models by enabling interaction with external environments, APIs, and code execution systems, thereby increasing the reliability and practical scope of the agent.

A central distinction in recent research concerns the orchestration strategy adopted by multi-agent systems. Some approaches rely on decentralized or dynamically expanding topologies, in which agents collaborate through adaptive communication patterns and evolving interaction structures [Luo et al. 2025]. Other approaches adopt more structured paradigms based on role specialization and controlled coordination schemes [Wang et al. 2024]. Although dynamic collaboration can increase flexibility and broaden problem-solving capacity, it may also introduce higher latency, unbounded context growth, and additional coordination overhead. In contrast, more structured orchestration strategies tend to favor determinism, controllability, and execution stability, albeit sometimes at the expense of adaptivity.

These design trade-offs reflect broader tensions in LLM-agent research, including flexibility versus control, scalability versus stability, and adaptive coordination versus predefined task flow. Understanding these trade-offs is particularly important for systems built on restricted-parameter models, where architectural decisions play a central role in compensating for the limited reasoning capacity of the underlying language model.

2.1. Multi-agent collaboration systems

Recent advances in multi-agent LLM systems suggest that decomposing complex tasks into specialized subproblems and assigning them to cooperating agents can improve both accuracy and efficiency [Jin et al. 2025]. Hierarchical and dynamic frameworks [Zhang et al. 2025, Zhang et al. 2026] organize multiple LLM agents into predefined yet adaptable structures, allowing complex queries to be decomposed into smaller and more manageable sub-tasks. Although such designs increase flexibility and problem-solving capacity, they often introduce scalability and latency challenges due to orchestration overhead and the dynamic accumulation of agents.

In the context of Table QA, recent frameworks have demonstrated the effectiveness of multi-agent collaboration for handling questions over long tables [Jin et al. 2025] and multiple tables [Zhu et al. 2024]. While these dynamic orchestration strategies can improve reasoning performance, they may also increase system complexity and response time as task depth and coordination demands grow.

In contrast, the L-TQA framework adopts a fixed structural design based on predefined pipelines. Rather than dynamically organizing agents into hierarchical configurations, L-TQA employs a central orchestrator that routes requests through a sequential and deterministic flow of tools and agents. This fixed-pipeline strategy mitigates scalability and latency issues by preventing uncontrolled agent expansion while preserving the ability to address complex tabular reasoning tasks.

2.2. Multi-agent communication

Efficient communication among LLM agents is a critical factor in the performance of multi-agent systems. Prior work [Wang et al. 2024, Zhu et al. 2024] has emphasized the importance of structured and dynamic communication protocols for enabling coordinated task execution across multiple agents. However, such frameworks often pay limited attention to the practical constraints imposed by restricted context windows, particularly in settings where smaller language models are employed. As interaction histories grow under dynamic or hierarchical coordination, the accumulated context can negatively affect both performance and reliability.

The L-TQA framework addresses this limitation by adopting a structured communication protocol that is tightly aligned with its fixed, non-dynamic pipeline architecture. By constraining message exchange and enforcing predictable interaction patterns, L-TQA reduces context-window pressure and maintains stable performance even when operating with smaller models.

2.3. Code based reasoning

For tasks involving intensive numerical operations or complex data transformations, recent systems have increasingly adopted code-generation agents [Bangar et al. 2025, Jiang et al. 2026] capable of producing executable Python or SQL programs to answer user queries. These approaches have demonstrated the effectiveness of code-based reasoning for structured question answering. However, their performance remains highly sensitive to ambiguity in the user query, which can propagate into incorrect or under specified execution logic.

The TabularQAPipeline in L-TQA follows a controlled and iterative code-execution paradigm [Raghav et al. 2025], while introducing additional preprocessing stages designed to improve query grounding before code synthesis. In particular, the pipeline employs a Column Descriptor to semantically characterize table attributes and a Column Selector to isolate only the attributes that are relevant to the user query. By restricting the contextual scope prior to code generation, L-TQA improves execution precision, reduces unnecessary token consumption, and decreases the frequency of debugging cycles during symbolic execution.

