TALON: A Multi-Agent Framework for Long-Table Exploration and Question Answering

Ruochun Jin[1], **Xiyue Wang**[1], **Dong Wang**[1†], **Haoqi Zheng**[1†], **Yunpeng Qi**[1], **Silin Yang**[2], **Meng Zhang**[1],

[1]College of Computer Science and Technology, National University of Defense Technology

[2]School of Computer Science, Peking University

Abstract

Table question answering (TQA) requires accurate retrieval and reasoning over tabular data. Existing approaches attempt to retrieve query-relevant content before leveraging large language models (LLMs) to reason over long tables. However, these methods often fail to accurately retrieve contextually relevant data which results in information loss, and suffer from excessive encoding overhead. In this paper, we propose TALON, a multi-agent framework designed for question answering over long tables. TALON features a planning agent that iteratively invokes a tool agent to access and manipulate tabular data based on intermediate feedback, which progressively collects necessary information for answer generation, while a critic agent ensures accuracy and efficiency in tool usage and planning. In order to comprehensively assess the effectiveness of TALON, we introduce two benchmarks derived from the WikiTableQuestion and BIRD-SQL datasets, which contain tables ranging from 50 to over 10,000 rows. Experiments demonstrate that TALON achieves average accuracy improvements of 7.5% and 12.0% across all language models, establishing a new state-of-the-art in long-table question answering. Our code is publicly available at: https://github.com/Wwestmoon/TALON.

1 Introduction

Table question answering (TQA) is the task of generating answers from tabular data in response to user queries (Pasupat and Liang, 2015), thereby enhancing data accessibility and usability without requiring specialized analytical skills. Recent studies have commonly employed large language models (LLMs) for this task by flattening entire tables into linearized textual sequences as input (Ye et al., 2023; Zhang et al., 2023; Wang et al., 2024b). However, LLMs often struggle to generalize effectively to long tables (*e.g.*, with more than 30 rows), with performance degrading as the table size increases (Chen, 2023). This issue is exacerbated when tables exceed the model's context window, since truncation inevitably discards essential information. To address these limitations, researchers have explored retrieval-augmented generation (RAG) (Sui et al., 2023; Lin et al., 2023; Chen et al., 2024) and Text-to-SQL (Nahid and Rafiei, 2024; Zhang et al., 2024; Cheng et al., 2023) approaches, both of which aim to supply LLMs with relevant table content instead of the full table. Specifically, RAG-based methods encode the table and retrieve rows or columns most relevant to the query embedding, while Text-to-SQL methods translate the query into SQL and execute it to obtain a targeted sub-table.

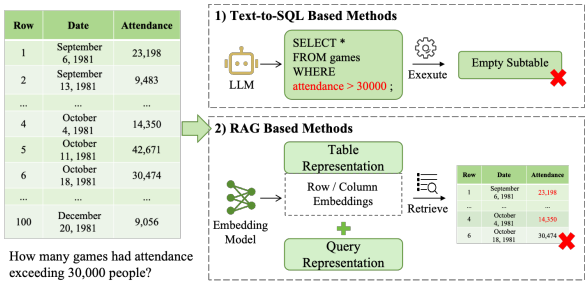

Figure 1: Failure case on Long-TQA. Text-to-SQL methods fail to recognize that the *attendance* column is stored as text, while RAG-based methods fail to retrieve relevant rows (e.g., *attendance* > *30,000*), as embedding models struggle to encode numerical comparison semantics.

Although these methods alleviate the limitations of long tables, they often fail to accurately understand the structure and content of long tables, which prevents them from extracting the information required to answer queries and results in inaccurate sub-tables. As illustrated in Figure 1, Text-to-SQL methods fail to recognize that the "*attendance*" column is stored as text, leading to erroneous outputs when `attendance > 30000` is executed. Similarly, RAG methods fail to interpret

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 27386–27402
November 4-9, 2025

numerical constraints, such as "*exceeding 30,000*", due to the limited capacity of embedding models to encode quantitative semantics. Moreover, RAG-based methods require encoding the entire table, incurring high computational cost and latency as the table size increases (Chen et al., 2024).

In order to overcome these challenges, we propose TALON, a multi-agent framework that dynamically gathers and processes relevant information to fully explore the structure and content of long tables, which relies solely on table schema as input without extra encoding. This design is inspired by human reasoning: similar to how humans first identify relevant areas and then iteratively refine their focus to answer a question. Specifically, TALON consists of three collaborative agents: (1) A planning agent that dynamically orchestrates tool usage based on the question and intermediate feedback; (2) A tool agent equipped with a suite of operations (*e.g.*, `get_value`, `get_column_meaning`, `python_code`) for content retrieval, structural analysis, and tabular data manipulation; (3) A critic agent that monitors execution correctness and plan effectiveness, which ensures robust and reliable reasoning.

As for evaluating TALON in long table question answering (Long-TQA), we construct two benchmarks WTQ-L and BirdQA, based on the WikiTableQuestion (Pasupat and Liang, 2015) and BIRD-SQL(Li et al., 2024a) datasets, featuring tables ranging from 50 to over 10,000 rows. Experimental results show that TALON significantly outperforms existing methods across multiple language models, achieving average accuracy improvements of 7.5% and 12.0%, and enabling scalable, robust reasoning over long tables.

Our contributions are three-fold as follows:

- We develop a specialized toolkit for tabular data that enables accurate content retrieval, structure analysis, and data manipulation;
- We first propose a multi-agent framework for Long-TQA, which progressively gathers relevant information and performs precise operations on the table to generate the final answer.
- We conduct experiments on datasets containing tables of varying lengths, ranging from 50 to over 10,000 rows, demonstrating the robustness and scalability of our approach on long tables.

2 Related Works

2.1 Table QA

Recent studies such as Dater(Ye et al., 2023), ReAcTable(Zhang et al., 2023), and their successors(Lu et al., 2024; Zhou et al., 2025) demonstrate strong reasoning and program synthesis capabilities for TQA. However, these methods require LLMs to process entire tables, which is often infeasible for long tables due to context length limitations.

To address the limitations of processing full tables, two main streams have emerged: Text-to-SQL and RAG methods. Text-to-SQL methods, such as MAC-SQL (Wang et al., 2025), TA-SQL (Qu et al., 2024), and DAIL-SQL (Gao et al., 2023), translate natural language queries into executable SQL statements over table schemas. Middleware (Gu et al., 2024) introduces a toolset for handling complex environments. TabSQLify (Nahid and Rafiei, 2024) and Alter (Zhang et al., 2024) first identify relevant sub-tables using Text-to-SQL and then apply LLMs for reasoning. While Text-to-SQL methods achieve strong performance on precise queries, they often struggle to align query intent with underlying value representations and to handle questions beyond SQL's expressive capabilities(Cheng et al., 2023). Moreover, SQL execution demands that tables and their values adhere to a strict structure, while ordinary tables can often accommodate numeric values stored as strings or columns with mixed types.

