AutoTQA: Towards Autonomous Tabular Question Answering through Multi-Agent Large Language Models

Jun-Peng Zhu
East China Normal University
& PingCAP, China
zjp.dase@stu.ecnu.edu.cn

Peng Cai
East China Normal University
pcai@dase.ecnu.edu.cn

Kai Xu
PingCAP, China
xukai@pingcap.com

Li Li
PingCAP, China
lili@pingcap.com

Yishen Sun
PingCAP, China
sunyishen@pingcap.com

Shuai Zhou
PingCAP, China
zhoushuai@pingcap.com

Haihuang Su
PingCAP, China
suhaihuang@pingcap.com

Liu Tang
PingCAP, China
tl@pingcap.com

Qi Liu
PingCAP, China
liuqi@pingcap.com

ABSTRACT

With the growing significance of data analysis, several studies aim to provide precise answers to users' natural language questions from tables, a task referred to as tabular question answering (TQA). The state-of-the-art TQA approaches are limited to handling only single-table questions. However, real-world TQA problems are inherently complex and frequently involve multiple tables, which poses challenges in directly extending single-table TQA designs to handle multiple tables, primarily due to the limited extensibility of the majority of single-table TQA methods.

This paper proposes AutoTQA, a novel **Auto**nomous **T**abular **Q**uestion **A**nswering framework that employs multi-agent large language models (LLMs) across multiple tables from various systems (e.g., TiDB, BigQuery). AutoTQA comprises five agents: the *User*, responsible for receiving the user's natural language inquiry; the *Planner*, tasked with creating an execution plan for the user's inquiry; the *Engineer*, responsible for executing the plan step-by-step; the *Executor*, provides various execution environments (e.g., text-to-SQL) to fulfill specific tasks assigned by the *Engineer*; and the *Critic*, responsible for judging whether to complete the user's natural language inquiry and identifying gaps between the current results and initial tasks. To facilitate the interaction between different agents, we have also devised agent scheduling algorithms. Furthermore, we have developed LinguFlow, an open-source, low-code visual programming tool, to quickly build and debug LLM-based applications, and to accelerate the creation of various external tools and execution environments. We also implemented a series of data connectors, which allows AutoTQA to access various tables from multiple systems. Extensive experiments show that AutoTQA delivers outstanding performance on four representative datasets.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685816

PVLDB Reference Format:
Jun-Peng Zhu, Peng Cai, Kai Xu, Li Li, Yishen Sun, Shuai Zhou, Haihuang Su, Liu Tang, Qi Liu. AutoTQA: Towards Autonomous Tabular Question Answering through Multi-Agent Large Language Models. PVLDB, 17(12): 3920 - 3933, 2024.
doi:10.14778/3685800.3685816

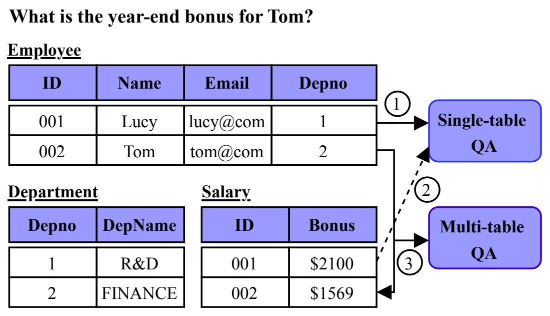
1 INTRODUCTION

Tabular Question Answering (TQA) [4, 7, 8, 12, 16, 20, 24, 27, 52–54, 59] is a crucial task in data analysis, focusing on providing answers from tables in response to a user's natural language (NL) inquiry. Data are commonly presented in tabular form in scenarios such as financial reports and statistical reports. Users often need expertise to handle and address questions in such scenarios effectively. TQA approaches address these issues without demanding extensive expertise in natural language processing and data analysis. In a real scenario, however, tables have complex forms, such as relational database (RDB) tables (i.e., structured tables) and web tables. Furthermore, TQA tasks can encompass the manipulation of multiple tables, including operations such as joins, set operations, and others. The ongoing exploration of a unified TQA framework applicable to various scenarios remains a prominent research focus within data analysis and natural language processing.

The state-of-the-art large language models (LLMs) such as ChatGPT [25], BLOOM [36] and LLaMA [39] have experienced rapid development, to achieve general artificial intelligence, showcasing remarkable zero-shot [44] capabilities in a variety of linguistic applications. However, while these LLMs demonstrate excellence in general knowledge, their performance within specific domains, such as data analysis, may yield somewhat erroneous answers and should not be entirely relied upon due to a lack of domain-specific training [60]. The emergence of LLMs presents new opportunities and challenges for the tasks associated with TQA.

Limitations of Prior Art. As far as we know, state-of-the-art TQA approaches have their limitations:

(1) **Lack of effective solutions in multi-table TQA.** Current TQA approaches [8, 16, 24, 52, 54] are typically limited to addressing single-table TQA tasks. In this context, users send a natural language inquiry, and these approaches can only analyze data from


What is the year-end bonus for Tom?

Employee

ID	Name	Email	Depno
001	Lucy	lucy@com	1
002	Tom	tom@com	2

Department

Depno	DepName
1	R&D
2	FINANCE

Salary

ID	Bonus
001	$2100
002	$1569

Figure 1: A Motivating Example for Multiple Tables TQA.

a single table specified by a user to provide answers. Typically, this process involves employing Python or SQL for data manipulation on a single table. However, a common scenario arises where, for example, for convenience of management, the company stores basic employee information in the *Employee* table. Simultaneously, the finance department establishes a *Salary* table for effective salary management and security, housing employee IDs and the corresponding salary details. When sending an inquiry such as "What is the year-end bonus for *Tom*?", it is required to perform a table join operation, joining the *Employee* and *Salary* tables using the employee ID as the join key. As depicted in Figure 1, in this instance, the *Employee* table and the *Salary* table individually (① or ②) cannot answer the user's inquiry. An operation of join (③) between the two tables is required. In addition to executing a table join, the task may require complex analyses, such as set operations. It may even be necessary to perform multiple complex operations in a specific order (or steps) on multiple tables. MultiTabQA [27] is a method that is able to address TQA tasks in multiple tables. Nevertheless, it requires elaborate pre-training and fine-tuning processes, posing significant complexity for the PingCAP. The task of multiple tables TQA has not yet received exhaustive research attention, and it continues to exhibit the following two limitations.

(2) **Lack of adequate exploration of LLM capabilities.** The state-of-the-art ReAcTable [54], uses a combination of Chain of Thought (CoT) [45] and ReAct [51] paradigm to facilitate responses to user's questions. However, relying solely on these two in-context learning approaches is inadequate to fully explore the capabilities of LLM. In the context of ReAcTable, the challenge arises when encountering an empty answer at an intermediate step, as the method faces difficulty distinguishing between an incorrect response and a genuinely empty one. Through extensive experiments, we have identified numerous factors contributing to the occurrence of empty answers. Furthermore, it is crucial to note that encoding the data from the table directly into the ReAcTable prompt [54] is not practical, especially for large tables in business operations. Another crucial capability of LLMs is their ability to learn from conversations. LLMs represent a type of generative artificial intelligence from conversations, and existing TQA methods utilizing LLMs do not fully exploit this potential. Providing appropriate guidance to LLMs is likely to result in more accurate responses and improve overall efficiency.

(3) **Lack of solutions for manipulating multiple tables from multiple systems.** The state-of-the-art TQA approaches [7, 12, 16, 20, 24, 54, 59] face challenges when simultaneously dealing with structured database tables and semi-structured tables (such as web tables). These tables could be stored in MySQL, S3 object storage, or obtained in real time through APIs. These approaches require users to undergo an intricate preprocessing and extract, transform, and load (ETL) process, thereby significantly increasing the complexity of the TQA task. Large-scale ETL processes executed in batches typically require significant resources. Users of TQA should be freed from these complexities.

Key Technical Challenges. As mentioned above, we summarize the challenges in the design of the TQA approaches for the different scenarios:

- The first technical challenge is how to design and implement a unified TQA framework capable of addressing both single table and multiple tables TQA tasks.
- The second technical challenge is how to maximize the utilization of the LLM's capabilities to provide answers while concurrently verifying the correctness of those answers.
- The third technical challenge is how to efficiently manage various tables from multiple systems.

Proposed Approach. To this end, we propose AutoTQA, an **Auto**nomous **T**abular **Q**uestion **A**nswering framework employing multi-agent large language models in this paper. To overcome the first two challenges, we introduce AutoTQA, designed to efficiently support the TQA tasks of single table and multiple tables. AutoTQA delves into multi-agent LLMs, involving five distinct roles: *User*, *Planner*, *Engineer*, *Executor*, and *Critic*. These agents collaborate to respond to user inquiries through conversations. The *User* formulates the question in natural language, the *Planner* generates a detailed execution plan that combines standard operation procedure (SOP) of business operations, the *Engineer* executes the plan step-by-step, and the *Executor* invokes the suitable LLM-based applications for data processing using Python or SQL. Throughout the task, the *Critic* intervenes to evaluate task completion and identify gaps in the plan and original tasks. Upon detecting gaps, the *Critic* instructs the *Planner* to revise the plan. To precisely schedule different agents to collaborate on tasks, we also propose agent scheduling algorithms. To address the third challenge, AutoTQA integrates a series of data connectors utilizing Trino [37, 40] to streamline access to data from multiple systems within a single query.

