MATA: Multi-Agent Framework for Reliable and Flexible Table Question Answering

Sieun Hyeon[1], Jusang Oh[2], Sunghwan Steve Cho[1], Jaeyoung Do[1,2*],
[1]Department of Electrical and Computer Engineering, Seoul National University,
[2]Interdisciplinary Program in Artificial Intelligence, Seoul National University
{zxc2692, dhwntkd412, steve97, jaeyoung.do}@snu.ac.kr

Abstract

Recent advances in Large Language Models (LLMs) have significantly improved table understanding tasks such as Table Question Answering (TableQA), yet challenges remain in ensuring reliability, scalability, and efficiency, especially in resource-constrained or privacy-sensitive environments. In this paper, we introduce MATA, a multi-agent TableQA framework that leverages multiple complementary reasoning paths and a set of tools built with small language models. MATA generates candidate answers through diverse reasoning styles for a given table and question, then refines or selects the optimal answer with the help of these tools. Furthermore, it incorporates an algorithm designed to minimize expensive LLM agent calls, enhancing overall efficiency. MATA maintains strong performance with small, open-source models and adapts easily across various LLM types. Extensive experiments on two benchmarks of varying difficulty with ten different LLMs demonstrate that MATA achieves state-of-the-art accuracy and highly efficient reasoning while avoiding excessive LLM inference. Our results highlight that careful orchestration of multiple reasoning pathways yields scalable and reliable TableQA. The code is available at https://github.com/AIDASLab/MATA.

1 Introduction

Tables serve as a foundational medium for representing structured data, central to data storage, organization, and analytics. Although advances in programming languages and database systems have improved the accessibility of tabular data (Wes McKinney, 2010; Chamberlin and Boyce, 1974), interacting with and interpreting tables remains a significant challenge for non-technical users who lack coding expertise. This challenge has motivated active research into table understanding tasks, particularly Table Question Answering (TableQA) (Pasupat and Liang, 2015; Lu et al., 2023b; Wu et al., 2025), which aims to enable natural language interaction with tables.

Recent progress in Large Language Models (LLMs) has dramatically expanded the possibilities for natural language interfaces to complex tabular information (Zhang et al., 2023; Wu et al., 2024; Fang et al., 2024), promising more accessible and user-friendly solutions for a broader audience. However, practical deployment of TableQA systems presents several persistent challenges that have not been fully addressed in prior work.

First, a critical limitation of prior TableQA methods is the lack of model-agnosticism. Most high-performing language models are closed-source and accessible only through paid APIs, which poses significant challenges in settings where privacy concerns, data ownership, or cost restrictions make the use of open-source models necessary. While previous studies (Wang et al., 2024; Zhang et al., 2024c; Lu et al., 2023a; Wang et al., 2025; Zhang et al., 2024a; Liu et al., 2024) have demonstrated strong performance with proprietary LLMs (OpenAI, 2024; Anthropic, 2025), it remains unclear whether comparable results can be achieved using open-source models, particularly those with smaller parameter sizes. In other words, the reliability of TableQA systems in such constrained and practical environments has yet to be thoroughly investigated.

Second, existing frameworks tend to rely on repeated LLM inferences to boost answer reliability, often incurring substantial computational costs. Techniques like Self-Consistency (Wang et al., 2023) or Best-of-N (Kang et al., 2025) have improved performance on complex tasks, but excessive LLM calls can lead to diminishing returns, higher expense, and, in some cases, even degrade accuracy due to over-prompting (Huang et al.,

*Corresponding author

Findings of the Association for Computational Linguistics: ACL 2026, pages 33436–33476

Prior Works	Model-Agnostic	LLM Call Optimization	Multi-Stage Verification	Multiple Metric Analysis	Multiple Reasoning Method
SynTQA (Zhang et al., 2024a)	✗	✗	✗	✗	CoT, SQL
Mix-SC (Liu et al., 2024)	✗	✗	✗	✗	CoT, Python
TabLaP (Wang et al., 2025)	✗	✗	✓	✗	CoT, Python
Chameleon (Lu et al., 2023a)	✗	✗	✗	✗	CoT, Python
ReAcTable (Zhang et al., 2024c)	✗	✗	✓	✓	Python, SQL
Chain-of-Table (Wang et al., 2024)	✗	✗	✗	✓	CoT, Python
MATA	✓	✓	✓	✓	CoT, Python, SQL

Table 1: Comparison with prior works. MATA uniquely addresses five key aspects of TableQA, unlike others.