Classical RAG approaches (Lin et al., 2023; Sui et al., 2023) encode entire tables and retrieve the most relevant rows or columns to LLMs, which can be computationally expensive. TableRAG(Chen et al., 2024) leverages query expansion combined with schema and cell retrieval to pinpoint crucial information. While embeddings capture semantic information, they often struggle to preserve structural relationships across rows and columns, making it difficult to retrieve relevant context or summarize information spanning multiple rows and columns. In this paper, we introduce a set of tools for accurate table information retrieval, structural understanding, and data manipulation.

2.2 Multi-Agent System

LLMs have demonstrated exceptional capabilities in environmental interaction and decision-making, offering new possibilities for the development of intelligent agent systems (Li et al., 2024c). Inspired by the specialization and division of labor in hu-

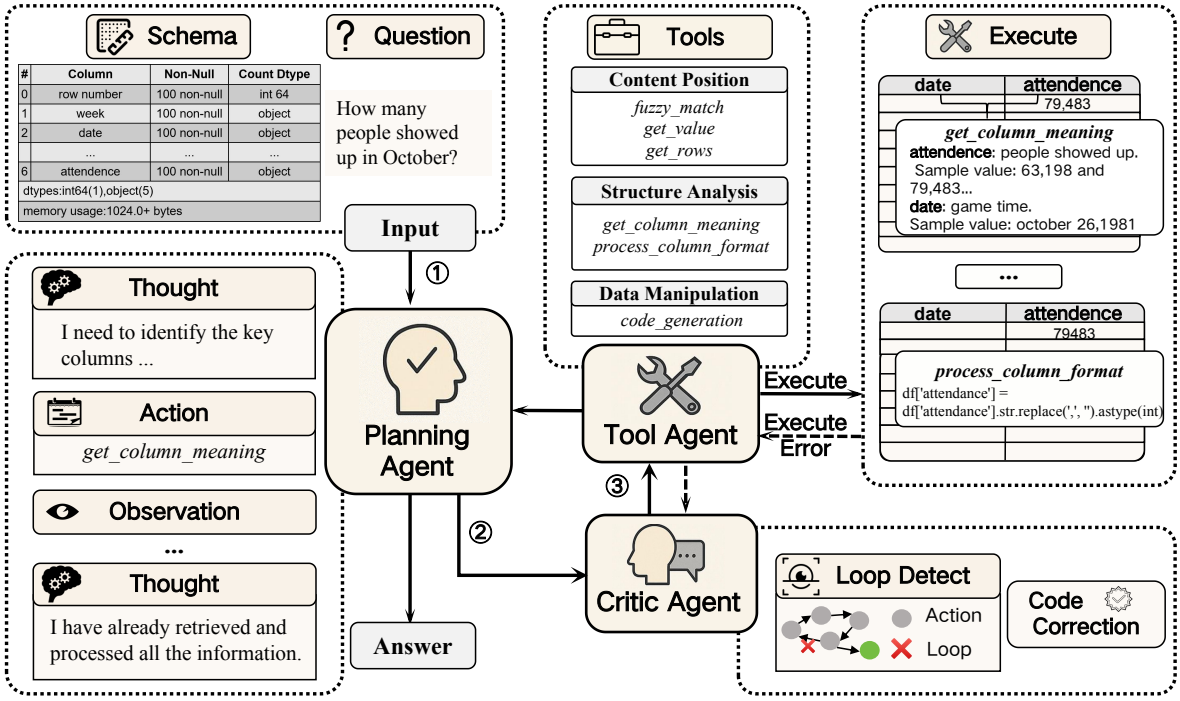

Figure 2: The framework of TALON, which includes: (1) A planning agent that dynamically orchestrates tool usage based on the question and intermediate feedback. (2) A tool agent equipped with a suite of operations for retrieving content, analyzing structure, and manipulating tabular data. (3) A critic agent that monitors execution correctness and plan effectiveness, ensuring robust and reliable reasoning.

man society, researchers have found that adopting a multi-agent collaborative paradigm can significantly improve the efficiency and accuracy of executing complex tasks (Qian, 2024; Islam et al., 2024; Li et al., 2024b).

The core idea of multi-agent systems is to decompose complex tasks into several subtasks, which are then completed through collaboration among agents to achieve the overall goal. Depending on the nature of the task, the collaboration modes among agents primarily follow three typical paradigms: first, linear pipeline collaboration, where agents process tasks in a fixed sequence (Yue et al., 2025; Liu et al., 2023; Niu et al., 2025; Zheng et al., 2025); second, collective decision-making, where agents reach consensus through negotiation (Cheng et al., 2024; Liang et al., 2024); and third, iterative optimization, where agents continuously improve output quality through feedback loops (Wang et al., 2024a; Tang, 2024). These collaboration methods provide the theoretical foundation and methodological support for building efficient multi-agent systems. In this paper, we propose a multi-agent framework that improves performance on Long-TQA by distributing the entire table QA process across three collaborative agents.

3 Method

In this paper, we propose a multi-agent framework for Long-TQA that facilitates accurate retrieval and reasoning over long tables. The framework consists of three specialized agents working collaboratively: the planning agent, which identifies appropriate tools for table interaction and formulates solution strategies based on intermediate feedback; the tool agent, which executes the selected tools to retrieve content, analyze table structure, and perform data manipulations; and the critic agent, which ensures the quality of the planning agent's strategy and the correctness of the tool agent's execution. To handle tables of varying lengths efficiently, we do not include the full content of tables in the prompt. Instead, tables are loaded into memory as Pandas DataFrames. The planning agent has access only to the table schema, which consists of column names and data types, while the tool agent is exclusively responsible for interacting directly with the table content.

3.1 Task Formulation

In Long-TQA, given a natural language question Q and a table T represented as: $T = \{v_{ij} \mid i = 1, \ldots, N;\ j = 1, \ldots, M\}$, where N is the number

of rows, M is the number of columns, and v_{ij} denotes the cell value at row i and column j. The objective is to produce an answer A through the collaborative efforts of LLMs denoted as $\mathcal{L}$.

3.2 Planning Agent

The planning agent generates reasoning steps and selects the most appropriate tools according to the contextual information it perceives. We adopt ReAct (Yao et al., 2023) as the backbone of our planning agent $\mathcal{L}_{\text{plan}}$, enabling the model to iteratively generate rationales and corresponding actions through intermediate feedback based on its own chain-of-thought (Wei et al., 2023). The planning agent allows concurrent invocations of the same tool with different arguments, enabling it to address multiple information needs in a single planning step and thereby reduce overall latency.

Formally, at each reasoning step $i \leq I_{\text{plan}}$, the planning agent produces a rationale r_i and an action $a_i = (\text{tool}_i(\text{args}_1), \ldots, \text{tool}_i(\text{args}_n))$, where $(r_i, a_i) \sim \mathcal{L}_{\text{plan}}(r_i, a_i \mid \tau_{i-1}, \phi_{\text{plan}})$, and $\tau_i = (r_1, a_1, o_1, \ldots, r_i, a_i, o_i)$ denotes the cumulative reasoning trajectory, ϕ_{plan} denotes the prompt of the planning agent (as detailed in Appendix E). The generated action a_i, after being validated by the critic agent, is then passed to the tool agent $\mathcal{L}_{\text{tool}}$, which returns an observation o_i that informs the planning agent's subsequent reasoning.