It is essential to note that when the *Engineer* is tasked with a specific sub-task, it does not execute the task directly, such as writing Python or SQL code. The *Engineer* only needs to describe the task and then delegate it to *Executor*. The *Executor* initializes with the LLM-based execution environment, adapting to various requirements (or tasks). AutoTQA decouples the *Engineer* from the specific execution process, making it easier to locate problems and improve accuracy in the LLM-based execution environment. Rapidly and efficiently building and debugging LLM-based applications presents challenges. To address this challenge, we developed and implemented LinguFlow, an open source, low code visual programming tool for the development and deployment of LLM-based applications. Serving as a low-code programming tool, LinguFlow helps application developers in the rapid construction, debugging, and deployment of LLM-based applications. Developers need only a fundamental grasp of LinguFlow blocks to organize business logic as a message flow using a directed acyclic graph (DAG). In addition, Linguflow integrates with the embedding service, significantly enhancing the performance of LLM-based applications by managing few-shot examples. This has proven to be highly effective in PingCAP's practice. LinguFlow has been released as an open source project [32].

AutoTQA aims to comprehensively explore the capabilities of LLMs, with a specific focus on TQA tasks, emphasizing their ability to learn from conversations and conversation programming.

Key Contributions. To summarize, this paper makes the following contributions:

- Our extensive experiments have demonstrated that the current TQA approaches fall short of achieving oracle TQA accuracy in complex and real business scenarios due to: lack of (1) an effective solution in multi-table TQA, (2) fully exploring LLM capabilities, and (3) a solution for handling various tables from multiple systems.
- To overcome all the technical challenges, we propose AutoTQA, a novel TQA framework based on multi-agent LLMs that supports both single table and multiple tables question-answering tasks. Through conversations between different agents, TQA tasks can be efficiently solved. We also propose agent scheduling algorithms to coordinate agents to enhance the accuracy of collaboration among diverse agents. Furthermore, we integrated a series of data connectors into the AutoTQA framework, enabling AutoTQA to streamline accessing data from multiple systems within a single query.
- We have designed and implemented LinguFlow, an open source, low-code visual programming tool, to rapidly develop, debug, and deploy LLM-based applications.
- Extensive experiments demonstrate that AutoTQA achieves outstanding accuracy across four representative datasets. AutoTQA has been deployed in the production environment at PingCAP.

Organization. The rest of the paper is organized as follows. We provide the preliminaries in Section 2. The overall architecture of AutoTQA and execution workflow are discussed in Section 3. The detailed implementation of AutoTQA is presented in Section 4. Experimental evaluation is discussed in Section 5. We review related work in Section 6 and conclude in Section 7.

2 PRELIMINARIES

2.1 Tabular Question Answering

Tabular Question Answering (TQA) is a significant research area in natural language processing and data analysis of the database community. Typically, a data analyst or user explores a series of tables $D = \{t_1, t_2, \ldots, t_n\}$, where t_i represents a table stored in arbitrary storage. Subsequently, the data analyst needs to perform a series of data transformation operations $T = (dt_1, dt_2, \ldots, dt_n)$ on specific tables, where dt_i denotes a data transformation operation. Following each data transformation operation, the data analyst needs to decide whether to proceed to the next data transformation operation or whether the current result resolves the user's inquiry. If it does, the exploration process is halted. During the data transformation process, data analysts frequently utilize data processing tools such as Python or SQL to process data, requiring proficiency in Python or SQL programming.

For example, as illustrated in Figure 1, when aiming to answer the user's inquiry "*What is the year-end bonus for Tom?*", the data analyst initially identifies that the relevant answers must be extracted from the *Employee* and *Salary* tables. Subsequently, the data analyst performs a *join* data transformation operation on these two tables, specifically, join(*Employee*, *Salary*), and then assesses whether the result can address the question. In more complex TQA scenarios, additional intricate data transformation operations may be required, including *set operations*, *nested queries*, and *regex*. Even in complex TQA tasks, multiple rounds often need to be performed to get the correct answer. In the present scenario, the user's question can be resolved by joining two tables, leading to the termination of the exploration.

The final output of the TQA task is varied. It can directly present the result extracted from the table to the user, either as a value (i.e., a table with a single row and a single column) or a tabular (i.e., a table with multiple rows and multiple columns). Alternatively, it can extract the answer from the table, comprehend the result, and deliver it to the user in natural language. There are also predefined answer formats that enable the data analyst to fill in specific sections according to a specified format and deliver them to the user. Specifically, for the industrial dataset used in this paper, the combination of natural language and specific output formats enhances the clarity of answers.

2.2 Large Language Models

LLMs. Large language models (LLMs) are sophisticated artificial intelligence systems trained on vast amounts of textual data to understand and generate human-like language. Many LLMs [25, 26, 36, 39, 41] are based on deep neural networks. One key element is the attention mechanism [42], which allows the model to focus on different parts of the input text when generating the output. The architecture involves multiple layers of interconnected neurons, and the training process involves adjusting the weights and biases of these connections to minimize the difference between the predicted values and actual outputs. The attention mechanism can be mathematically [58] expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q, K, and V represent the input query, key, and value vectors, respectively. The softmax function normalizes the attention scores, and d_k is the dimensionality of the key vectors. This mechanism allows the model to assign different weights to different parts of the input sequence, enhancing its ability to capture context and relationships in the data.

LLM prompting. Prompting is a versatile technique where specific instructions are given to LLMs by adding prompts to the input. This method allows for customized responses without the need for retraining the model, and the prompts can be crafted manually or automatically learned from the data. Common prompting techniques encompass zero-shot prompting [44], few-shot prompting [2], Chain-of-Thought (CoT) [45], ReAct [51], and others.

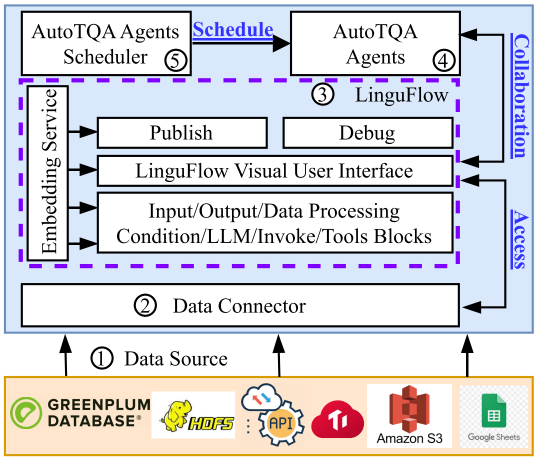


Figure 2: The Architecture of AutoTQA.

LLM agent. LLM agents [19, 47, 50, 57] are applications that perform complex tasks by combining LLMs with key modules such as planning and memory. These modules enhance the agent's capabilities in terms of task execution and responsiveness. The LLMs serve as the primary controller or "brain" of the agent, which controls the flow of operations required to complete a task or respond to user requests. In summary, LLM agents are applications that integrate LLMs with strategic components such as planning and memory to execute complex tasks.

3 THE OVERVIEW OF AUTOTQA

In this section, we first present the overview architecture of AutoTQA. Then, the execution workflow of AutoTQA is introduced.

3.1 The Overall Architecture of AutoTQA

The overall architecture of AutoTQA is shown in Figure 2. Next, we detail the core components of the architecture.

① **Data Source**. In AutoTQA, there are no restrictions on the source of the table data, whether it is stored in distributed databases like TiDB[13] or Greenplum [22], in a distributed file system like HDFS [38], or in a spreadsheet form like Google Sheets [10]. In PingCAP's practice, specific tables are stored in BigQuery [9] and TiDB [13], while some real-time data are accessible directly through APIs. Migrating all data into a unified system, such as TiDB serverless [31], would require extensive ETL efforts, be time-consuming, and impractical. The AutoTQA preserves the original location of all table data and exposes metadata from various table data sources through data connectors. These data connectors implemented with Trino abstract the complexity of interacting with multiple systems.

② **Data Connector**. The data connector serves as a vital component that enables the integration of diverse data sources, acting as an interlayer between an LLM-based application (or execution environment) and the data source. These data connectors are purposefully designed to enable seamless interaction with various data sources, including TiDB, PostgreSQL, local files, Google Sheets, Apache Thrift, and APIs. Their core function lies in abstracting the intricacies of interfacing with different data storage systems, simplifying the querying process. By providing a unified interface, data connectors empower users to effortlessly manipulate and combine data from disparate sources, fostering enhanced data interoperability and expanding the capabilities of analytics and processing across heterogeneous environments.