2025; Chen et al., 2024a). An optimal TableQA framework must judiciously manage the number of LLM inferences, balancing accuracy with efficiency, which is a requirement that previous work has rarely addressed in depth.

Third, while LLMs support a range of reasoning strategies, including Chain-of-Thought (CoT) (Wei et al., 2022), Program-of-Thought (PoT) (Chen et al., 2023), and text-to-SQL (text2SQL) (Zhong et al., 2017), most existing TableQA frameworks fail to exploit this full diversity. Previous studies (Liu et al., 2024; Wang et al., 2025; Zhang et al., 2024a) have shown that text-based reasoning (e.g., CoT) may excel for ambiguous or intuitive queries, while code-based (e.g., PoT) or SQL-based (e.g., Text2SQL) approaches can provide greater precision in numerical reasoning. However, the relative advantage of each reasoning path depends heavily on the characteristics and training data of the underlying model (see Appendix A), and no single approach is universally optimal. A robust TableQA system should therefore generate and validate answers through multiple complementary reasoning paths to maximize reliability across different models and scenarios.

Incorporating all these considerations, in this paper, we propose a novel TableQA framework, MATA (Multi-Agent Framework for TableQA). MATA orchestrates diverse reasoning strategies using a coordinated set of LLM agents and lightweight tools. Its architecture includes mechanisms for multi-step answer verification and an efficient scheduling algorithm that minimizes unnecessary LLM calls, thereby enhancing both reliability and efficiency.

The key distinction of MATA is that reasoning diversity does not require a fixed inference budget. Instead of always executing all reasoning paths, MATA uses lightweight controllers to decide which CoT, PoT, and text2SQL branches are necessary and when verification can stop.

To rigorously evaluate the model-agnostic capabilities and practical effectiveness of MATA, we conduct comprehensive experiments with ten different language models, spanning both small with up to 10B parameters, and large models with more than 10B parameters. In contrast to prior work, we employ three complementary evaluation metrics (i.e., exact match, fuzzy matching, and token-level F1) to more accurately reflect the varied output formats of LLMs in TableQA tasks.

Notably, MATA achieves up to 40.1% improvement in exact match, 21.9% in fuzzy matching, and 33.1% in F1 score over the strongest baseline on a challenging benchmark, underscoring its superior effectiveness. Importantly, these gains are not simply a result of indiscriminately increasing or minimizing the number of LLM inference calls. Instead, MATA attains high accuracy by dynamically invoking an appropriate number of inference steps—striking a careful balance between reasoning diversity and computational efficiency, rather than blindly maximizing or minimizing LLM usage compared to other baselines. Our main contributions are as follows:

(1) We propose MATA, a model-agnostic multi-agent framework that integrates Chain-of-Thought, Program-of-Thought, and text-to-SQL for robust answer generation and verification.

(2) MATA demonstrates strong performance across three evaluation metrics on diverse open-source and proprietary LLMs, overcoming the reliance on large-scale models. Furthermore, it achieves both accuracy and efficiency through intelligent inference optimization.

2 Related Work

Earlier studies utilized Transformer-based architectures (Vaswani et al., 2017) and Pre-trained Language Models (PLMs). Concurrently, PLMs such as DeBERTaV3 (He et al., 2023), and others (Lewis et al., 2020; Raffel et al., 2020; Brown