Unlike iterative self-reflection frameworks that rely on repeated regeneration cycles, the L-TQA method emphasizes controlled task decomposition before code synthesis. In tabular QA scenarios, where schema structure is explicit and tasks are often well-bounded, this fixed strategy provides a practical balance between precision and orchestration efficiency.

3. Proposed Multi-Agent Architecture for Tabular Question-Answer

In this work, a pipeline is defined as an ordered sequence of agents and tools that coordinate LLM-based calls for specific functions, such as Python script generation and execution. To illustrate this concept, we present the proposed Tabular Question Answer Pipeline, whose goal is to answer natural-language questions about a dataset by generating and executing a Pandas-based query [Reback et al. 2020].

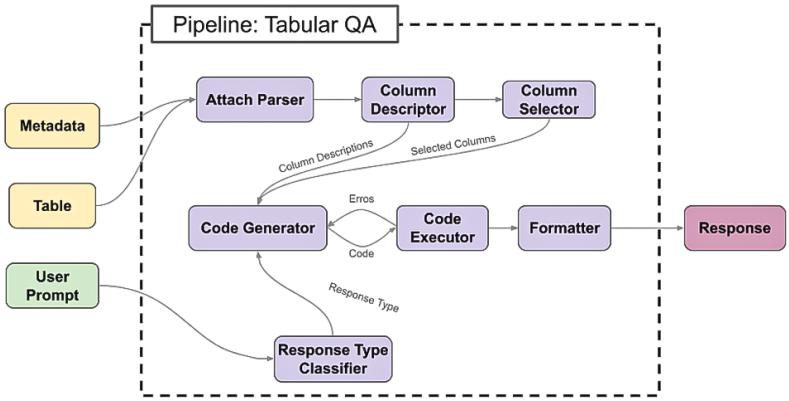

Figure 1. Proposed Architecture Workflow

The pipeline takes as input the user prompt, the tabular data, and, when available, associated metadata. These inputs are processed by a set of components, each producing intermediate outputs that are consumed by downstream agents in a sequential manner, ultimately leading to the final response (see Figure 1).

The execution order of the components is predefined and static. Throughout the pipeline, tools and agents exchange intermediate representations, where the output of one stage serves as the input to subsequent stages. In addition, the L-TQA incorporates a retry loop between the Code Generator agent and the Code Executor tool, allowing the system to iteratively refine generated queries whenever execution errors occur.

During preprocessing, the pipeline analyzes the tabular input by automatically generating column-level descriptions, identifying data types, and extracting a representative three-row sample. The user prompt is then used to infer the expected response schema.

In the context of Tabular Question Answering (TabularQA), the system categorizes the expected output into one of five predefined types: Boolean, Number, Category, Number Array, or Category Array. This schema prediction guides the subsequent query generation stage.

All intermediate outputs are then passed to the Code Generator agent, which produces the corresponding Pandas query. The generated query is executed by the Code Executor tool, and the resulting output is subsequently formatted into a natural-language response returned to the user.

3.1. Tools

3.1.1. Attachment Parser

The Attachment Parser is responsible for processing and standardizing user-provided tabular files, converting them into structured representations compatible with the downstream stages of the pipeline. As input, the component receives a list of pairs, each consisting of a tabular data file and an associated metadata file in JSON format, thereby establishing an explicit correspondence between the table content and its auxiliary semantic description.

The tabular file format is identified based on the file extension. The parser supports multiple formats, including CSV, Excel (XLSX, XLS, XLSM, and XLSB), TSV, TXT, Parquet, and tabular JSON. All tabular files are loaded using the Pandas library and converted into DataFrame objects, which serve as the internal standard representation for subsequent data manipulation.

Metadata files are loaded as Python dictionaries, preserving semantic descriptions and contextual attributes provided by the user. After validation of each pair of files, the tabular data and its metadata are encapsulated into TableInput objects, which constitute the input units for the analysis and code generation stages.

3.1.2. Code Executor

The Code Executor is responsible for securely executing the code generated by the Code Generator under strict runtime isolation. To prevent unintended memory sharing across components, the execution context is serialized into a temporary environment before code execution. The generated code is then wrapped with auxiliary routines for input handling, output collection, and stream capture.

After execution, the component deserializes the generated results, releases temporary resources, and returns a structured object containing either the execution output or the corresponding runtime error information.