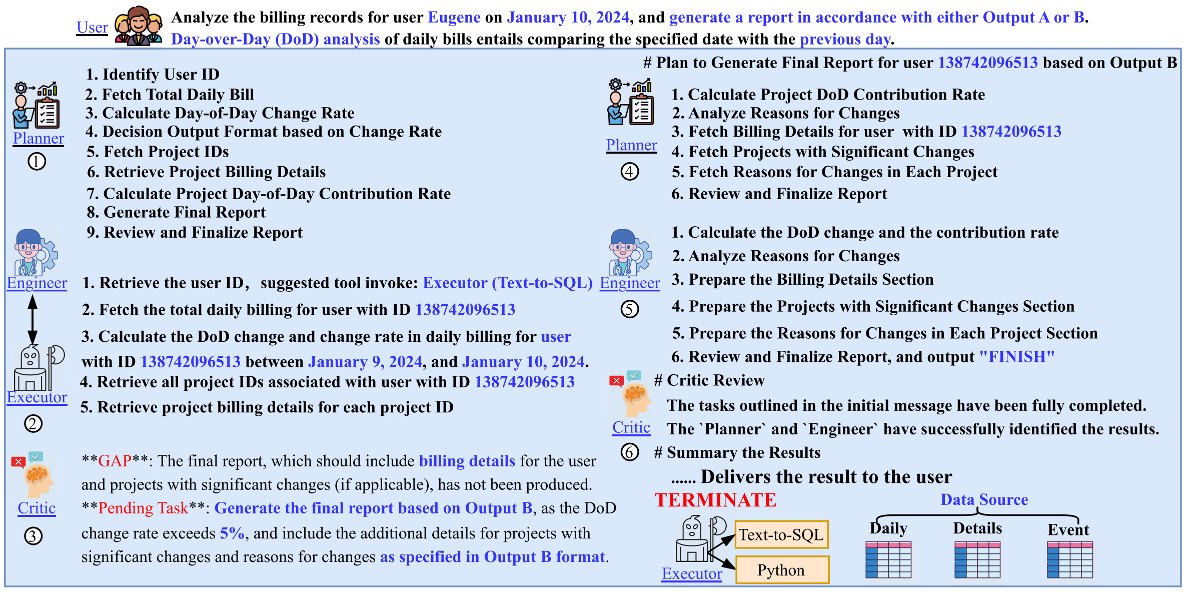
③ **LinguFlow**. AutoTQA requires the ReAct paradigm to interact with external tools (for example, text-to-SQL) based on LLM to complete tasks assigned by *Engineer*. Consequently, the rapid development and debugging of external tools emerge as the primary research challenge in this paper. Building upon this, we introduce LinguFlow, an open source visual low-code tool designed to streamline the rapid development, debugging, and deployment of LLM-based applications. The essence of LinguFlow lies in its low code architecture, enabling developers to construct business logic through a message flow based on DAG. In essence, LinguFlow empowers developers by providing a user-friendly environment where the focus is on the application's business logic, reducing the barriers to entry for LLM-based application development. Within LinguFlow, we incorporate an embedding service to enhance the performance of LLM-based applications. This service plays a pivotal role in addressing the specific needs of LLM-based applications that benefit from a few-shot prompting techniques to improve accuracy and correctness. The implementation details are provided in Section 4.2.

④ **AutoTQA Agents**. The LLM agents play crucial roles in handling specific responsibilities within the context of TQA. This ensemble comprises five distinct agents, namely *User*, *Planner*, *Engineer*, *Executor*, and *Critic*. Each agent is assigned a unique role in the TQA tasks, and their collaboration is crucial for completing TQA tasks. We refer to them as a *digital employee group*. Section 4.3 provides more comprehensive details about the specific functions and roles of these agents, providing a detailed insight into how these components collaboratively contribute to the overall effectiveness of AutoTQA.

⑤ **AutoTQA Agents Scheduler**. The effectiveness of the AutoTQA in TQA tasks is notably influenced by the scheduling of different agents for specific TQA tasks. Within agent interactions, a crucial control mechanism is implemented by setting a maximum round limit, terminating the interaction when the number of rounds reaches the specified threshold. To further optimize this process and achieve efficient interaction between agents, we introduce three schedulers: the circular rolling-based scheduler, the LLM-based scheduler based on the agent function description, and the finite-state machine-based scheduler. Each scheduler brings a unique approach to task allocation, and their performance variations have direct implications on both task accuracy and cost efficiency within the TQA framework. The implementation details are provided in Section 4.3.

3.2 AutoTQA Execution Workflow

In this section, we illustrate how AutoTQA provides the execution plan through *Planner* and *Engineer* in response to the user's questions. AutoTQA interacts with external LLM-based applications through *Executor*, while *Critic* is responsible for evaluating the complete completion of the task.



Analyze the billing records for user Eugene on January 10, 2024, and generate a report in accordance with either Output A or B.
Day-over-Day (DoD) analysis of daily bills entails comparing the specified date with the previous day.

1. Identify User ID
2. Fetch Total Daily Bill
3. Calculate Day-of-Day Change Rate
4. Decision Output Format based on Change Rate
5. Fetch Project IDs
6. Retrieve Project Billing Details
7. Calculate Project Day-of-Day Contribution Rate
8. Generate Final Report
9. Review and Finalize Report

1. Retrieve the user ID，suggested tool invoke: Executor (Text-to-SQL)
2. Fetch the total daily billing for user with ID 138742096513
3. Calculate the DoD change and change rate in daily billing for user with ID 138742096513 between January 9, 2024, and January 10, 2024.
4. Retrieve all project IDs associated with user with ID 138742096513
5. Retrieve project billing details for each project ID

GAP: The final report, which should include billing details for the user and projects with significant changes (if applicable), has not been produced.
Pending Task: Generate the final report based on Output B, as the DoD change rate exceeds 5%, and include the additional details for projects with significant changes and reasons for changes as specified in Output B format.

# Plan to Generate Final Report for user 138742096513 based on Output B

1. Calculate Project DoD Contribution Rate
2. Analyze Reasons for Changes
3. Fetch Billing Details for user with ID 138742096513
4. Fetch Projects with Significant Changes
5. Fetch Reasons for Changes in Each Project
6. Review and Finalize Report

1. Calculate the DoD change and the contribution rate
2. Analyze Reasons for Changes
3. Prepare the Billing Details Section
4. Prepare the Projects with Significant Changes Section
5. Prepare the Reasons for Changes in Each Project Section
6. Review and Finalize Report, and output "FINISH"

# Critic Review
The tasks outlined in the initial message have been fully completed.
The `Planner` and `Engineer` have successfully identified the results.
# Summary the Results
...... Delivers the result to the user
TERMINATE

Figure 3: An example for the execution workflow of AutoTQA.

An example is depicted in Figure 3. The execution results of each step plan are not shown in the figure, and the formats of Output A and Output B are not provided. Firstly, the *User* submits the question: "Analyze the billing records for user *Eugene* on January 10, 2024" to AutoTQA. Subsequently, the *Planner* formulates an initial execution plan based on the user input, determining that this task comprises nine steps (①). Due to the complexity of PingCAP's business operations, to help the *Planner* better formulate the execution plan, the *User* agent incorporates some standard operating procedures (SOP) from PingCAP's business. The plan is then allocated to *Engineer* through the finite state machine-based scheduler embedded in AutoTQA. The *Engineer* executes the plan step by step, refining it by incorporating additional details and specifying the external applications (i.e., tools) to be utilized. The formulated query is subsequently handed over to *Executor*. The *Executor* invokes the specified tool, carrying out the task allocated by *Engineer*. When specific code (SQL or Python) needs to be written or numerical calculations need to be performed, the *Engineer* assigns specific tasks to the *Executor*. The main reason is that AutoTQA wanted to decouple the functionality of the *Engineer* agent from specific external tools, allowing us to debug and enhance each part individually. In PingCAP practice, we found that many errors were generated by external tools rather than by the agent's design itself. To complete the plan, the *Engineer* and *Executor* engage in multiple rounds of interaction (②). Simultaneously, the scheduler directs the *Critic* to assess the completion status of the plan. If incomplete, the *Critic* identifies gaps and pending tasks (③) by understanding the execution context and the initial task. Subsequently, the *Planner* is dispatched to revise the plan, facilitating the accurate completion of the task of *User* (④). This iterative process repeats and the scheduler determines the appropriate agent (⑤). Finally, the *Critic* assesses task completion, summarizes the output for the user, and completes the entire process (⑥).

The completion of a task frequently involves numerous iterations, with AutoTQA specifying a maximum number of rounds. This value is chosen carefully in PingCAP's business as a trade-off between accuracy and cost. Furthermore, as interaction rounds increase, longer contexts amplify the likelihood of errors in context comprehension by the agent. In external application invokes, we leverage text-to-SQL, a crucial component that contributes to AutoTQA's accuracy in answering user inquiries. These tools are developed, debugged, and deployed using LinguFlow. The text-to-SQL tool interacts with the multiple tables using the data connector component.

4 IMPLEMENTATION

In this section, we begin with an overview of the data connector implementation. We then proceed to present the LinguFlow implementation, followed by a thorough exploration of the implementations of the agent and scheduler for AutoTQA. The section concludes with a detailed discussion of the exception handling and retry mechanism employed by AutoTQA.