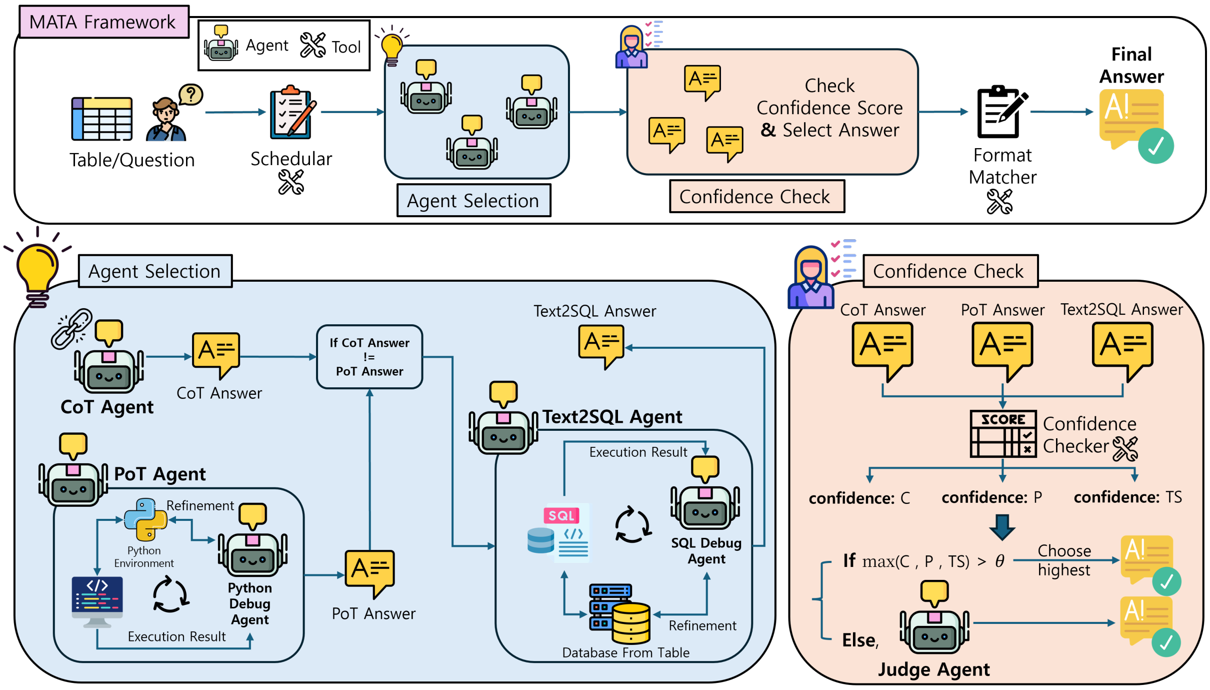

Figure 1: Overview of MATA. The current situation is when the Scheduler(*Sch*) selected PoT first. MATA integrates three complementary reasoning methods (CoT, PoT, text2SQL) through a multi-agent workflow. The Scheduler (*Sch*) prioritizes PoT or text2SQL reasoning based on the table and question, with CoT executed simultaneously. Candidate answers are evaluated by the Confidence Checker (*CC*); if no candidate meets the confidence threshold, the Judge Agent (*JA*) verifies the final answer. The Format Matcher (*FM*) ensures answers are concise.

et al., 2020) enhanced performance by fine-tuning on table-specific or SQL datasets (Patnaik et al., 2024; Zhang et al., 2024b; Li et al., 2024; Liu et al., 2022; Jiang et al., 2022; Zhao et al., 2022; Gu et al., 2022). Despite promising results, these methods typically underperform LLM-based approaches on out-of-distribution data, though their value in demonstrating PLM capabilities for tabular understanding remains significant.

To mitigate LLM limitations in numerical reasoning, recent methods adopted parallel reasoning paths (Liu et al., 2024; Zhang et al., 2024a; Wang et al., 2025), integrating text-based (e.g., CoT) and code-based reasoning (e.g., PoT, text2SQL). Final answers are selected from candidates generated via textual inference or executed code, often aided by self-consistency mechanisms, external LLM evaluators, or verification models (Liu et al., 2024; Zhang et al., 2024a; Ni et al., 2023). Other strategies explored table decomposition into sub-questions (Ye et al., 2023), iterative reasoning (Wang et al., 2024; Zhang et al., 2024c), and external tool integration (Lu et al., 2023a). Additionally, some methods involve generating and refining SQL or Python code using the same LLM to produce final answers (Cheng et al., 2023; Madaan et al., 2023). Nevertheless, many of these approaches heavily rely on proprietary LLMs, with limited evaluation using open-source models.

3 Methodology

3.1 Preliminary

In this section, we delineate the components and modules comprising our framework, MATA, while defining the notations employed throughout the paper. The modules within MATA are categorized into two groups: **Tools** and **Agents**.

Tools refer to small language models with parameter sizes under 500M. Given that all LLMs used in our experiments possess at least 3B parameters, these tools are markedly smaller in comparison, thereby incurring minimal inference overhead. MATA leverages three such tools.