To improve robustness and consistency, we apply the self-consistency mechanism (Wang et al., 2023). For each input τ_{i-1}, we sample the planning agent $\mathcal{L}_{\text{plan}}$ k times to generate a set of candidate action sequences: $\{a_i^n\}_{n=1}^k = \{a_i^1, a_i^2, \ldots, a_i^k\}$, We then apply majority voting over the sampled sequences to select the most consistent plan:

$$a_i^* = \arg \max_{a \in \{a_i^n\}} \text{count}(a),$$

where $\text{count}(a)$ denotes the number of times sequence a appears among k samples.

3.3 Tool Agent

The tool agent receives tool names verified by the critic agent, executes the corresponding operations, and returns the resulting observations to the planning agent. Inspired by human-like strategies for processing long tables, we designed a comprehensive toolset that supports content retrieval, structural analysis, and data manipulation. The toolset is organized into three key categories:

(1) Table Content Retrieval, which utilizes `fuzzy_match`, `get_value`, and `get_row` tools by executing the pre-defined function to enable robust data localization and row-level extraction.

(2) Table Structure Analysis, which comprises tools of `get_column_meaning` and `process_column_format` by invoking the LLM for comprehending table structure and normalizing column content.

(3) Table Data Manipulation, where `code_generation` performs logical reasoning and numerical computation by invoking the LLM to generate python code, thereby mitigating hallucinations in LLM outputs. The pseudo-code of the toolset is provided in Appendix B.

Formally, upon receiving an action a_i validated by the critic agent, the tool agent $\mathcal{L}_{\text{tool}}$ executes the corresponding tool_i and generates an observation o_i depending on whether the tool need to invoked the LLM:

$$o_i = \begin{cases} f_{\text{tool}_i}(a_i, T), & \text{if not invoking LLM} \\ \mathcal{L}_{\text{tool}}(o_i \mid r_i, a_i, \phi_{\text{tool}_i}), & \text{otherwise} \end{cases}$$

where ϕ_{tool_i} denotes the prompt guiding the LLM-based tool (as detailed in Appendix E). The observation o_i is then returned to the planning agent to inform the subsequent reasoning steps.

3.4 Critic Agent

The critic agent reviews the plans generated by the planning agent, detects any loops, and forwards the corrected tool invocations to the tool agent. Moreover, it evaluates the quality of tool executions performed by the tool agent, corrects any erroneous outputs, and returns the revised code for re-execution.

Loop Resolution A loop is detected when an action generated in the current iteration duplicates any action from previous iterations. Formally, the loop detection function $L(\{a_1, \ldots, a_i\})$ is defined as:

$$L = \begin{cases} 1, & \text{if } \exists\, j < i \text{ such that } a_i \cap a_j \neq \emptyset, \\ 0, & \text{otherwise.} \end{cases}$$

If $L = 1$, indicating a detected loop, the critic agent $\mathcal{L}_{\text{critic}}$ is invoked to revise the current step. Given the previous trajectory τ_{i-1} and the repeated action a_i, the critic agent generates a revised rationale and an alternative action $(r_i^{\text{critic}}, a_i^{\text{critic}})$ by:

$$\mathcal{L}_{\text{critic}}(r_i^{\text{critic}}, a_i^{\text{critic}} \mid r_i, a_i, \tau_{i-1}, \phi_{\text{loop}}),$$

where ϕ_{loop} denotes the prompt guiding the critic (as detailed in Appendix E). The pair $(r_i^{\text{critic}}, a_i^{\text{critic}})$ is then appended to the trajectory τ_{i-1}, resulting in an updated trajectory τ_i and the revised action a_i^{critic} will deliver to the tool agent. This mechanism enables the system to mitigate redundant behavior and maintain effective reasoning.

Code Correction When the tool agent $\mathcal{L}_{\text{tool}}$ generates a python code snippet c_i that results in an execution error $e_i = \text{error}(c_i)$, the critic agent generates a revised python code:

$$c_i^{\text{corr}} \sim \mathcal{L}_{\text{critic}}\left(c_i^{\text{corr}} \mid c_i, e_i, \phi_{\text{critic}}\right),$$

where ϕ_{critic} denotes the prompt guiding the correction process (as detailed in Appendix E). The corrected code c_i^{corr} is then executed by the tool agent to obtain the output o_i^{corr}, which is subsequently passed to the planning agent for further decision making. This error correction mechanism prevents the propagation of errors and ensures that only valid observations are retained for subsequent reasoning steps.

4 Experiments

4.1 Experimental Setup

Datasets. To evaluate the effectiveness of the proposed multi-agent framework on Long-TQA, we constructed and utilized two key datasets: WikiTableQuestions-Large(WTQ-L) and BirdQA. Detailed statistics of these datasets are provided in Appendix A.

(1) **WTQ-L**: WikiTableQuestions (WTQ) (Pasupat and Liang, 2015) is a classic benchmark dataset for table reasoning. We construct a specialized subset, termed WTQ-L, consisting of 341 long tables with row counts ranging from 50 to over 500, covering long tables of different sizes. To enable a fine-grained analysis, we further stratify these tables into three categories based on their row counts: 50–100 rows, 100–200 rows, and 200+ rows. Comprehensive experiments are conducted across these stratified groups.

(2) **BirdQA**: Derived from the BIRD-SQL dataset (Li et al., 2024a), which focuses on large-scale database-driven Text-to-SQL tasks. We transform the original tasks into TQA tasks, extracting 314 question-answering instances involving a single table. The tables in the BirdQA dataset are of a huge scale, with an average of 44,000 rows, which can fully simulate the complexity and data volume of long tables in real business scenarios.

Baselines. We conduct comprehensive experiments comparing TALON against several baseline methods: End-to-End QA leverages an LLM to directly generate answers using the entire table as input, whereas Python QA generates and executes Python code to obtain answers. Binder (Cheng et al., 2023) decomposes questions into executable SQL sub-programs. Dater (Ye et al., 2023) employs a parsing-execution-filling strategy with sub-table decomposition, followed by answer generation using LLMs.

Both TableRAG (Chen et al., 2024) and RowCol-Retrieval encode tabular data to retrieve relevant information followed by program-assisted table manipulation and analysis using PyReact.

Metrics. Both datasets are evaluated using Exact Match (EM) accuracy.

Implementation Details. To ensure comprehensive evaluation, we conduct experiments with three LLMs: Qwen2.5-72B-Instruct (Qwen et al., 2025), DeepSeek-V3 (Liu et al., 2024a), and GPT-4o-mini (AI, 2023). For all baseline methods, we adopt their original settings, ensuring fair and reproducible comparisons. For all experiments of TALON, we set the temperature as 0.7, Top P as 0.95, Top K as 5, and maximum output length as 1024. We also implement the self-consistency mechanism by sampling 5 independent reasoning paths. For the Qwen2.5-72B-Instruct model, we perform 5 independent generations and report the average results due to the single-generation constraint. We set the maximum planning iterations to $I_{\text{plan}} = 10$ and the maximum refinement iterations for the critic agent to $I_{\text{critic}} = 3$. We process tabular data as a pandas DataFrame, incorporate the data schema into the prompt, and employ zero-shot learning to generate answers. Agent prompts are provided in Appendix E.