4.1 Data Connector

To enable multiple table operations in TQA tasks, the pivotal component is the data connector. The primary purpose of a data connector is to register any external data source with Trino. In our implementation, to acquire real-time data updates, we directly extract data through APIs and register it into a unified interface, with Trino selected as the intermediate unified form. The interface implementations include (1) *ListSchemaNames* to register the database list; (2) *ListTables* to register the table list of the specified database; (3) *GetTableMetadata* to register specific metadata of the table, such as columns, their types, and index keys; (4) *GetSplit* to retrieve several splits meeting query requirements, with splits used for *GetRows* to obtain complete row data; (5) *GetRows* to retrieve all the row data based on a given split. For tables stored in other systems (or files), users of TQA tasks need only implement five interfaces to customize the data connector, greatly reducing the ETL effort. Implementing these five interfaces conceals the disparities among data sources for TQA tasks, allowing the convenient manipulation of diverse tables in a single query. In the text-to-SQL application of *Executor*, Trino SQL is generated to abstract the intricacies of the underlying system. In summary, a data connector aims to map all necessary table metadata into a unified Trino interface, facilitating simultaneous operations on different tables across multiple systems within a single query.

4.2 LinguFlow: A Low-Code Tool for Developing LLM-based Application

The direct users of LinguFlow are developers and application builders who leverage LLM and associated frameworks for data analysis and data processing. Previously, they had to navigate both business logic and understand LLM APIs and frameworks like Langchain [17]. The DAG is an abstraction of the message flow in LLM-based applications in LinguFlow. It represents business logic and consists of nodes and edges.

In LinguFlow, a node in the DAG represents an instantiated block and is a central element. These nodes serve to decompose intricate LLM-based applications into reusable blocks, allowing developers to seamlessly select and organize them according to business logic. Blocks that adhere to development specifications are registered in a registry. LLMs application developers can readily comprehend the input/output and function definitions, facilitating swift instantiation of these blocks as nodes within the DAG to implement LLMs applications.

LinguFlow comprises six block types: input/output defining for LLMs applications, data_process for common data processing operations like text merging, condition for common condition comparisons, LLMs, invoke other applications and components, and tools (e.g., Google Search). This capability proves valuable in scenarios, where interaction with external environments is required using ReAct paradigm. Applications built with LinguFlow can be deployed on the cloud with one click using its publishing capability. Additionally, LinguFlow provides online run and debug functionality, allowing users to run and debug applications. Ultimately, AutoTQA can easily refer to cloud-based applications via URLs. The advent of LinguFlow has greatly improved the efficiency of application development and deployment, allowing users to focus on TQA tasks themselves. LinguFlow is extensively used in PingCAP's practices.

Embedding Service. In LinguFlow, we employ the widely embraced technique of in-context learning with few-shot examples to enhance the accuracy of LLMs for specific question-answering tasks. This method entails presenting the LLMs with task-related examples or demonstrations, enabling them to capture context-specific and enhance performance. To achieve this goal, we develop a dedicated embedding service that isolates the corpus between different scenarios using *namespace*. We incorporate new examples or idempotently update existing data in the corpus through the use of *upsert* operation. The *query* operation enables the retrieval of top-k related data based on embedding, and *delete* operation removes data from the corpus using unique IDs. Our embedding service

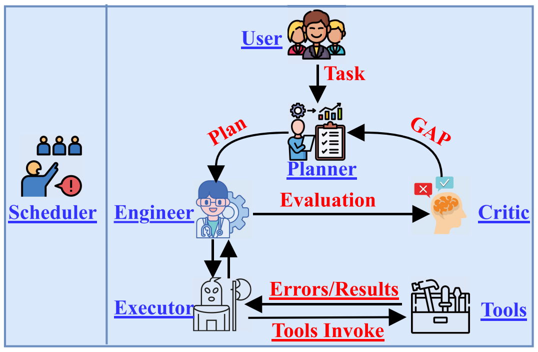
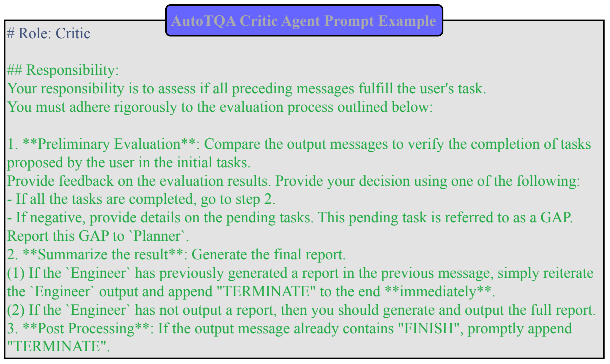


Figure 4: The agents of AutoTQA.


AutoTQA Critic Agent Prompt Example

```
# Role: Critic

## Responsibility:
Your responsibility is to assess if all preceding messages fulfill the user's task.
You must adhere rigorously to the evaluation process outlined below:

1. **Preliminary Evaluation**: Compare the output messages to verify the completion of tasks
proposed by the user in the initial tasks.
Provide feedback on the evaluation results. Provide your decision using one of the following:
- If all the tasks are completed, go to step 2.
- If negative, provide details on the pending tasks. This pending task is referred to as a GAP.
Report this GAP to `Planner`.
2. **Summarize the result**: Generate the final report.
(1) If the `Engineer` has previously generated a report in the previous message, simply reiterate
the `Engineer` output and append "TERMINATE" to the end **immediately**.
(2) If the `Engineer` has not output a report, then you should generate and output the full report.
3. **Post Processing**: If the output message already contains "FINISH", promptly append
"TERMINATE".
```

Figure 5: The prompt example of AutoTQA Critic agent.

supports Pinecone [30] and Qdrant [33] as both a vector database and a vector search engine. All are available as plugins, making it easy to add additional support for vector databases. Few-shot learning is a commonly employed optimization technique that maximizes the LLM's capabilities without necessitating fine-tuning. The embedding service also provides the flexibility to select few-shot examples.

4.3 AutoTQA Agent and Scheduler

The agents serve as the central component of AutoTQA, ensuring accurate responses to user questions. Various agents are assigned specific tasks based on their functional responsibilities, collaborating to answer user questions. Figure 4 illustrates the agents and their interactions. The AutoTQA comprises five agents, each with distinct roles that interact collaboratively to fulfill the specified task. Initially, the *User* receives a task submitted in natural language, describing the user's question. Figure 3 illustrates that for particular tasks, users may require additional specific information (e.g., DoD analysis in Figure 3) when describing a question. Subsequently, AutoTQA encapsulates the user's task and submits it to the *Planner*. The *Planner* is responsible for decomposing tasks into fine-grained subtasks suitable for execution. The decomposing process may follow the standard operating procedure (SOP) of the task. The *Engineer* receives these fine-grained subtasks and proceeds with step-by-step implementation. The *Engineer* refines each subtask, incorporates necessary information by conversation context into the

Algorithm 1: AutoTQA Agent Circular Rolling Scheduling.

```
1 Function RollingScheduling(current_round):
      /* Initiates the circular rolling scheduling sequence. */
2     circular_scheduling_queue = {Planner, Engineer, Executor, Critic};
3     if !Task.isComplete() and current_round ≤ max_round then
4         next_agent = rolling_scheduling_queue(circular_scheduling_queue);
5         current_round++;
6     return next_agent;
```

Algorithm 2: AutoTQA Agent LLM Scheduling.

```
1 Function LLMScheduling(current_round):
2     if !Task.isComplete() and current_round ≤ max_round then
          /* The LLM automatically selects the most suitable
             agent based on the function description of each
             agent and conversation context. */
3         next_agent = llm(agent_list, agent_function_description, conversation_context);
4         current_round++;
5     return next_agent;
```

plan, and initiates execution. In cases that involve interaction with external applications (e.g., text-to-SQL) during execution, the *Engineer* delegates this interaction to the *Executor*. Upon completion of the execution, the *Executor* returns the results to the *Engineer*. The *Engineer* continues the next steps based on the returned context content. The occurrence of an error may indicate an incorrect condition for the task assigned to *Executor* by the *Engineer*, prompting modification attempts and subsequent retries. Alternatively, the scheduler may assign *Critic* to analyze the cause of the error and determine if the current result aligns with the expectations of initial tasks. Figure 4 illustrates that both *Executor* and *Critic* have the potential to be scheduled following the *Engineer*. Figure 5 presents a fragment of the prompting for *Critic*, delineating its particular responsibilities. In case *Critic* identifies gaps, it reports them to the *Planner* for plan revision. The process iterates repeatedly until the tasks specified by the user are completed. The scheduler schedules the agents of AutoTQA based on the defined scheduling algorithm.