The Scheduler (*Sch*) serves as the inaugural module, processing the input Table (T) and Question (Q). Based on several features extracted from T (e.g., table size, schema, data types) as well as the semantic meaning of the input Q, *Sch* determines whether to perform PoT or text2SQL reasoning first. It is implemented using a lightweight model combining MobileBERT (Sun et al., 2020) and a two-layer MLP, totaling 24.65M parameters.

The Confidence Checker (*CC*) module ingests *T*, *Q*, and the candidate answers derived from CoT, PoT, and text2SQL reasoning. It assigns a confidence score to each candidate. Prior research (Gu et al., 2022) has shown that DeBERTaV3 (He et al., 2023) achieves strong performance in table understanding. Accordingly, *CC* is built by fine-tuning DeBERTaV3-large, which has approximately 435M parameters. For more implementation details of the Scheduler and the Confidence Checker, refer to Appendix B.

The Format Matcher (*FM*) is the final tool in the pipeline. It converts long, verbose responses into concise entities. *FM* is implemented using the qwen2.5-instruct model with 0.5B (500M) parameters, used without any fine-tuning.

Agents denote LLM-based modules with at least 3B parameters. In our experiments, all agents share the same backbone LLM and differ only in their role-specific prompts; *Sch*, *CC*, and *FM* are separate lightweight tool models. MATA incorporates a total of six agents. Specifically, the CoT Agent (*CoTA*), PoT Agent (*PoTA*), and text2SQL Agent (*t2SA*) are responsible for performing CoT, PoT, and text2SQL reasoning, respectively. Owing to the inherent propensity for syntactic errors in code generation, *PoTA* and *t2SA* are supplemented by the Python Debug Agent (*PDA*) and SQL Debug Agent (*SDA*), which rectify outputs from their respective counterparts. Finally, the Judge Agent (*JA*) aggregates candidate answers from CoT, PoT, and text2SQL to adjudicate the ultimate response.

For brevity, subsequent sections will reference each module by its designated abbreviation.

3.2 Dataset for *Sch* and *CC*

We curate a large-scale training dataset [1] specifically designed to support the fine-tuning of lightweight tools like the *Sch* and *CC* in multi-reasoning TableQA frameworks. To construct it, we leverage three established public TableQA datasets—WikiTQ (Pasupat and Liang, 2015), TabMWP (Lu et al., 2023b), and TabFact (Chen et al., 2020)—and conduct inferences using three LLMs: phi4-14B (Abdin et al., 2024), Qwen2.5-Coder-14B (Hui et al., 2024), and CodeLLaMA-13B (Rozière et al., 2024). These inferences span three complementary reasoning modes: CoT, PoT, and text2SQL, yielding a total of 173,664 samples from 57,888 unique table-question (*T*, *Q*) pairs, with each pair generating outputs and correctness labels from all three reasoning modes.

[1] The dataset is publicly available at https://github.com/AIDASLab/MATA/tree/main/experiments, complete with documentation and licensing for free research use, to foster advancements in efficient and adaptable TableQA systems.

Algorithm 1 Agent Selection with Scheduler(*Sch*)

$sol_{cot}, A_{cot} \leftarrow CoTA(T, Q)$
if Use_Scheduler == True **then**
  prob_pot, prob_sql $\leftarrow Sch(T, Q)$
  if prob_pot >= prob_sql **then**
    $\mathcal{C}_{pot}, A_{pot} \leftarrow$ Code&Debug($PoTA, T, Q, N$)
    if $A_{cot} \neq$ Last Answer in $\mathcal{A}_{pot}$ **then**
      $\mathcal{C}_{sql}, A_{sql} \leftarrow$ Code&Debug($t2SA, T, Q, N$)
    else
      $\mathcal{C}_{sql}, A_{sql} \leftarrow \emptyset$
  else
    $\mathcal{C}_{sql}, A_{sql} \leftarrow$ Code&Debug($t2SA, T, Q, N$)
    if $A_{cot} \neq$ Last Answer in A_{sql} **then**
      $\mathcal{C}_{pot}, A_{pot} \leftarrow$ Code&Debug($PoTA, T, Q, N$)
    else
      $\mathcal{C}_{pot}, A_{pot} \leftarrow \emptyset$
else
  $\mathcal{C}_{pot}, A_{pot} \leftarrow$ Code&Debug($PoTA, T, Q, N$)
  $\mathcal{C}_{sql}, A_{sql} \leftarrow$ Code&Debug($t2SA, T, Q, N$)
return $sol_{cot}, A_{cot}, \mathcal{C}_{pot}, A_{pot}, \mathcal{C}_{sql}, A_{sql}$

Sch was trained by labeling each (*T*, *Q*) pair based on whether the PoT or text2SQL path produced the correct answer. *CC* was trained using labels indicating whether each of the CoT, PoT, and text2SQL paths yielded a correct answer.