3.2. Agents

3.2.1. Column Descriptor

The Column Descriptor performs the semantic interpretation of tabular data by transforming DataFrame columns into structured textual descriptions. Within the pipeline, it acts as an intermediate layer between raw tabular inputs and downstream reasoning components, producing a semantic data dictionary that enriches the table representation.

To generate these descriptions, the agent combines two complementary sources of information: (i) the technical schema, including column names and inferred data types, and (ii) empirical evidence derived from representative row samples, specifically the first three records of the table. By integrating structural and contextual signals, the component produces column profiles that are both semantically informative and operationally useful for subsequent reasoning stages.

3.2.2. Column Selector

The Column Selector identifies the minimal subset of columns required to answer a user query from the semantically enriched table representation produced by the previous stage. Its input combines the user's question with the semantic profiles of the columns, allowing the selection process to be guided by column semantics rather than solely by column names. The output is generated in a typed structured format, ensuring that the result corresponds to a valid set of selected columns.

By restricting the working table representation to the smallest sufficient subset of attributes, the Column Selector reduces the search space for query generation, making the overall process both more precise and more computationally efficient.

3.2.3. Code Generator

The Code Generator produces executable Python code, typically based on Pandas, to answer a user's natural-language request over structured tabular data. However, it does not execute the generated code. Instead, its role is restricted to synthesizing syntactically valid and schema-compliant code through a large language model (LLM).

The agent receives a fully structured input object composed of a natural language query, the table description and the selected columns, which are the respective Column Descriptor and Column Selector outputs, and an expected response type, which defines the required output format (e.g., "number", "category", "list of categories" or "boolean"). Finally, the output is an executable code, strictly conformed to a predefined schema via structured LLM generation.

3.2.4. Response Type Classifier

Prior to the column description stage, the pipeline predicts the expected format of the final answer using the Response Type Classifier. Given a user's natural-language question, this agent employs an LLM to classify the expected output into one of four predefined categories: number, category, list of categories, or boolean.

The model is prompted with explicit decision rules and constrained to return exactly one label through structured output generation, ensuring schema-compliant responses. By explicitly modeling the expected response type, the system reduces ambiguity and improves alignment between the generated code and the intended semantic form of the final response.

3.2.5. Formatter

The Formatter converts the structured outputs produced by the Code Executor into natural-language answers. To do so, it receives the original user question, the execution result, and the generated code, and integrates these elements into a structured prompt. This prompt is then submitted to a configurable language model, which may be deployed either locally or remotely, to generate the final response.

4. Experimental Setup

In this section, we describe the suite of tests applied to the pipeline to validate its effectiveness relative to other current solutions. To this end, we simulate the SemEval-2025 Task 8: Question Answering over Tabular Data competition, which guided the design and development of our architecture.

To ensure a valid evaluation, we used the official open-source evaluation script available in the public repository[1]. This is the same script employed to evaluate the participating teams in SemEval 2025 Task 8.

4.1. Dataset

We evaluated our approach using DataBench [Grijalba et al. 2024], the official benchmark for SemEval-2025 Task 8, alongside its context-constrained variant, DataBench Lite (capped at 20 rows per dataset). The core corpus encompasses real-world databases across five domains, featuring QA pairs classified into five expected answer types: boolean, number, category, list[number], and list[category].

For clarity and consistency, we adopt a standardized nomenclature for the data partitions and their respective lite counterparts. The development splits used for architecture validation are denoted as **Dev** and **Dev-lite**. The complete dataset for comprehensive assessment is referred to as **QA** and **QA-lite**. Finally, the split used for the final ranking is designated as **Test** and **Test-lite**.

4.2. Environment

The experiments were conducted on a system consisting of an AMD Ryzen 7 5700X 8-core processor, an RTX 5070 Ti GPU with 16 GB of VRAM, and 64 GB of DDR4 RAM. The system was implemented in Python with LangGraph, relying on open-source libraries such as Pandas and NumPy for dataset manipulation and mathematical computation. Finally, Ollama, running on Windows 11, was used to initialize and execute LLMs, reducing test time and improving efficiency.

5. Results

This section presents the empirical evaluation of the proposed pipeline. We first evaluated all candidate models on the **Dev** and **Dev-lite** splits to validate their baseline performance and select the most capable architectures.