In the initial design, we created only three agents: *User*, *Planner*, and *Engineer*. We simulated a minimal number of digital employees to perform TQA tasks. Unlike the current design, the *Engineer* needed to program directly using programming languages such as SQL or Python. Additionally, during this process, the *Engineer* may perform calculations. We found that this process was very error-prone and that it was difficult to determine the cause of the errors. Therefore, we first decouple the *Engineer* and *Executor* roles, making it easier to debug errors that occurred during the process. This decoupling also allowed AutoTQA to incorporate more execution environments, which ultimately led to the development of LinguFlow. In addition, we added a *Critic* agent to help judge the completion of tasks. This agent functions as a product manager or project manager within a project team, bridging the gap between the current result and the initial task to ensure timely task completion. This formed a digital employee group with five agents. These agents can work together to complete tasks and improve accuracy.

Scheduler. The scheduler of AutoTQA is responsible for scheduling among different agents. An effective scheduling strategy has the potential to significantly minimize costs and enhance the accuracy of TQA tasks. AutoTQA includes three schedulers: the circular rolling-based scheduler, the LLM-based scheduler, and the FSM-based scheduler.

Circular Rolling-based Scheduler. The Algorithm 1 introduces a circular rolling-based agent scheduler. With a static scheduling sequence, the scheduler assigns agents to perform their respective tasks in the given order. In Algorithm 1, a dedicated scheduling queue is generated (line 2). Excluding the *User* agent is crucial for the scheduler, given its responsibility for submitting user-specified questions. Hence, it remains ineligible for rescheduling until the ongoing task is complete. Subsequently, if the current task remains incomplete and that the round has not surpassed the maximum limit, the *next_agent* is chosen from the scheduling queue using a circular rolling mechanism (lines 3-5). Scheduling based on circular rolling does not align with our definition of each agent's responsibilities in numerous scenarios. Our experiments demonstrate that this scheduling algorithm cannot achieve the correct result within the number of rounds specified *max_round* in Section 5. This scheduling algorithm reveals that AutoTQA cannot rely on a static sequence to achieve correct agent scheduling.

LLM-based scheduler. The Algorithm 2 presents a scheduling strategy based on the description of agent functions. It is a dynamic scheduling algorithm that selects the agent to be scheduled based on the conversation context and agent function descriptions (line 3). LLM-based schedulers may, to some extent, generate scheduling sequences that are not in line with expectations. This is primarily due to a misunderstanding of the conversation context. The longer the conversation context, the higher the likelihood of scheduling errors. The insight from this scheduling algorithm is that AutoTQA requires the implementation of a dynamic agent loop in the scheduling process. On the other hand, while studying scheduling algorithms, we found that the scheduling relationships of agents can be naturally described by finite-state machines. This inspires us to implement the finite state machine-based (i.e., FSM-based) scheduling algorithm.

FSM-based Scheduler. The Algorithm 3 presents an agent scheduling approach based on a finite state machine (FSM). Each agent is treated as a state in the finite state machine. When a condition is satisfied, it triggers an *entry action* in FSM. Upon completion of the action, AutoTQA switches from one agent to another. Here, the responsibilities of each agent serve as transition conditions (or events). When the current task requires an agent with specific responsibilities, the scheduler is activated to schedule the agent and facilitate the state transition. This algorithm allowed us to decouple the agent scheduling information from the agent function design.

Algorithm 3: AutoTQA Agent Finite State Machine Scheduling.

```
Function FSMScheduling(Agent prev_agent, current_round):
    if !Task.isComplete() and current_round ≤ max_round then
        if prev_agent == 'User' then
            next_agent = 'Planner';
            current_round++;
        if prev_agent == 'Planner' then
            next_agent = 'Engineer';
            current_round++;
        if prev_agent == 'Engineer' then
            next_agent = 'Executor' | 'Critic';
            current_round++;
        if prev_agent == 'Executor' then
            next_agent = 'Engineer';
            current_round++;
        if prev_agent == 'Critic' then
            next_agent = 'Planner';
            current_round++;
    return next_agent;
```

The FSM-based scheduling algorithm reduces the cost of inferring the *next_agent* based on the conversation context. Based on the agent function designed by AutoTQA, establish predecessor and successor agent relationships, and define their transitions. This guarantees an efficient and accurate scheduling process. In this algorithm, AutoTQA does not give a fixed scheduling sequence but adapts it dynamically through interactions. If the current task remains incomplete and the maximum round limit is not surpassed, the scheduler determines the most suitable agent (*next_agent*) based on the current agent (*prev_agent*) and conversation context (lines 2-17).

The main distinction between LLM-based scheduling and FSM-based scheduler lies in the fact that FSM-based explicitly illustrates the scheduling relationships, whereas LLM-based scheduling relies on the LLM's comprehension of each agent's function descriptions and conversation context. In an LLM-based scheduler, there might be scheduling sequences that do not align with expectations. We believe that agent scheduling remains an important research topic in the design of more complex agent scenarios.

4.4 Discussion

Exception handling and retry mechanism. When the *Engineer* submits the SQL or Python programming tasks to the *Executor*, the *Executor* initializes the corresponding code execution environment. Specifically, SQL operations are distributed to the underlying system through the unified data connector interface. Within this process, two situations require special handling: (1) The *Executor* returns an error, which requires the next iteration to continue processing via the retry mechanism. This is likely attributed to issues such as the faulty recall table or column. (2) The *Executor* returns an empty value. An empty value does not necessarily indicate an empty result and might signify a conditional error. In the industrial dataset of PingCAP, case sensitivity often results in empty values being returned. We collect similar error information and provide it to the *Engineer* as common knowledge. The *Engineer* often refines the results through the retry mechanism. Throughout the experiments, this mechanism effectively improved the accuracy of industrial datasets at PingCAP.

5 EXPERIMENTAL EVALUATION

In this section, we empirically evaluate the design of AutoTQA using four representative benchmarks. In particular, we mainly focus on the following research questions (RQs):

- **RQ1:** How does the accuracy of AutoTQA compare to methods specifically designed for single table TQA tasks?
- **RQ2:** How does the accuracy of AutoTQA on multiple tables TQA tasks focused on the industrial dataset?
- **RQ3:** How do the various agent scheduling algorithms implemented by AutoTQA affect the accuracy of the TQA tasks?
- **RQ4:** What is the impact of the embedding service (few-shot prompting) on accuracy?

In the following, we answer **RQ1** in Section 5.2, **RQ2** in Section 5.3, **RQ3** in Section 5.4 and **RQ4** in Section 5.5.

5.1 Experimental Setup

5.1.1 LLMs. The prompt-based LLMs employed in our experiment include (1) GPT-4-0613 (referred to as GPT-4); (2) GPT-4-1106-preview (referred to as GPT-4-turbo); (3) GPT-3.5-turbo-16k (referred to as GPT-3.5-turbo). The temperature parameter is universally set to 0 to prioritize reproduction across all models. The maximum number of rounds (i.e., max_round) was configured to 50 in our experiment for AutoTQA. By default, AutoTQA utilizes the FSM-based agent scheduler.

5.1.2 Datasets. In our experimental setup, we utilize four datasets, including three public datasets: WikiTQ [29], FeTaQA [24], TabFact [46], and an industrial dataset obtained from PingCAP's business operations.

- **WiKiTQ:** WiKiTQ serves as a representative dataset for TQA tasks on a single table, comprising $18,496$ complex questions related to Wikipedia tables. The WiKiTQ dataset includes complex questions necessitating multi-step reasoning and involving various data operations, such as comparison, aggregation, and arithmetic computation.
- **FeTaQA:** The data is sourced from Wikipedia. Typically, the answer takes the form of a free-form response. The dataset contains $9,329$ question-answer pairs.
- **TabFact:** TabFact is a large-scale dataset of manually annotated statements related to Wikipedia tables. Their relations are classified as "true" or "false". TabFact is a dataset specifically created to evaluate language inference in structured data, which encompasses a combination of *reasoning skills*

in both symbolic and linguistic domains. We use the small test data set to evaluate all methods, which includes 1, 998 question-answer pairs [6, 54].

- **Industrial Dataset:** This industrial dataset comprises billing data from TiDB Cloud, PingCAP's database as a service (DBaaS) offering. It encompasses aggregated and detailed billing data, database event data, and operational status data from real-world scenarios. This dataset facilitates the detection of significant changes in a tenant's billing, enables drill-down analysis to identify the largest contributing unit to the change, and helps confirm whether the billing change resulted from a database operation or an abnormal SQL query. The dataset exceeds 300 GiB in size. Real-time data is acquired through the API and seamlessly integrated into the AutoTQA via the data connector. Substantial amounts of question-answering pairs data have been accumulated. The complexity of industrial dataset far exceeds that of the other datasets discussed in this paper. On average, each question in this dataset references 4 tables.

We carefully annotated 5 few-shot examples for the industrial dataset of PingCAP to evaluate AutoTQA and its variants.

5.1.3 Evaluation Metrics. We evaluate AutoTQA using the following metrics: (1) For WikiTQ, the official metric is denotation *accuracy*; (2) For FeTaQA, we employ the *ROUGE-1*, *ROUGE-2*, and *ROUGE-L* metrics; (3) For TabFact, the evaluation metric is *string matching* [54]; (4) For the Industrial Dataset, the standard metric is *result accuracy*. These generated results have been thoroughly reviewed by senior engineers at PingCAP. In this paper, we collectively refer to these metrics as *accuracy*.