3.3 MATA

MATA obtains the optimal answer *A* for the given table *T* and question *Q* through a 3-stage process. The overall workflow is illustrated in Figure 1, and the complete algorithm is provided in the Appendix. (See Algorithm 4).

Agent Selection with Scheduler(*Sch*) As illustrated in Algorithm 1, upon receiving an input table (*T*) and question (*Q*), MATA employs the Scheduler (*Sch*) to determine the execution priority between the PoT agent (*PoTA*) and the text2SQL Agent (*t2SA*). Concurrently, the CoT Agent (*CoTA*) conducts text-based reasoning to produce both the answer and the associated reasoning path. We denote the CoTA solution text as sol_{cot} and its extracted answer as A_{cot}; both are passed to *CC* and, when needed, *JA*. In contrast to *PoTA* and *t2SA*, *CoTA* forgoes additional refinement, as code-based reasoning benefits substantially from debugging, whereas text-based reasoning exhibits marginal improvements even with supplementary LLM inferences (refer to Appendix D). Consequently, *CoTA* is invoked only once to preserve computational

Algorithm 2 Code Generation&Debugging

```
function Code&Debug(Agent, T, Q, N)
    Initialize 𝒞, A ← ∅
    if Agent == PoTA then
        Debug ← PDA
    else
        Debug ← SDA
    code^0, A^0 ← Agent(T, Q)
    Append (code^0, A^0) to 𝒞, A
    for i = 0 to N − 1 do
        code^{i+1}, A^{i+1} ← Debug(T, Q, code^i, A^i)
        Append (code^{i+1}, A^{i+1}) to 𝒞, A
        if Stop_condition == True then
            break
    return 𝒞, A
```

efficiency.

Subsequently, the agent selected by *Sch*—either *PoTA* or *t2SA*—is executed. If its output aligns with the answer generated by *CoTA*, the remaining agent is omitted, and the final answer selection proceeds using only these two candidates.

If the user chooses not to use *Sch*, MATA executes both *PoTA* and *t2SA* without prioritization, thereby generating candidate answers from all three reasoning pathways. This approach, however, elevates the LLM inference overhead, presenting a inherent trade-off between comprehensiveness and efficiency.

Code Generation&Debugging As previously noted, the *CoTA* is invoked only once to ensure cost efficiency, whereas *PoTA* and *t2SA* undergo an additional debugging phase facilitated by their respective Debug Agents. Algorithm 2 delineates the code generation and debugging workflow.

The *PoTA* and *t2SA*, in conjunction with their corresponding Debug Agents—the Python Debug Agent (*PDA*) and SQL Debug Agent (*SDA*)—engage in a unified iterative process. Initially, *PoTA* and *t2SA* generate the requisite code, which is executed to yield either an output or error messages. The generated code, along with its execution results, is then forwarded to the pertinent Debug Agent, which refines and debugs the code. The revised code is re-executed to produce updated outputs or error messages. This iterative debugging loop continues for up to N cycles.

To curtail LLM inference expenses, an early termination criterion is implemented: the loop halts if the newly generated code exhibits substantial similarity to its predecessor and yields identical execution results. All iterations of the code and their associated outputs from this loop are subsequently conveyed to the final answer selection phase as candidate responses. The maximum number of debugging iterations N serves as a hyperparameter. Empirical evaluations indicate that $N = 3$ suffices for optimal performance (refer to Appendix D for hyperparameter optimization details).

Algorithm 3 Final Answer Decision

```
C, P, TS ← CC(T, Q, sol_cot, A_cot, 𝒞_pot, A_pot, 𝒞_sql, A_sql)
if max(C, P, TS) > θ then
    A_f ← arg max_{A∈{A_cot, A_pot, A_sql}} {C, P, TS}
else
    A_f ← JA(T, Q, sol_cot, A_cot, 𝒞_pot, A_pot, 𝒞_sql, A_sql)
if len(A_f) > 100 then
    A_f ← FM(A_f)
```

Final Answer Decision Algorithm 3 describes this process. The answer candidates generated through interactions among the agents are then passed to the final answer selection phase. In this step, the Confidence Checker (*CC*) computes confidence scores for each reasoning path and its corresponding answer. If the score of at least one candidate exceeds a predefined threshold θ, the system skips the additional inference step by the Judge Agent (*JA*) and selects the highest-scoring answer, thereby avoiding an unnecessary LLM agent call. We set θ to 0.1 through hyperparameter tuning (see Appendix D).