As detailed in Table 1, `qwen2.5-coder:7b` and `rnj-1:8b` achieved the highest accuracy. The `qwen2.5-coder:7b` model reached 80.0% on the Dev set and 82.5% on the Dev-Lite set. These results indicate that models optimized for code generation exhibit an advantage in tasks requiring executable queries. Conversely, general-purpose models, such as `glm4:9b` and `llama3.1:8b`, yielded lower accuracy scores.

Table 1. Accuracy of candidate local SLMs on the Dev and Dev-lite splits.

Model	Dev	Dev-lite
`qwen2.5-coder:7b`	**80.0%**	**82.5%**
`glm4:9b`	57.5%	61.2%
`llama3.1:8b`	65.6%	62.5%
`rnj-1:8b`	78.4 %	80.3 %

To identify performance patterns, we analyzed the models according to the expected answer types. Figure 2 illustrates this distribution. The results show how each

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

model handles different structural requirements and indicate the consistent performance of the top two models across the various categories.

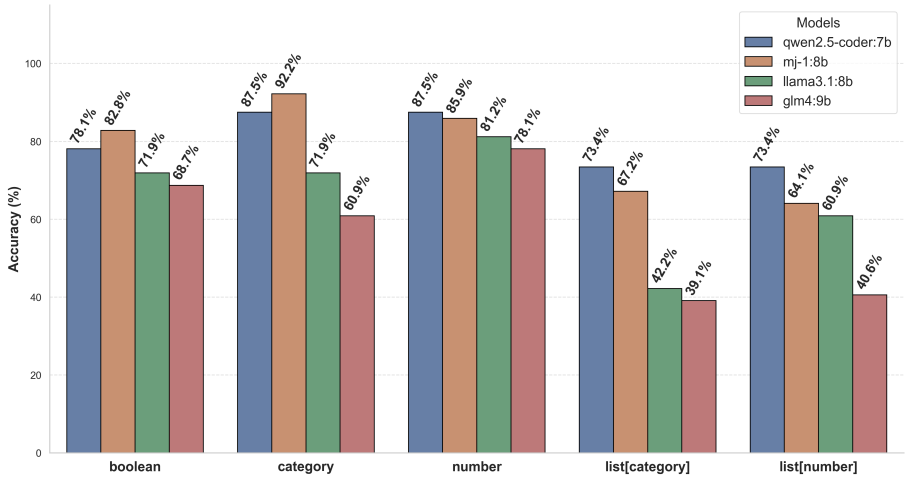

Figure 2. Accuracy distribution of candidate models by expected answer type on the Dev split.

Based on the initial validation, we selected the two top-performing models (`qwen2.5-coder:7b` and `rnj-1:8b`) for a comprehensive evaluation on the **QA** split. Table 2 presents these findings. Under this expanded evaluation, `rnj-1:8b` outperformed `qwen2.5-coder:7b`, achieving 75.14% on the **QA** split and 78.58% on the lite variant.

Table 2. Accuracy of the top two selected local LLMs evaluated on the QA split.

Model	QA	QA-lite
`qwen2.5-coder:7b`	75.13%	77.54%
`rnj-1:8b`	**75.14 %**	**78.58 %**

To benchmark our system against the state-of-the-art under restricted-parameter configurations, we compared our results with ScottyPoseidon, the top-performing submission using models with up to 9B parameters. This comparison was conducted exclusively on the **Dev** and **Dev-lite** splits, which were the specific partitions available to competitors for validation during the active competition phase. As shown in Table 3, our proposed pipeline outperformed their approach on both splits.

Table 3. Performance comparison between the proposed pipeline and the top-ranked restricted-parameter pipeline on the Dev partitions.

Model	Dev	Dev-lite
L-TQA (`qwen2.5-coder:7b`)	**80.00%**	**82.50%**
L-TQA (`rnj-1:8b`)	78.44%	80.31%
ScottyPoseidon	73.18%	73.75%

Following the validation phase, we evaluated our pipeline on the official **Test** and **Test-lite** splits to replicate the exact ranking conditions of the SemEval-2025 Task

8 competition. As detailed in Table 4, the proposed system established new state-of-the-art results in the restricted-parameter category ($\leq$ 9B). The L-TQA configuration utilizing `rnj-1:8b` would have ranked first overall with 77.01% accuracy, while the `qwen2.5-coder:7b` variant would have placed third, outperforming the most of the submitted approaches.