4.2 Main Results

The performance of TALON was benchmarked against several baseline methods across two datasets, WTQ-L and BirdQA, utilizing three distinct large language models (GPT-4o-mini, Qwen2.5-72B-Instruct, and DeepSeek-V3), with average performance also reported. The results, as presented in Table 1, are summarized as follows:

1. TALON consistently outperformed all baseline methods across all datasets and models. As indicated by the bolded scores in the table, TALON achieved the highest performance in every evaluated scenario. For instance, on the

Method	GPT-4o-mini		Qwen2.5-72B-Ins		DeepSeek-V3		Average.	
	WTQ-L	BirdQA	WTQ-L	BirdQA	WTQ-L	BirdQA	WTQ-L	BirdQA
End-to-End QA	55.5	-	34.0	-	58.3	-	49.3	-
Python QA	53.8	-	36.3	-	52.5	-	47.5	-
Binder	61.0	60.8	60.3	59.2	56.7	59.6	59.3	59.9
Dater	54.7	-	51.2	-	46.1	-	50.7	-
RowColRetrieval	47.2	48.4	49.9	42.7	51.6	53.8	49.6	48.3
TableRAG	52.1	54.7	59.6	53.5	59.1	60.2	56.9	56.1
TALON (Ours)	**67.7** ↑6.7	**72.0** ↑11.2	**66.6** ↑6.3	**70.0** ↑10.8	**66.2** ↑7.1	**73.7** ↑13.5	**66.8** ↑7.5	**71.9** ↑12.0

Table 1: Table reasoning results on WTQ-Large and BirdQA with GPT-4o-mini, Qwen2.5-72B-Instruct, DeepSeek-V3, and . Bold denotes the best performance and underline denotes the second-best performance, a dash ("–") indicates that the input exceeded the model's context window limit. The red upward arrows indicate the margin of improvement over the second-best performing method.

WTQ-L dataset, TALON with GPT-4o-mini scored 67.7%, surpassing the second-best method, Binder, with the improvement of 6.7%. Similarly, on the BirdQA dataset with DeepSeek-V3, TALON achieved 73.7%, outperforming TableRAG by 7.1%. Overall, TALON outperforms the second-best model by an average of 7.5% on WTQ-L and 12.0% on BirdQA. This consistent outperformance demonstrates the robust and superior capabilities of TALON.

2. TALON demonstrates consistent performance across diverse LLM backbones. On the WTQ-L dataset, TALON outperforms the second-best methods by 6.7% on GPT-4o-mini, 6.3% on Qwen2.5-72B-Instruct, and 7.1% on DeepSeek-V3, with the performance gap being relatively narrow, ranging from 0.4% to 0.7%. On the BirdQA dataset, the improvements are 11.2% on GPT-4o-mini, 10.8% on Qwen2.5-72B-Instruct, and 13.5% on DeepSeek-V3, with the difference range being slightly wider but still comparable, around 2.7%. This consistency demonstrates that the TALON architecture effectively enhances Long-QA performance across different models, showcasing its strong backbone-agnostic stability.

4.3 Analysis

We conduct extensive experiments to analyze the following questions:

1. *Does TALON still perform well on the WTQ dataset?*

Due to computational constraints, we evaluate on a randomly sampled subset of 500 tables from the original WTQ dataset. Full-dataset results with Qwen2.5-72B-Instruct are provided in Appendix C.2. The results are presented in Table 2. Experimental results show that our method also outperforms baselines. When using GPT-4o-mini, it achieves a 9.6% improvement compared to the second-best performing method. Similarly, it demonstrates consistent gains across different models, with an average improvement of 9.4% on Qwen2.5-72B-Instruct and a 5.9% improvement on DeepSeek-V3 compared to the second-best performing method. These results highlight the universality of our approach, confirming that its effectiveness is independent of table length.

Method	GPT-4o mini	Qwen2.5 72B-Ins	DeepSeek V3	Avg.
End-to-End QA	56.3	42.6	61.9	53.6
Python QA	51.2	41.9	55.0	49.4
Binder	61.4	58.5	59.0	59.6
Dater	59.8	60.2	64.0	61.3
RowColRetrieval	55.2	53.4	54.0	54.2
TableRAG	55.2	59.6	62.0	58.9
TALON (Ours)	**71.0** ↑9.6	**69.6** ↑9.4	**69.9** ↑5.9	**70.2** ↑8.9

Table 2: Results on WTQ with GPT-4o-mini, Qwen2.5-72B-Instruct, and DeepSeek-V3. Bold denotes the best performance, and underline denotes the second-best performance. The red upward arrows indicate the margin of improvement over the second-best performing method.

2. *How do different lengths in WTQ-L affect the performance of the method?*

Table 3 presents the performance of TALON on WTQ-L across different table length intervals: 50–100, 100–200, and 200+ rows. The results demonstrate that our method consistently outperforms all baseline approaches across all length categories. Furthermore, as shown in Table 2, Dater outperforms Binder on the WTQ dataset, but underperforms on the WTQ-L dataset. This performance

Method	GPT-4o-mini			Qwen2.5-72B-Ins			DeepSeek-V3		
	50-100	100-200	200+	50-100	100-200	200+	50-100	100-200	200+
End-to-End QA	56.7	55.6	54.2	33.3	43.8	25.0	55.6	61.1	58.3
Python QA	51.2	54.6	55.6	32.3	41.1	35.4	58.1	55.6	43.8
Binder	55.6	66.7	60.6	55.6	66.7	58.5	54.0	63.3	52.8
Dater	51.5	59.8	52.8	50.5	57.8	45.3	53.5	43.3	41.5
RowColRetrieval	49.5	42.2	50.0	47.5	52.2	50.0	47.5	51.1	56.3
TableRAG	49.5	56.7	50.0	56.1	64.4	58.3	55.6	63.3	58.3
TALON (Ours)	**65.2** ↑8.5	**71.1** ↑4.4	**66.7** ↑6.1	**64.8** ↑8.7	**70.0** ↑3.3	**65.0** ↑6.5	**64.8** ↑6.7	**73.3** ↑10.0	**60.4** ↑2.1

Table 3: Performance of different methods on subsets with varying row counts of WTQ-L. Bold denotes the best performance, and underline denotes the second-best performance. The red upward arrows indicate the margin of improvement over the second-best performing method.

discrepancy can be attributed to Dater's approach of encoding the entire table into the LLM during inference, which leads to degraded effectiveness on long tables. We further compare TALON against the End-to-End QA baseline using GPT-4 on WTQ-L: End-to-End QA achieves only 55.65%, while TALON reaches 69.64%. As the number of rows increases, it becomes progressively more challenging for the LLM to identify relevant rows and correctly evaluate numerical conditions accurately, which aligns with the findings reported in (Chen, 2023).