If none of the candidates exceed θ, the *JA* is invoked to determine the final answer. The *JA* may consider the confidence scores provided by *CC* or make its own judgment independently. Ultimately, the final answer is chosen through a two-step validation process involving both the *CC* and *JA*.

Additionally, we observed that in some cases, LLMs generate excessively verbose responses as final answers, which is undesirable in TableQA tasks where ground truths are typically short phrases or entities. This issue stems from the backbone LLMs' limited instruction-following ability and can degrade performance. To address this, we introduce a lightweight Format Matcher (*FM*)—a 500M-parameter model—that extracts concise entities from overly long responses when the final answer exceeds 100 characters. In such cases, *FM* extracts the entity and returns it as the final answer.

4 Experiments

4.1 Baselines and Benchmarks

To evaluate performance on open-source LLMs and new datasets not considered in prior research setups, we select three baselines whose official code

-		TabLaP			SynTQA			MixSC			**MATA**		
	Models	EM	fuzzy	F1	EM	fuzzy	F1	EM	fuzzy	F1	EM	fuzzy	F1
Small LLM	llama3.2-3b	0.188	0.290	0.247	0.597	0.654	0.602	0.201	0.303	0.252	**0.736**	**0.766**	**0.736**
	mistral-7b	0.049	0.231	0.102	0.639	0.680	0.645	0.271	0.385	0.289	**0.861**	**0.880**	**0.861**
	phi4-mini-3.8b	0.333	0.483	0.362	0.813	0.827	0.813	0.500	0.593	0.528	**0.819**	**0.847**	**0.819**
	qwen2.5-3b	0.396	0.479	0.400	0.694	0.737	0.694	0.438	0.517	0.442	**0.868**	**0.883**	**0.868**
	qwen2.5-7b	0.444	0.522	0.444	0.813	0.866	0.815	0.597	0.657	0.597	**0.951**	**0.955**	**0.951**
Large LLM	mistral-small-24b	0.764	0.784	0.773	**0.896**	**0.918**	**0.896**	0.806	0.813	0.810	**0.896**	0.896	**0.896**
	cogito-32b	**0.931**	**0.934**	**0.931**	0.868	0.886	0.868	0.903	0.908	0.903	0.903	0.903	0.903
	qwen2.5-32b	0.611	0.687	0.656	0.861	0.892	0.861	0.785	0.802	0.789	**0.917**	**0.917**	**0.917**
	GPT-4o	0.653	0.655	0.653	**0.951**	**0.961**	**0.951**	0.833	0.835	0.833	0.903	0.903	0.903
	Claude-3.7-Sonnet	0.868	0.868	0.868	**0.965**	**0.970**	**0.965**	0.924	0.924	0.924	0.951	0.951	0.951
	Average	0.524	0.593	0.544	0.810	0.839	0.811	0.626	0.674	0.637	**0.881**	**0.890**	**0.881**

Table 2: Evaluation results on the **Penguins in a Table** benchmark under our evaluation protocol. We report Exact Match(EM) accuracy, fuzzy matching, and F1 scores for each model. Bold indicates the best performance; underlined scores are the second best.