5.1.4 Baseline Methods. We compare the implementation of **AutoTQA** with the following methods:

- **Codex [4]**. The Codex generates the final answer directly through the use of in-context learning.
- **Binder [8]**. The Binder is a neural-symbolic framework free from training that maps the task input to a program. It allows binding a unified API of LLM functionalities to a programming language (e.g., SQL, Python) to extend its grammar coverage, and thus tackle more diverse questions. It is a model based on LLMs and uses openAI Codex (code-davinci-002) as the default LLM.
- **Dater [52]**. Dater addresses TQA tasks by decomposition of tables. Dater suggests that tables might be too large for LLMs to argue about. It is also an LLM-based approach, utilizing GPT-3 Codex (code-davinci-002) as the language model.
- **ReAcTable [54]**. ReAcTable uses LLM to decompose the problem into multiple steps and generate code-based logical operations to process tabular data as required. The generated code is then executed by external code executors using the ReAct paradigm, and the resulting intermediate table is fed back into the LLMs to support subsequent reasoning steps. It uses Codex (code-davinci-002) as the default LLM.
- **ReAcTable (GPT-3.5-turbo)**. It is a variation of ReAcTable and employs GPT-3.5-turbo as the default base LLM model. The voting mechanism was not used.

Table 1: Performance of Baselines on WikiTQ Dataset.

Method	Accuracy
Codex	47.6%
Binder	61.9%
Dater	65.9%
ReAcTable	68.0%
ReAcTable (GPT-3.5-turbo)	52.4%
ReAcTable (GPT-4)	67.3%
ReAcTable (GPT-4-turbo)	66.1%
AutoTQA (GPT-3.5-turbo)	66.0%
AutoTQA (GPT-4)	**75.3%**
AutoTQA (GPT-4-turbo)	73.2%

Table 2: Performance of Baselines on FeTaQA Dataset.

Method	ROUGE-1	ROUGE-2	ROUGE-L
Codex	0.62	0.40	0.52
Dater	0.66	0.45	0.56
ReAcTable	0.71	0.46	0.61
AutoTQA (GPT-3.5-turbo)	0.71	0.44	0.60
AutoTQA (GPT-4)	0.75	0.51	0.64
AutoTQA (GPT-4-turbo)	**0.77**	**0.53**	**0.67**

- **ReAcTable (GPT-4)**. This is a variant of ReAcTable that utilizes GPT-4 as its default base LLM model. The voting mechanism was not utilized.
- **ReAcTable (GPT-4-turbo)**. It is a variation of ReAcTable that uses GPT-4-turbo as its default base LLM model. The voting mechanism was not utilized.
- **AutoTQA (GPT-3.5-turbo)**. We introduce the AutoTQA method, employing the GPT-3.5-turbo model. Specifically, our choice is GPT-3.5-turbo-16k, taking into account the token limit of 16, 385 for this model.
- **AutoTQA (GPT-4)**. We present the AutoTQA method, utilizing the GPT-4 model. Specifically, we opt for GPT-4-0613, considering the token limit of 8, 192 applicable to this model.
- **AutoTQA (GPT-4-turbo)**. We present the AutoTQA method, utilizing the GPT-4-turbo model. Specifically, we opt for the GPT-4-1106-preview, considering the token limit of 128, 000 for this model.

Due to variations in benchmark performance, we report the benchmark-specific best-performing baselines. Specifically, we exclusively report LLM-based approaches since, as indicated in [52, 54], these approaches have achieved state-of-the-art performance.

5.2 Single Table Evaluation (RQ1)

In this section, we evaluate the performance of various methods for TQA tasks of a single table across WiKiTQ, FeTaQA, and TabFact benchmark datasets. The results are presented in Tables 1, 2, and 3, respectively. It is important to note that AutoTQA using different base models all utilize FSM-based scheduler. The impact of different scheduling algorithms on accuracy is discussed in Section 5.4.

Table 3: **Performance of Baselines on TabFact Dataset.**

Method	Accuracy
Codex	72.6%
Binder	85.1%
Dater	85.6%
ReAcTable	86.1%
ReAcTable (GPT-3.5-turbo)	73.1%
ReAcTable (GPT-4)	83.4%
ReAcTable (GPT-4-turbo)	85.0%
AutoTQA (GPT-3.5-turbo)	79.4%
AutoTQA (GPT-4)	87.4%
AutoTQA (GPT-4-turbo)	**88.7%**

In Table 1, when AutoTQA uses GPT-3.5-turbo as the default model, the performance is slightly worse than that of ReAcTable, with a performance gap of 2%. However, when AutoTQA uses GPT-4-turbo as the base model, the performance improves by 5.2% compared to ReAcTable, and when GPT-4 is used as the base model, the performance improves by 7.3%. This observation suggests that opting for GPT-4 or GPT-4-turbo as the default LLM improves overall performance than GPT-3.5-turbo. This is due to (1) the superior inference performance of GPT-4 and GPT-4-turbo, and (2) the enhanced design of AutoTQA's agent, such as scheduling and exception-handling and retry mechanisms, which further improve accuracy. The full exploration of cooperative abilities among different agents is facilitated when equipped with GPT-4 or GPT-4-turbo models, given their inherently strong reasoning capabilities from conversation context. Furthermore, there exists a slight performance discrepancy between AutoTQA (GPT-4) and AutoTQA (GPT-4-turbo).

In Table 2, we observe that the performance of ReAcTable is similar to that of AutoTQA (GPT-3.5-turbo). AutoTQA thoroughly explores the conversational capabilities inherent in LLMs and their proficiency in acquiring reasoning skills from diverse conversational contexts. Both AutoTQA (GPT-4) and AutoTQA (GPT-4-turbo) outperform ReAcTable. The rationales align closely with those for using the results of the WiKiTQ dataset.

The data presented in Table 1 and Table 2 indicates that AutoTQA can introduce a dynamic agent loop during problem-solving by agent scheduling, agent function and retry, leading to enhanced accuracy in results. This also underscores the superior conversation and conversation programming capabilities of AutoTQA (GPT-4) and AutoTQA (GPT-4-turbo).

In Table 3, AutoTQA (GPT-3.5-turbo) exhibits a performance 6.7% worse than ReAcTable, and its performance is also inferior to Binder and Dater. AutoTQA (GPT-3.5-turbo) struggles when faced with "true" or "false" questions, occasionally providing neutral answers. However, AutoTQA (GPT-4) and AutoTQA (GPT-4-turbo) outperform other methods. This highlights that combining the robust GPT-4-turbo model with the agent-based design of AutoTQA results in superior performance.

The utilization of GPT-3.5-turbo, GPT-4, and GPT-4-turbo models in ReAcTable is demonstrated in Tables 1 and 3. Nevertheless, ReAcTable fails to achieve state-of-the-art performance. In contrast, AutoTQA demonstrates superior performance on these datasets.

Table 4: Performance of Baselines on Industrial Dataset.

Method	Accuracy
AutoTQA (GPT-3.5-turbo)	45.0%
AutoTQA (GPT-4)	**85.0%**
AutoTQA (GPT-4-turbo)	80.0%

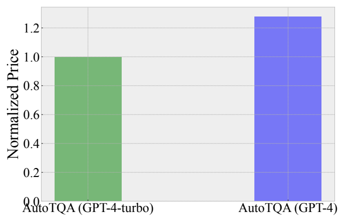


Figure 6: Normalized Average Price of Each TQA Task on Industrial Dataset.

This can be attributed to (1) AutoTQA formulates execution plans using the *Planner*, allowing for iterative updates. AutoTQA evaluates the completion of the plan by considering conversation context and identifying gaps, providing an advantage over ReAcTable. In error handling, AutoTQA may employ the *Critic* to assess errors and their causes from the conversation context; (2) AutoTQA's unique agent design and scheduling mechanism make better use of LLM's ability to reason and learn from conversation contexts. The design of AutoTQA maximizes the contextual learning capabilities of LLMs.

Finding 1. AutoTQA attains superior performance compared to other LLM-based approaches by leveraging the potent GPT-4 and GPT-4-turbo models. This is primarily due to the agent design (e.g., agent scheduling and retry mechanisms) of AutoTQA, which fully utilizes the LLM's ability to reason and learn from conversation contexts. Meanwhile, it also can harness powerful external applications.

5.3 Multiple Tables Evaluation (RQ2)

In this section, we evaluate the accuracy of the AutoTQA multiple tables TQA using the industrial dataset at PingCAP. Since Binder, Dater, and ReAcTable cannot be directly applied to multiple tables TQA tasks, our evaluation focuses solely on AutoTQA under different LLMs.