Table 4. Top ranking SLM pipelines compared to the proposed L-TQA on the official Test splits.

Ranking	Team	Test	Test-lite
1	**L-TQA(`rnj-1:8b`)**	**77.01 %**	**77.39%**
2	ScottyPoseidon	76.63%	74.71%
3	**L-TQA(`qwen2.5-coder:7b`)**	**73.95%**	**74.52%**
4	Dataground	68.97%	69.35%
5	NexGenius	65.64%	66.22%
6	Tree-Search	64.56%	64.94%
7	Basharat Ali	43.10%	43.87%

To further assess the upper-bound performance of the proposed pipeline, we evaluated proprietary large language models accessed through external APIs. These models typically benefit from significantly larger training corpora, more parameters, and extensive instruction tuning.

To this end, we employed Gemini 2.5 Flash, which offers a strong balance between cost and performance. Notably, we preserved the same L-TQA structure and prompting strategy previously used with smaller models. Under this setup, the approach achieved 90.61% accuracy on the Databench Full Test split, ranking second, as shown in Table 5.

Table 5. Top 5 ranking pipelines compared to L-TQA

Ranking	Team	Test
1	TeleAI	95.02%
2	**L-TQA(gemini-2.5-flash)**	90.61%
3	AILS-NTUA	89.85%
4	SRPOL AIS	89.66%
5	sonrobok4	89.46%
6	langtechdata61	88.12%

Considering that this benchmark corresponds to the primary task of the SemEval competition, our method would have ranked second overall, surpassed only by the TeleAI team. Importantly, all pipeline design decisions, including prompt tuning and validation, were carried out exclusively with smaller models on the development split. Therefore, the strong performance observed with a larger model highlights the robustness and generalizability of the proposed L-TQA framework.

6. Conclusion and Future Works

The experimental evaluation showed that controlling the context provided to the code-generation component positively affects performance. By introducing targeted signals,

such as predicting expected output types through the `ResponseTypeClassifier` and filtering the schema via the `SelectedColumns`, the pipeline supplies lightweight yet highly effective structural cues that improve query synthesis accuracy. Furthermore, the results show that code-specialized models (e.g., `qwen2.5-coder:7b` and `rnj-1:8b`) are significantly better suited to the task than general-purpose language models under the same parameter constraints.

Despite these advances, the study also highlights important limitations. First, although the proposed pipeline demonstrates that small open-weight models ($\leq$ 9B parameters) can achieve highly competitive results, a performance gap remains relative to substantially larger proprietary models. This indicates that the proposed design can significantly enhance the effectiveness of compact models, but does not fully eliminate the advantages associated with scale. In addition, the analysis across expected answer types revealed that these models struggle to maintain high accuracy on more complex output structures, particularly list-type responses. Such formats yielded lower accuracy than simpler boolean or single-value numerical queries, suggesting a limitation in the ability of small models to generate accurate multi-value outputs.

Future work will focus on addressing these limitations. In particular, we plan to investigate strategies for improving the handling of complex output types, especially those involving list-based tabular operations. This includes exploring mechanisms to strengthen the models' ability to reason over multiple values, maintain structural consistency in generated outputs, and produce more reliable code for multi-item retrieval scenarios. We also intend to examine whether additional schema-aware signals or intermediate planning steps can further improve performance in these more challenging cases.

Overall, the proposed approach establishes a **new state-of-the-art** methodology for restricted-parameter models on the SemEval-2025 Task 8 benchmark. By effectively combining structured schema guidance with code-oriented models, L-TQA provides a robust, efficient, and privacy-preserving solution for local deployment environments. Moreover, the seamless transfer of the same architecture to larger proprietary models, yielding highly competitive upper-bound results, further highlights the robustness and generalizability of the proposed approach.

References

Bangar, A., Kumar, A., Mishra, S., and Modi, A. (2025). Exploration lab IITK at SemEval-2025 task 8: Multi-LLM agent QA over tabular data. In Rosenthal, S., Rosá, A., Ghosh, D., and Zampieri, M., editors, *Proceedings of the 19th International Workshop on Semantic Evaluation (SemEval-2025)*, pages 2165–2169, Vienna, Austria. Association for Computational Linguistics.