3. *How does TALON compare with the state-of-the-art Text-to-SQL methods?*

Since BirdQA is derived from the Bird-SQL benchmark, we additionally evaluate our method against mainstream Text-to-SQL approaches. As shown in Table 4, TALON achieves the highest accuracy compared to directly applying Text-to-SQL methods, thereby validating both the representativeness of BirdQA and the effectiveness of TALON. We evaluated MAC-SQL (Wang et al., 2025) with GPT-4o-mini using oracle schema access and obtained 50.79% accuracy. This relatively low performance can be attributed to the challenge LLMs face when generating precise SQL queries in a single step. In contrast, TALON decomposes complex problems into multiple steps and leverages tool invocation to generate intermediate results, such as filtered rows or computed aggregates, which allows the LLM to more effectively retrieve question-relevant information from the table and to facilitate subsequent reasoning steps.

Method	EM(%)
Bird (Li et al., 2024a)	55.8
MAC-SQL (Wang et al., 2025)	59.6
TA-SQL (Qu et al., 2024)	56.1
DAIL-SQL (Gao et al., 2023)	54.8
TALON (Ours)	**72.0**

Table 4: Comparison of TALON with mainstream methods of Text-to-SQL on BirdQA.

5. *Can TALON decrease computational cost?*

As shown in Figure 3, the token consumption of TALON remains largely stable regardless of the table length, since it only requires the table schema rather than the full table content. End-to-End QA consumes fewer tokens than TALON for shorter tables; however, its token usage increases rapidly as the table length grows. Dater, on the other hand, consistently exhibits high token consumption, which also increases significantly with the table length.

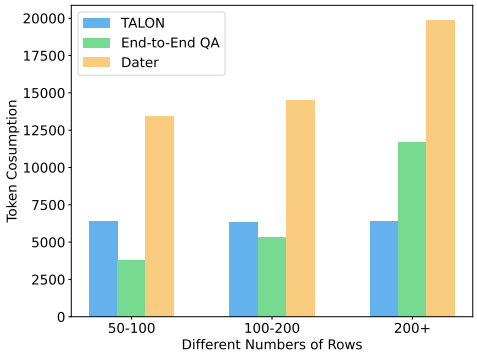

Figure 3: Token consumption across different methods.

4. *Does TALON have preferences when choosing tools?*

As shown in Figure 4, `Get_Row` and `Get_Value` are the two most frequently invoked functions, highlighting that accurately locating the answer remains a key challenge in solving TQA tasks. An-

other important tool is `Python_Code`, which is crucial for handling computational tasks, especially when dealing with ultra-long tables such as those in BirdQA. This suggests that, given sufficient memory, converting long tables into DataFrames for processing is a viable solution. Moreover, the tool selection is not uniform across datasets, reflecting its adaptability to different data characteristics and task requirements.

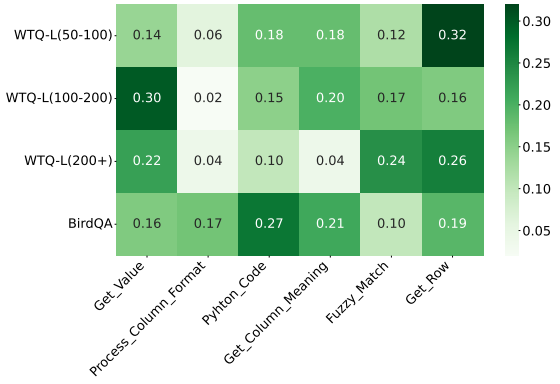

Figure 4: Distribution of tools by datasets.

6. *What is the iteration distribution of the datasets?*

As illustrated in Figure 5, both WTQ-L and BirdQA exhibit similar trends in iteration distribution. Compared to WTQ-L, BirdQA exhibits a higher concentration of items in earlier iterations, with a peak at iteration 3. In contrast, WTQ-L shows a more gradual trend, peaking at iteration 4. This difference may be attributed to WTQ-L's inherently diverse table structures and the complexity of its question formulations, which likely require additional iterations for effective resolution. Notably, both datasets converge to minimal activity beyond iteration 6, suggesting that the models are generally capable of resolving most tasks efficiently within a limited number of iterations.

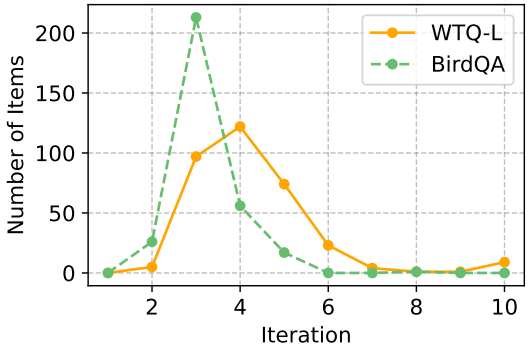

Figure 5: Number of items per iteration for each dataset.

7. *What is the performance when using different base LLMs for different roles?*

We conducted experiments with different base LLMs for each role in TALON to evaluate the effectiveness of our multi-agent design. In this setup, the planning agent employs Qwen-2.5-72B-Instruct for robust reasoning and planning, the tool agent relies on the smaller Qwen-2.5-7B-Instruct for efficient tool execution, and the critic agent utilizes GPT-4o to provide high-fidelity evaluation of both planning and tool execution. This configuration achieved an overall accuracy of 67.26% on the WTQ-L dataset, supporting our intuitive design of a strong planning agent, a lightweight tool agent, and a powerful critic agent.

4.4 Ablation Study

We conduct ablation experiments on each key module on WTQ-L using GPT-4o-mini, with the results shown in Table 5. The Self-Consistency module has the most significant impact, with a 4.3% drop in performance when removed. The critic agent is also a key component of TALON that plays a crucial role in refining the model's decision-making process. When removed, the performance drops by 3.0%, indicating its significant contribution to the multi-agent framework. More specifically, both loop resolution and code correction in the critic agent are equally important. Removing them leads to performance drops of 1.8% and 1.2%, respectively.

Method	EM (%)
TALON	67.0
w/o Self-Consistency	62.7 (↓4.3)
w/o Critic Agent	64.0 (↓3.0)
w/o Loop Resolution	65.2 (↓1.8)
w/o Code Correction	65.8 (↓1.2)

Table 5: Ablation study of each module on WTQ-L.

We then investigated the role of each tool on BirdQA, with the results shown in Table 6. When `Python_Code` was removed, the average number of iterations increased from 3 to 7, resulting in a significant performance drop of 44.4%. This decline can be attributed to the large size of the BirdQA dataset, which necessitates substantial data aggregation and consumption. Without python code, this process would depend on LLMs to gather data incrementally. The tool `get_column_meaning` also plays a critical role, as it helps the model effec-

Setting	EM(%)
w/ all	72.0
w/o get_value	69.4 (↓2.6)
w/o fuzzy_match	68.5 (↓3.5)
w/o get_row	70.4 (↓1.6)
w/o get_column_meaning	63.4 (↓8.6)
w/o process_column_format	69.4 (↓2.6)
w/o python_code	27.6 (↓44.4)

Table 6: Ablation study of each tool on BirdQA.

tively understand the table's structure and content. Removing any other tool similarly led to a decrease in performance metrics, emphasizing the indispensable and unique contribution of each function.