In Table 4, we present the experimental results on the industrial dataset collected by PingCAP. In this paper, we acquire the necessary data through the APIs, enabling real-time data injection through the data connectors developed by PingCAP. In Table 4, it is evident that the performance of AutoTQA (GPT-3.5-turbo) is 40% lower than that of AutoTQA (GPT-4). The performance difference can be explained by (1) the intricate structure and substantial volume of data in the industrial data set collected by PingCAP, together with the complex SOP for this task. The resulting complexity leads *Planner* to generate a highly complex plan, and AutoTQA (GPT-3.5-turbo) is more prone to errors in handling such complexity; (2) the superior reasoning ability of GPT-4 compared to GPT-3.5-turbo enhances performance in tackling complex plans; (3) the unique design of agents in AutoTQA, such as agent scheduling, further enhances accuracy. In the industrial dataset, where each question involves operations across multiple tables that encompass intricate joins, regularization, and others. Similarly, AutoTQA (GPT-4-turbo) shows only a performance decline of 5.0% compared to AutoTQA (GPT-4).

Finding 2. In real-world scenarios with demanding multi-table TQA tasks, AutoTQA demonstrates notable performance. This underscores AutoTQA's ability to achieve optimal performance across diverse TQA scenarios.

In the industrial dataset, we also provide the average normalized cost associated with each task. More precisely, the processing cost of AutoTQA (GPT-4) for a given task is 1.28× higher than that of AutoTQA (GPT-4-turbo) as shown in Figure 6. In industrial business operations, PingCAP considers the cost per task a key factor when selecting distinct base LLMs. Therefore, we prioritize the cost-effective choice of GPT-4-turbo as the base model of AutoTQA, as its accuracy is merely 5.0% inferior to that of AutoTQA (GPT-4). The senior engineers consider the accuracy of AutoTQA (GPT-4-turbo) in addressing TQA problems to be acceptable at PingCAP.

Finding 3. When handling intricate commercial TQA tasks, AutoTQA configured with the GPT-4-turbo base model can achieve accuracy levels that satisfy engineers. And it also offers a more cost-effective solution for businesses.

5.4 The Effect of Scheduling Algorithms on Accuracy (RQ3)

In this section, we utilize the industrial dataset to demonstrate that agent scheduling algorithms significantly impact accuracy. The performance effects of various scheduling algorithms in terms of accuracy are presented in Table 5.

In Table 5, AutoTQA (GPT-4) and AutoTQA (GPT-4-turbo), when configured with the circular rolling-based agent scheduling algorithm, do not provide correct answers to user questions within the *max_round* constraint. The circular rolling-based scheduling algorithm specifies a scheduling sequence. The erroneous generation of this scheduling sequence leads to incorrect messages being received by *next_agent* from the conversation contexts, resulting in misguided decisions by these agents. The subsequent agent captures these errors, amplifying the impact.

The performance comparison between LLM-based and FSM-based scheduling in Table 5 indicates only a difference of 5%. In practical business, AutoTQA is more likely to leverage both scheduling algorithms at PingCAP business operations. The LLM-based scheduling method involves the scheduler generating dynamic scheduling sequences based on each agent's function descriptions and conversation contexts. In contrast, FSM-based scheduling aligns more closely with the definition of different agent function descriptions. The primary reason for the lower accuracy of LLM-based scheduling compared to FSM-based scheduling is that, in some cases, the LLM-based scheduler schedule sequence is based on the function description of different agents, which may deviate from the user's intention. Consequently, errors arise in the agent's assessment of specific steps, and these inaccuracies propagate throughout the conversation contexts.

Finding 4. AutoTQA employs multi-agent LLMs, and their performance is contingent on the agent scheduling algorithm. Notably, the circular rolling-based scheduler exhibits the poorest performance. The FSM-based scheduling algorithm achieves the best performance.

Subsequently, we evaluate the number of rounds needed by AutoTQA with the LLM-based scheduler and the FSM-based scheduler to complete each task in the industrial dataset. The average rounds are presented in Figure 7. In the LLM-based scheduler, the number of rounds is 1.6× that of the FSM-based scheduler for the same question answering tasks. In conclusion, AutoTQA employing the FSM-based scheduler exhibits faster convergence.

Finding 5. AutoTQA employs the FSM-based scheduler to orchestrate agents, resulting in a reduction of scheduling rounds. This facilitates faster convergence, thereby significantly enhancing the user experience.

5.5 The Effect of Embedding Service on Accuracy (RQ4)

In this section, we evaluate the effect of the embedding service on enhancing the accuracy of AutoTQA (GPT-4-turbo). Due to the cost-effective attributes of GPT-4-turbo, our main focus in PingCAP business operations is to improve AutoTQA (GPT-4-turbo) accuracy. The experimental results are presented in Table 6.

AutoTQA (GPT-4-turbo) exhibited a substantial accuracy improvement with an increase in the number of few-shot examples from 0 to 5. This observation implies that few-shot examples significantly impact accuracy. We observe that with an increase in the number of few-shot examples, the increment in accuracy performance initially rises and subsequently declines. In our experiment with a substantial 300 GiB of industrial data, a mere 5 few-shot examples proved sufficient to boost the performance to 80.0%. This highlights the impact of few-shot examples on the performance of AutoTQA (GPT-4-turbo). Selecting the most effective few-shot examples to improve accuracy is very challenging. On the one hand, more few-shot examples are not necessarily better; on the other hand, different few-shot examples significantly impact accuracy.

Finding 6. The performance of AutoTQA in terms of accuracy depends on a few-shot examples. By moderately increasing the number of examples, AutoTQA performance can be improved.

6 RELATED WORKS

TQA. There are several state-of-the-art methods available for tackling single-table TQA tasks, broadly classified into approaches based

Table 5: Performance of AutoTQA on Industrial Dataset using Different Scheduling Algorithms.

Method	Scheduler	Accuracy
AutoTQA (GPT-4)	Circular Rolling-based	N/A
	LLM-based	80.0%
	FSM-based	**85.0%**
AutoTQA (GPT-4-turbo)	Circular Rolling-based	N/A
	LLM-based	75.0%
	FSM-based	**80.0%**

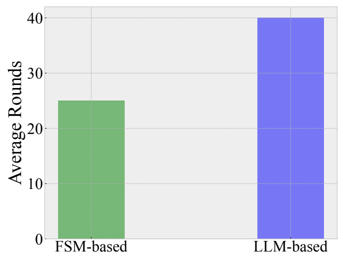


Figure 7: Average Rounds on the Industrial Dataset.

Table 6: Performance of Embedding Service on Industrial Dataset for AutoTQA (GPT-4-turbo).

Few-shot	Accuracy
0-shot	60.0%
1-shot	63.5%
2-shot	69.0%
3-shot	77.5%
4-shot	79.0%
5-shot	**80.0%**

on fine-tuning and those based on LLMs. **(1) Fine-tuning Approaches**. Table-BERT [6] employs a rule-based method to convert tabular data into natural language sentences. TaPas [20] improves the BERT architecture by integrating the capability to encode tables as part of its input. TAPEX [20] is designed to emulate the SQL executor within the BART model. SaMoE [55] incorporates the mixture-of-experts (MoE) paradigm into table-based fact verification. PASTA [11] introduces six types of sentence-table cloze tasks, pre-training on a synthesized corpus of 1.2 million items from WikiTables. TaCube [56] adopts a precomputation-based approach aimed at enhancing the performance of PLM in numerical reasoning. OmniTab [15] proposes an all-encompassing pre-training method that leverages both natural and synthetic data to bolster models with proficiencies in handling both types of data. MultiTabQA [27] is a multi-table TQA method that uses a single table TQA model *tapex-base* as the base model. **(2) LLM-based Approaches**. Binder [8] generates programs in programming languages and extends their capabilities to address common sense problems. Dater [52] tackles the TQA tasks through table decomposition. ReAcTable [54] uses LLMs to break down problems into multiple steps tailored to process tabular data as needed.

LLM agents. The work presented in [49] provides a survey of agents based on LLM. Voyager [43] is the LLM-powered embodied lifelong learning agent in Minecraft that continuously explores the world. Generative Agents [28] create credible simulations of human behavior, dynamically adapting to changing experiences and environments. BOLAA [21] orchestrates a multi-agent strategy, which enhances the action interaction ability of agents. AutoGen [47] serves as a generic framework for building diverse applications of various complexities and LLM capacities. AgentVerse [5] is a multi-agent framework that can effectively orchestrate a collaborative group of agents. ChatDev [34, 35] offers an easy-to-use, highly customizable, and extendable framework, which is based on LLMs and serves as an ideal scenario for studying collective intelligence. CAMEL [19] is a novel cooperative agent framework that allows communicative agents to collaborate autonomously. D-Bot [58] is an LLM-based database diagnosis system that can automatically acquire knowledge from diagnostic documents.
LLM-based visual programming tools. Low-code LLM [3] is a novel human-LLM interaction framework. LangFlow [18] is a dynamic graph where each node is an executable unit. Rivet [14] is a low code prompt chain that focuses on building complex flows. PromptFlow [23] is designed to streamline the end-to-end development cycle of LLM-based applications. PromptChainer [48] is an interactive interface for visually programming chains. ChainForge [1] is a visual toolkit for prompt engineering.