Chen, W. (2023). Large language models are few(1)-shot table reasoners. In Vlachos, A. and Augenstein, I., editors, *Findings of the Association for Computational Linguistics: EACL 2023*, pages 1120–1130, Dubrovnik, Croatia. Association for Computational Linguistics.

Grijalba, J. O., Ureñ-López, L. A., Martínez-Cámara, E., and Camacho-Collados, J. (2025). Semeval-2025 task 8: Question answering over tabular data. In *Proceedings of the 19th International Workshop on Semantic Evaluation (SemEval-2025)*, pages 2512–2522.

Grijalba, J. O., Ureña-López, L. A., Cámara, E. M., and Camacho-Collados, J. (2024). Question answering over tabular data with databench: A large-scale empirical evaluation of llms. In *Proceedings of LREC-COLING 2024*, Turin, Italy.

Jiang, Y., Wei, F., Bao, E., Li, Y., Ding, B., Yang, Y., and Xiao, X. (2026). Accurate table question answering with accessible llms.

Jin, R., Wang, X., Wang, D., Zheng, H., Qi, Y., Yang, S., and Zhang, M. (2025). TALON: A multi-agent framework for long-table exploration and question answering. In Christodoulopoulos, C., Chakraborty, T., Rose, C., and Peng, V., editors, *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 27397–27413, Suzhou, China. Association for Computational Linguistics.

Luo, J., Zhang, W., Yuan, Y., Zhao, Y., Yang, J., Gu, Y., Wu, B., Chen, B., Qiao, Z., Long, Q., Tu, R., Luo, X., Ju, W., Xiao, Z., Wang, Y., Xiao, M., Liu, C., Yuan, J., Zhang, S., Jin, Y., Zhang, F., Wu, X., Zhao, H., Tao, D., Yu, P. S., and Zhang, M. (2025). Large language model agent: A survey on methodology, applications and challenges.

Nan, L., Hsieh, C., Mao, Z., Lin, X. V., Verma, N., Zhang, R., Kryściński, W., Schoelkopf, H., Kong, R., Tang, X., Mutuma, M., Rosand, B., Trindade, I., Bandaru, R., Cunningham, J., Xiong, C., and Radev, D. (2022). FeTaQA: Free-form table question answering. *Transactions of the Association for Computational Linguistics*, 10:35–49.

Raghav, R., Vemali, A. P., Aswal, D., Ramesh, R., and Bhupal, A. (2025). Scottyposeidon at semeval-2025 task 8: Llm-driven code generation for zero-shot question answering on tabular data. In *Proceedings of the 19th International Workshop on Semantic Evaluation (SemEval-2025)*, pages 2197–2204.

Reback, J., McKinney, W., Van Den Bossche, J., Augspurger, T., Cloud, P., Klein, A., Hawkins, S., Roeschke, M., Tratner, J., She, C., et al. (2020). pandas-dev/pandas: Pandas 1.0. 5. *Zenodo*.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. (2023). Llama: Open and efficient foundation language models.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W. X., Wei, Z., and Wen, J. (2024). A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6).

Wang, Z., Moriyama, S., Wang, W.-Y., Gangopadhyay, B., and Takamatsu, S. (2025). Talk structurally, act hierarchically: A collaborative framework for llm multi-agent systems.

Zhang, J., Fan, Y., Cai, K., Sun, X., and Wang, K. (2025). Osc: Cognitive orchestration through dynamic knowledge alignment in multi-agent llm collaboration.

Zhang, W., Zeng, L., Xiao, Y., Li, Y., Cui, C., Zhao, Y., Hu, R., Liu, Y., Zhou, Y., and An, B. (2026). Agentorchestra: Orchestrating multi-agent intelligence with the tool-environment-agent(tea) protocol.

Zhu, J.-P., Cai, P., Xu, K., Li, L., Sun, Y., Zhou, S., Su, H., Tang, L., and Liu, Q. (2024). Autotqa: Towards autonomous tabular question answering through multi-agent large language models. *Proc. VLDB Endow.*, 17:3920–3933.