To explain the proposed function calls, we performed an additional experiment with our method using only the `Python code` on the WTQ-L dataset, with the result shown in Table 7. Consider a common practical challenge: user queries often do not exactly match the table content. For example, when a user requests a "**16mm**" camera, directly calling a Python function like `get_value("16mm")` would return an empty value if the table stores "**16 mm**" with a space, leading to retrieval failure and subsequent error propagation. In our method, when `get_value` fails to return a match, the model automatically invokes a `fuzzy_matching` tool, which returns potentially matching values and corresponding row indexes such as "**16 mm**", helping the model better understand the table content and plan the next steps more accurately.

Method	**50–100**	**100–200**	**200+**	**Average**
PYREACT	45.45	55.56	41.67	47.53
TALON (Ours)	**65.2**	**71.1**	**66.7**	**67.70**

Table 7: Ablation study on the impact of tool usage for TALON on WTQ-L.

5 Conclusion

In this paper, we propose a multi-agent framework for Long-TQA, **TALON**, which improves performance by distributing the table QA process across three specialized agents, including a planning agent, a tool agent, and a critic agent. We develop a specialized toolkit, inspired by human-like table processing, that enables accurate content retrieval, structural analysis, and data manipulation on tabular data. Furthermore, to comprehensively evaluate the performance of our method, we developed two benchmarks, WTQ-L and BirdQA, with table sizes ranging from 50 to over 10,000 rows. Experiments on both datasets demonstrate that our method achieves state-of-the-art results, with average improvements of 7.5% and 12.0%, while maintaining robustness and scalability, making it well-suited for real-world deployment.

Limitations

Due to budget constraints, we conduct experiments on only three models, and testing on more models could provide stronger evidence. Due to the scarcity of long-table datasets, we plan to focus on constructing more benchmarks for long-table QA and conducting further experimental validation in the future. Our method primarily focuses on structured tables with well-defined schemas. For hierarchical or flattened tables without explicit schemas, a preprocessing step is required to convert them into a standardized dataframe representation. This limitation introduces additional complexity. Extending our framework to directly handle such table formats represents a key direction for future work.

Acknowledgment

This work was supported by NSFC No.62302503, NUDT Youth Independent Innovation Science Fund Project Grant No.ZK23-15, the Open Research Fund from State Key Laboratory of High Performance Computing of China Grant No.202401-09, and Young Elite Scientists Sponsorship Program by CAST No. YESS20240767.

References

Open AI. 2023. Gpt-4 technical report. arXiv preprint arXiv:2303.08774.

Si-An Chen, Lesly Miculicich, Julian Martin 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. Preprint, arXiv:2410.04739.

Wenhu Chen. 2023. Large language models are few(1)-shot table reasoners. Preprint, arXiv:2210.06710.

Yi Cheng, Wenge Liu, Jian Wang, Chak Tou Leong, Yi Ouyang, Wenjie Li, Xian Wu, and Yefeng Zheng. 2024. Cooper: Coordinating specialized agents towards a complex dialogue goal. Proceedings of the AAAI Conference on Artificial Intelligence, 38(16):17853–17861.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. 2023. Binding language models in symbolic languages. In The Eleventh International Conference on Learning Representations.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. Preprint, arXiv:2308.15363.

Yu Gu, Yiheng Shu, Hao Yu, Xiao Liu, Yuxiao Dong, Jie Tang, Jayanth Srinivasa, Hugo Latapie, and Yu Su. 2024. Middleware for llms: Tools are instrumental for language agents in complex environments. Preprint, arXiv:2402.14672.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. Preprint, arXiv:2405.11403.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2024a. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. Advances in Neural Information Processing Systems, 36.

Kenneth Li, Aspen K. Hopkins, David Bau, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. 2024b. Emergent world representations: Exploring a sequence model trained on a synthetic task. Preprint, arXiv:2210.13382.

Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. 2024c. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. Vicinagearth, 1(1):9.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2024. Encouraging divergent thinking in large language models through multi-agent debate. Preprint, arXiv:2305.19118.

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), pages 9909–9926.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. arXiv preprint arXiv:2412.19437.

Tianyang Liu, Fei Wang, and Muhao Chen. 2024b. Rethinking tabular data understanding with large language models. In Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 450–482, Mexico City, Mexico. Association for Computational Linguistics.

Zhiwei Liu, Weiran Yao, Jianguo Zhang, Le Xue, Shelby Heinecke, Rithesh Murthy, Yihao Feng, Zeyuan Chen, Juan Carlos Niebles, Devansh Arpit, Ran Xu, Phil Mui, Huan Wang, Caiming Xiong, and Silvio Savarese. 2023. Bolaa: Benchmarking and orchestrating llm-augmented autonomous agents. Preprint, arXiv:2308.05960.

Xinyuan Lu, Liangming Pan, Yubo Ma, Preslav Nakov, and Min-Yen Kan. 2024. Tart: An open-source tool-augmented framework for explainable table-based reasoning. Preprint, arXiv:2409.11724.

Md Mahadi Hasan Nahid and Davood Rafiei. 2024. TabSQLify: Enhancing reasoning capabilities of LLMs through table decomposition. In Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 5725–5737, Mexico City, Mexico. Association for Computational Linguistics.

Zihan Niu, Zheyong Xie, Shaosheng Cao, Chonggang Lu, Zheyu Ye, Tong Xu, Zuozhu Liu, Yan Gao, Jia Chen, Zhe Xu, and 1 others. 2025. Part: Enhancing proactive social chatbots with personalized real-time retrieval. In Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 4269–4274.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. arXiv preprint arXiv:1508.00305.

Chen and Qian. 2024. ChatDev: Communicative agents for software development. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. 2024. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-sql generation. Preprint, arXiv:2405.15307.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025. Qwen2.5 technical report. Preprint, arXiv:2412.15115.

Yuan Sui, Jiaru Zou, Mengyu Zhou, Xinyi He, Lun Du, Shi Han, and Dongmei Zhang. 2023. Tap4llm: Table provider on sampling, augmenting, and packing semi-structured data for large language model reasoning. arXiv preprint arXiv:2312.09039.

Xunzhu and Tang. 2024. CodeAgent: Autonomous communicative agents for code review. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 11279–11313, Miami, Florida, USA. Association for Computational Linguistics.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. Mac-sql: A multi-agent collaborative framework for text-to-sql. Preprint, arXiv:2312.11242.

Qineng Wang, Zihao Wang, Ying Su, Hanghang Tong, and Yangqiu Song. 2024a. Rethinking the bounds of llm reasoning: Are multi-agent discussions the key? Preprint, arXiv:2402.18272.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. Preprint, arXiv:2203.11171.