7 CONCLUSION

In this paper, we introduce AutoTQA, an autonomous tabular question answering framework that leverages multi-agent large language models. It comprises five agents collaborating to accomplish user-specified tasks. We also propose agent scheduling algorithms to orchestrate agents. Moreover, the utilization of the ReAct paradigm enables AutoTQA to interact with external applications. We introduce LinguFlow, an open source visual tool that facilitates swift development, debugging, and deployment of LLM-based applications. Extensive experiments demonstrate that AutoTQA achieves outstanding accuracy across four representative datasets. AutoTQA has been deployed in the production environment at PingCAP.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments and feedback. This work was supported in part by the National Natural Science Foundation of China under Grant No. U22B2020, and in part by PingCAP. Peng Cai and Kai Xu are the corresponding authors of this paper.

REFERENCES

[1] Ian Arawjo, Chelse Swoopes, Priyan Vaithilingam, Martin Wattenberg, and Elena Glassman. 2023. ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing. *arXiv preprint arXiv:2309.09128* (2023).

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *NIPS* 33 (2020), 1877–1901.

[3] Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu, Wang You, Ting Song, Yan Xia, et al. 2023. Low-code LLM: Visual Programming over LLMs. *arXiv preprint arXiv:2304.08103* (2023).

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[5] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *ICLR*.

[6] Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2019. Tabfact: A large-scale dataset for table-based fact verification. *arXiv preprint arXiv:1909.02164* (2019).

[7] Zhoujun Cheng, Haoyu Dong, Zhiruo Wang, Ran Jia, Jiaqi Guo, Yan Gao, Shi Han, Jian-Guang Lou, and Dongmei Zhang. 2022. HiTab: A Hierarchical Table Dataset for Question Answering and Natural Language Generation. In *ACL*. 1094–1110.

[8] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2022. Binding language models in symbolic languages. *arXiv preprint arXiv:2210.02875* (2022).

[9] Pavan Edara and Mosha Pasumansky. 2021. Big metadata: when metadata is big data. *PVLDB* 14, 12 (2021), 3083–3095.

[10] Google. 2023. Goole Sheets. Retrieved in Novermber, 2023 from https://www.google.com/sheets/about/.

[11] Zihui Gu, Ju Fan, Nan Tang, Preslav Nakov, Xiaoman Zhao, and Xiaoyong Du. 2022. PASTA: table-operations aware fact verification via sentence-table cloze pre-training. *arXiv preprint arXiv:2211.02816* (2022).

[12] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Mueller, Francesco Piccinno, and Julian Eisenschlos. 2020. TaPas: Weakly Supervised Table Parsing via Pre-training. In *ACL*. 4320–4333.

[13] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *PVLDB* 13, 12 (2020), 3072–3084.

[14] Ironclad. 2023. Rivet. Retrieved in Novermber, 2023 from https://rivet.ironcladapp.com/.

[15] Zhengbao Jiang, Yi Mao, Pengcheng He, Graham Neubig, and Weizhu Chen. 2022. OmniTab: Pretraining with natural and synthetic data for few-shot table-based question answering. *arXiv preprint arXiv:2207.03637* (2022).

[16] Nengzheng Jin, Joanna Siebert, Dongfang Li, and Qingcai Chen. 2022. A survey on table question answering: recent advances. In *China Conference on Knowledge Graph and Semantic Computing*. Springer, 174–186.

[17] LangChain. 2023. LangChain. Retrieved in Novermber, 2023 from https://www.langchain.com/.

[18] LangFlow. 2023. LangFlow. Retrieved in Novermber, 2023 from https://github.com/logspace-ai/langflow.

[19] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2024. Camel: Communicative agents for" mind" exploration of large language model society. *NIPS* 36 (2024).

[20] Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2021. TAPEX: Table Pre-training via Learning a Neural SQL Executor. In *ICLR*.

[21] Zhiwei Liu, Weiran Yao, Jianguo Zhang, Le Xue, Shelby Heinecke, Rithesh Murthy, Yihao Feng, Zeyuan Chen, Juan Carlos Niebles, Devansh Arpit, et al. 2023. Bolaa: Benchmarking and orchestrating llm-augmented autonomous agents. *arXiv preprint arXiv:2308.05960* (2023).

[22] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: a hybrid database for transactional and analytical workloads. In *SIGMOD*. 2530–2542.

[23] Microsoft. 2023. PromptFlow. Retrieved in Novermber, 2023 from https://github.com/microsoft/promptflow.

[24] Linyong Nan, Chiachun Hsieh, Ziming Mao, Xi Victoria Lin, Neha Verma, Rui Zhang, Wojciech Kryściński, Hailey Schoelkopf, Riley Kong, Xiangru Tang, et al. 2022. FeTaQA: Free-form table question answering. *ACL* 10 (2022), 35–49.

[25] OpenAI. 2022. Introducing ChatGPT. Retrieved in August, 2023 from https://openai.com/blog/chatgpt.

[26] OpenAI. 2023. GPT-4. Retrieved in Novermber, 2023 from https://openai.com/research/gpt-4.

[27] Vaishali Pal, Andrew Yates, Evangelos Kanoulas, and Maarten de Rijke. 2023. MultiTabQA: Generating Tabular Answers for Multi-Table Question Answering. *arXiv preprint arXiv:2305.12820* (2023).

[28] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *UIST*. 1–22.

[29] Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. *arXiv preprint arXiv:1508.00305* (2015).

[30] Pinecone. 2023. Pinecone. Retrieved in June, 2023 from https://www.pinecone.io/.

[31] PingCAP. 2023. TiDB Serverless. Retrieved in January, 2023 from https://www.pingcap.com/tidb-serverless/.

[32] PingCAP. 2024. LinguFlow. Retrieved in March, 2024 from https://github.com/pingcap/LinguFlow.

[33] Qdrant. 2024. Qdrant. Retrieved in January, 2024 from https://qdrant.tech/.

[34] Chen Qian, Xin Cong, Wei Liu, Cheng Yang, Weize Chen, Yusheng Su, Yufan Dang, Jiahao Li, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Communicative Agents for Software Development. arXiv:2307.07924 [cs.SE]

[35] Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2023. Experiential Co-Learning of Software-Developing Agents. arXiv:2312.17025 [cs.CL]

[36] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100* (2022).

[37] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *ICDE*. IEEE, 1802–1813.

[38] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *MSST*. Ieee, 1–10.

[39] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[40] trinodb. 2023. Trino. Retrieved in October, 2023 from https://trino.io/.

[41] Immanuel Trummer. 2023. Can Large Language Models Predict Data Correlations from Column Names? *PVLDB* 16, 13 (2023), 4310–4323.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[43] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291* (2023).

[44] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *NIPS* 35 (2022), 24824–24837.

[46] Jianshu Chen Yunkai Zhang Hong Wang Shiyang Li Xiyou Zhou Wenhu Chen, Hongmin Wang and William Yang Wang. 2020. TabFact : A Large-scale Dataset for Table-based Fact Verification. In *ICLR*. Addis Ababa, Ethiopia.

[47] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155* (2023).

[48] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. 2022. Promptchainer: Chaining large language model prompts through visual programming. In *CHI*. 1–10.

[49] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864* (2023).

[50] Tianbao Xie, Fan Zhou, Zhoujun Cheng, Peng Shi, Luoxuan Weng, Yitao Liu, Toh Jing Hua, Junning Zhao, Qian Liu, Che Liu, et al. 2023. Openagents: An open platform for language agents in the wild. *arXiv preprint arXiv:2310.10634* (2023).

[51] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).

[52] 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. *arXiv preprint arXiv:2301.13808* (2023).

[53] Xuanliang Zhang, Dingzirui Wang, Longxu Dou, Qingfu Zhu, and Wanxiang Che. 2024. A Survey of Table Reasoning with Large Language Models. *arXiv preprint arXiv:2402.08259* (2024).

[54] Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M Patel. 2024. ReAcTable: Enhancing ReAct for Table Question Answering. *PVLDB* 17(8) (2024), 1981–1994.

[55] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625* (2022).

[56] Fan Zhou, Mengkang Hu, Haoyu Dong, Zhoujun Cheng, Shi Han, and Dongmei Zhang. 2022. Tacube: Pre-computing data cubes for answering numerical-reasoning questions over tabular data. *arXiv preprint arXiv:2205.12682* (2022).

[57] Wangchunshu Zhou, Yuchen Eleanor Jiang, Long Li, Jialong Wu, Tiannan Wang, Shi Qiu, Jintian Zhang, Jing Chen, Ruipu Wu, Shuai Wang, et al. 2023. Agents: An open-source framework for autonomous language agents. *arXiv preprint arXiv:2309.07870* (2023).

[58] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2023. D-bot: Database diagnosis system using large language models. *arXiv preprint arXiv:2312.01454* (2023).

[59] Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. 2021. TAT-QA: A Question Answering Benchmark on a Hybrid of Tabular and Textual Content in Finance. In *ACL*. 3277–3287.

[60] Jun-Peng Zhu, Peng Cai, Boyan Niu, Zheming Ni, Kai Xu, Jiajun Huang, Jianwei Wan, Shengbo Ma, Bing Wang, Donghui Zhang, et al. 2024. Chat2Query: A Zero-Shot Automatic Exploratory Data Analysis System with Large Language Models. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 5429–5432.