Zilong Wang, Hao Zhang, Chun-Liang Li, Julian Martin Eisenschlos, Vincent Perot, Zifeng Wang, Lesly Miculicich, Yasuhisa Fujii, Jingbo Shang, Chen-Yu Lee, and Tomas Pfister. 2024b. Chain-of-table: Evolving tables in the reasoning chain for table understanding. Preprint, arXiv:2401.04398.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. Preprint, arXiv:2201.11903.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. Preprint, arXiv:2210.03629.

Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. 2023. Large language models are versatile decomposers: Decompose evidence and questions for table-based reasoning. In Special Interest Group on Information Retrieval.

Shengbin Yue, Siyuan Wang, Wei Chen, Xuanjing Huang, and Zhongyu Wei. 2025. Synergistic multi-agent framework with trajectory learning for knowledge-intensive tasks. Preprint, arXiv:2407.09893.

Han Zhang, Yuheng Ma, and Hanfang Yang. 2024. Alter: Augmentation for large-table-based reasoning. arXiv preprint arXiv:2407.03061.

Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. 2023. Reactable: Enhancing react for table question answering. Preprint, arXiv:2310.00815.

Haoqi Zheng, Dong Wang, Silin Yang, Yunpeng Qi, Ruochun Jin, and Liyang Xu. 2025. Logical DA: Enhancing data augmentation for logical reasoning via a multi-agent system. In Findings of the Association for Computational Linguistics: ACL 2025, pages 6843–6855, Vienna, Austria. Association for Computational Linguistics.

Wei Zhou, Mohsen Mesgar, Annemarie Friedrich, and Heike Adel. 2025. Efficient multi-agent collaboration with tool use for online planning in complex table question answering. Preprint, arXiv:2412.20145.

A Dataset Information

WTQ-L is sampled from the WikiTableQuestion dataset, covering the portion of the WTQ testset with more than 50 rows. Furthermore, we divide it into three subsets: one containing tables with 50-100 rows, another with 100-200 rows, and the last with tables containing more than 200 rows. BirdQA is sampled from Bird-SQL. Compared to WTQ-L, BirdQA contains more rows that exceed the context window limit of LLM, presenting a greater challenge to the model. More detailed information, including the number of samples and the average number of lines, can be found in Table 8.

Dataset	Sample Range	Sample Nums	Average Rows
WTQ-L	50-100	198	62.03
	100-200	90	122.67
	200+	53	360.28
BirdQA	-	314	44488.52

Table 8: Sample numbers and average rows for WTQ-L and BirdQA datasets.

B Tools Information

We designed a comprehensive toolset that supports content retrieval, structural analysis, and data manipulation. The relevant tools are summarized in Table 11: the left column presents the tool names along with their descriptions, while the right column outlines the algorithmic logic of each tool. For fuzzy matching, we employ the `token_set_ratio` function from the fuzzywuzzy library to measure the similarity between query strings and table cell contents. A higher score reflects a greater degree of lexical overlap between the compared strings. To improve efficiency, we incorporate parallel computation, allowing the method to process large-scale tables containing more than 20,000 rows while maintaining response times within a few hundred milliseconds. Based on empirical analysis, we set the fuzzy matching threshold to 70%.

C Supplementary Experiments

C.1 Supplementary Baseline Evaluation

To ensure a comprehensive and fair evaluation, we include several widely adopted baselines of TableQA for an extended comparison on the WTQ-L dataset using GPT-4o-mini, following the implementation details in Section 4.1. Specifically, we reproduce:

- **RethinkTable** (Liu et al., 2024b), which adopts Direct Prompting and Pyagent with Mix-Self-Consistency and Normalization;
- **TabSQLify** (Nahid and Rafiei, 2024), which first identifies relevant sub-tables using Text-to-SQL and then applies LLMs for reasoning;
- **Chain of Table** (Wang et al., 2024b), which incorporates function calls to iteratively update the table and form a tabular reasoning chain;
- **MACT** (Zhou et al., 2025), which involves a planning agent and a coding agent that collaboratively use tools to answer questions.

As shown in Table 9, TALON consistently outperforms all baseline methods across different table sizes, further demonstrating the robustness of our approach under varying table scales.

Method	50-100	100-200	200+	Average
ReThinkTable	62.12	66.67	60.38	63.05
MACT	60.61	65.56	58.33	61.56
TablSQLify	63.13	56.67	56.60	60.40
TALON(Ours)	**65.20**	**71.10**	**66.70**	**67.03**

Table 9: Comparison of TALON with mainstream TableQA methods on WTQ-L.

C.2 Performance on the full WTQ dataset.

We evaluated these methods on the full WTQ dataset using the Qwen-2.5-72B-Instruct model in the table 10. These results demonstrate that our proposed method achieves comparable results, indicating its generalization ability and effectiveness across diverse table reasoning scenarios.

Method	WTQ
MACT (Zhou et al., 2025)	69.4
RethinkTable (Liu et al., 2024b)	69.6
Chain-of-table (Wang et al., 2024b)	68.3
TALON(Ours)	**69.7**

Table 10: Performance Comparison on WTQ.

D Case Study

We present a concrete case demonstrating how TALON leverages multi-agent collaboration to answer a question based on a large table by dynamically gathering and processing relevant tabular information. The table and corresponding question are shown below.

Table with More Than 100 Rows:

Common name	District	Population (2009)	...	Mayor
Acre	North	46,300	...	Shimon Lancry
Afula	North	40,500	...	Avi Elkabetz
...	...	...	...	...
Bat Yam	Tel Aviv	130,000	...	Shlomo Lahiani

Question: What is the number of cities located in either the **Tel-Aviv** or **Center** districts possessing a population less than 65,000?

Given a table and a user query, TALON first analyzes the query to identify relevant columns by invoking the `get_column_meaning` tool. It then applies `fuzzy_matching` to locate the rows and columns containing the target districts. The results indicate that both the *Common name* and *District* columns include "Tel Aviv". TALON then combines this information with the user query to identify the truly relevant column, *District*, thereby ensuring that subsequent computations are performed on the correct column and preventing errors in the final answer. Since the question requires numeric computation, TALON examines the *population* column and finds that its values are stored as strings. It calls the `process_column_format` tool to convert this column into integers. Finally, TALON generates Python code to compute the answer. Figure 6 shows the reasoning trace.

E Prompt

The prompts used in our method are shown in Figures 7 to 11.

Step-by-Step Trace

1. **Thought:** I need to understand the meaning of each column for better information retrieval.

   Action: `get_column_meaning()`

   Observation: Population (2009): Number of inhabitants in the city as of 2009.

2. **Thought:** Locate occurrences of "Tel Aviv" and "Center".

   Action: `fuzzy_match("Tel Aviv"), fuzzy_match("Center")`

   Observation: The columns below may contain relevant values for 'Tel Aviv':

   Common name: 'Tel Aviv', The index is...

   District: 'Tel Aviv', The index is...

3. **Thought:** Convert `Population (2009)` to a numeric format for proper filtering.

   Action: `process_column_format("Population (2009)")`

   Observation: The column `Population (2009)` has been processed...

4. **Thought:** Write Python code to compute the desired count.

   Action:

```
filtered_cities = df[(df["District"].isin(["Tel Aviv", "Center"]))
                  & (df["Population (2009)"] < 65000)]
result = len(filtered_cities)
```

   Observation: The final answer is...

Figure 6: Step-by-step reasoning trace demonstrating our method's approach to processing the query "Find cities in Tel Aviv and Center districts with population less than 65,000". The trace shows: (1) understanding column semantics, (2) fuzzy value matching, (3) data type processing, and (4) query execution with final result.

Tools	Code

`Get_Value`
Description: Search for a specific value.
	Function `Get_Value`($table$, $value$)**:**
	$results \leftarrow \emptyset$
	foreach column c in $table$ **do**
	  foreach row r in c **do**
	  if $r = value$ **then**
	  $results \leftarrow results \cup \{(c.name, r.index)\}$
	return if $results = \emptyset$ **then** ($"notfound"$, $null$, $null$) **else** $results$

`Get_Row` **Description:** Return the row content of the row index.	**Function** `Get_Row`($table$, row_index)**:** `// Return the row content of the row index` **return** $table[row\_index]$

`Fuzzy_Match`
Description: Identify potential variations of the value.
	Function `Fuzzy_Match`($value$, $table$, k)**:**
	$matches \leftarrow [\,]$ **foreach** column c in $table$ **do**
	  foreach cell v in c **do**
	  $score \leftarrow$ FuzzyMatch($value, v$)
	  if $score$ is above threshold **then**
	  Append $(v, c.name, score)$ to $matches$
	Sort $matches$ by $score$ in descending order
	return top-k entries from $matches$

`Get_Column_Meaning`
Description: Identify the key column and corresponding meaning based on the question.
	Function `Get_Column_Meaning`($question$, $table$, k)**:**
	$columns, meanings \leftarrow LLM(question, table)$
	$results \leftarrow \emptyset$
	foreach column c in $columns$ **do**
	  if c is relevant **then**
	  $samples \leftarrow$ Randomly select k values from column c in $table$
	  Add $(c, meanings[c], samples)$ to $results$
	return $results$

`Process_Column_Format`
Description: Process the format of a specific column based on the thought and question.
	Function `Process_Column_Format`($column$, $question$, $thought$)**:**
	$samples \leftarrow$ Randomly select k values from column c in $table$
	$need\_processing, code \leftarrow LLM(question, column, thought)$
	if $need\_processing$ **then**
	  $Execute(code)$
	  $samples \leftarrow$ Get sample values from $column$
	return $(samples)$

`Python_Code` **Description:** Generate Python code for numerical computation and logic reasoning.	**Function** `Python_Code`($table$, $trajectory$)**:** `// Generate Python code using LLM` $code \leftarrow LLM(table, trajectory)$ **return** $code$

Table 11: Functions and corresponding pseudo-code cross-reference table of the toolset.

Given a table schema of a large table, you need to answe the question based on the table. The table is a pandas dataframe in Python. The name of the dataframe is `df`. Your task is to use tools to answer the question.

Tools:
- **get_column_meaning**() […]
- **find_columns_containing_value_fuzzy**(value：str) […]
- **find_column_format(column：str)** […]
- **get_value(value：str)** […]
- **get_row(row_index：str)** […]
- **def()** […]

Response Format:
- Strictly follow the given format to respond:
  - **Thought**: you should always print about the thinking process about what to do based on the previous observation.
  - **Action**: Directly output the tool you chose to use.
  - **Observation**: the result of the action.
  - ... (this Thought/Action/Observation can repeat N times)
  - **Final Answer**: the final answer to the original input question, only print the answer.

Notes:
- Do not use markdown or any other formatting in your responses.
- Ensure the last line is only "Final Answer: answer".
- Directly output the Final Answer rather than outputting by Python.
- Ensure to have a concluding thought that verifies the table, observations and the statement before giving the final answer.
- You can call a same tool multiple times simultaneously. For example, you can output Action: tool1(args1), tool1(args2) at one same time.

Now, given a table , please answer the question: "{query}".
{table_schema}
The table is a pandas dataframe in Python. The name of the dataframe is `df`. Your task is think step by step, use tools to answer the question based on the table.

Begin!

Figure 7: The prompt of the planning agent.

Given a partial table preview along with a thought, focusing only on information relevant to the question and the thought, analyze the table and extract only the most relevant information.

Note:
- the given rows only contain a small portion of the table, not indicate the entire table.
- Directly output the most relevant columns that might be related to the question and the thought, and the column meaning of a column.
- Directly output the columns and column meaning without any additional text or explanations.

Output Format:

Column name:
{Column Meaning}

Table preview:
{sample_rows}

Question:
{question}

Thought:
{thought}

Figure 8: The prompt for get_column_meaning tool in the tool agent.

The column type of '{column_name}' is {column_dtype}.
Here are some example values from the column: {sampled_values}.
You should determine whether there is a need to process the column using Python code for the given purpose.
Determine whether the given column needs processing using Python based on its content and format.
The column can be processed by df["{column_name}"]

Purpose:
{thought}

Notes:
If the column's format needs additional transformation to calculate the result based on the purpose, then it needs to be processed and transformed into the correct type.

- Use df to process the column, and output the code in the format of:
```python
# your code here
```
- Only generate code to process the column.
- If the column's format is already correct, just return "[[No]]".

Figure 9: The prompt for process_column_format tool in the tool agent.

You have to read the table and the query, and all the thinking process, to identity the root cause of the error.
Analyze the execution trace and error message to:

1. Diagnose the root cause of the failure based on the error message and the trajectory.
2. Validate the code logic against the provided DataFrame (df) based on the observations.
3. Formulate a corrected code.

Critical Constraints:
- The table is already available as a pandas DataFrame (df).
- Never sample the table data during analysis.

Output Format:
You should directly output the revised python code with the format of ```python\n``` to avoid such an error,no other additional text.

Here is the thinking process:
{trajectory}

Here is the error message:
{error}

Figure 10: The loop resolution prompt of the critic agent.

As an AI assistant analyzing tabular data, I've noticed you've repeatedly called the same data processing tool multiple times.
Based on previous observations, please:

1. Re-evaluate your current approach
2. Generate a new 'thought' process that considers:
- Efficiency improvements
- Avoiding redundant operations including all the history actions
- Leveraging previously obtained results from the trajectory

3. Provide a revised 'action' plan that:
- avoid duplicate tool calls
- generate a tool or final answer based on the trajectory

Format your response with clear "**Thought**:" and "**Action**:" sections, ensuring the solution is both technically sound and resource-efficient.

Output Format:
- Thought: Only print about the thinking process about what to do next .
- Action: Directly output the tool you chose to use.

Here is the trajectory:
{trajectory}

Here is the repeated action:
{action}

Figure 11: The code correction prompt of the critic agent.