-		TabLaP			SynTQA			MixSC			**MATA**		
	Models	EM	fuzzy	F1	EM	fuzzy	F1	EM	fuzzy	F1	EM	fuzzy	F1
Small LLM	llama3.2-3b	0.067	0.357	0.130	0.089	0.231	0.120	0.081	0.372	0.144	**0.354**	**0.563**	**0.381**
	mistral-7b	0.036	0.331	0.119	0.227	0.367	0.270	0.082	0.355	0.151	**0.294**	**0.473**	**0.321**
	phi4-mini-3.8b	0.056	0.334	0.126	0.202	0.366	0.253	0.144	0.411	0.203	**0.273**	**0.457**	**0.295**
	qwen2.5-3b	0.163	0.417	0.195	0.208	0.364	0.245	0.163	0.417	0.197	**0.291**	**0.471**	**0.317**
	qwen2.5-7b	0.079	0.255	0.094	0.302	0.450	0.336	0.169	0.368	0.190	**0.354**	**0.557**	**0.393**
Large LLM	mistral-small-24b	0.322	0.478	0.352	0.391	0.543	0.431	0.378	0.530	0.410	**0.573**	**0.724**	**0.606**
	cogito-32b	0.440	0.614	0.483	0.443	0.591	0.481	0.430	0.614	0.476	**0.577**	**0.723**	**0.609**
	qwen2.5-32b	0.268	0.533	0.317	0.398	0.553	0.436	0.297	0.551	0.341	**0.577**	**0.721**	**0.607**
	GPT-4o	0.556	0.722	0.595	0.476	0.607	0.503	0.494	0.692	0.540	**0.595**	**0.740**	**0.629**
	Claude-3.7-Sonnet	0.612	0.763	0.655	0.489	0.633	0.540	0.619	**0.767**	0.659	**0.620**	0.764	**0.664**
	Average	0.260	0.480	0.307	0.322	0.471	0.362	0.286	0.508	0.331	**0.451**	**0.619**	**0.482**

Table 3: Evaluation results on the **TableBench** benchmark under our evaluation protocol. Bold and underline follow Table 2.

is publicly available.

SynTQA (Zhang et al., 2024a) ensembles text2SQL and end-to-end table QA (E2E TQA) models, leveraging their complementary strengths: text2SQL excels in numerical reasoning and handling large tables, while E2E TQA performs better with ambiguous questions. A lightweight selector (feature-based or LLM-based) chooses the final answer. For fair comparison, we used the same frozen LLM and prompts across all SynTQA components.

MixSC (Liu et al., 2024) integrates textual and symbolic reasoning via a self-consistency mechanism. It utilizes GPT-3.5 for direct prompting and Python code execution, aggregating outputs for robustness. A normalization module (NORM) further enhances stability against structural perturbations.

TabLaP (Wang et al., 2025) employs multiple LLMs for table QA, delegating numerical reasoning to Python scripts generated by an LLM (NumSolver), while using MixSC (Liu et al., 2024) for non-numerical questions. An LLM-based selector (AnsSelector) chooses the more reliable answer, aided by a trustworthiness evaluator (TwEvaluator) for reliability estimation.

We exclude certain baselines from the main cross-backbone comparison for the following reasons: their official publicly released code is exclusively compatible with the closed-source GPT series LLMs (Lu et al., 2023a; Zhang et al., 2024c), critical version dependencies that prevented execution (Cheng et al., 2023; Ye et al., 2023), or significant performance degradation when switching to different models, rendering comparisons meaningless (Wang et al., 2024; Gao et al., 2023). For GPT-specific tool-use baselines, we provide a separate GPT-4o-based comparison in Appendix E.5.

We used two benchmarks of varying difficulty. To verify that MATA does not rely on the distribution of the training datasets used for *Sch* and *CC*, we deliberately selected benchmark datasets different from the three datasets (WikiTQ, TabMWP, TabFact) used in training.

Penguins in a Table (Srivastava et al., 2023) is a diagnostic dataset from BIG-bench designed to test basic table reasoning. It presents a single table of penguin species with attributes such as height and weight, and asks simple factual or comparative questions. This isolates core table understanding without involving complex language or multi-step logic, making it a relatively easy task.

TableBench (Wu et al., 2025) is a benchmark for complex table QA, spanning 18 subcategories including fact-checking, numerical reasoning, data analysis, and visualization. Tables are sourced from diverse domains such as finance, sports, and science. The benchmark emphasizes real-world complexity and demands diverse reasoning strategies.

4.2 Metrics and Setup

In many previous TableQA studies, Exact Match (EM) accuracy has been the most commonly used evaluation metric. However, EM alone is insufficient for assessing the quality of answers generated by LLMs. Therefore, unlike prior work, we employed two additional metrics alongside EM. First, we used fuzzy matching[2], a metric widely adopted in studies (King and Flanigan, 2024; Cheng et al., 2024; Nekvinda and Dušek, 2021) to measure textual similarity based on Levenshtein distance (Levenshtein, 1966). Second, we adopted the SQuAD-style token-level F1 score (Rajpurkar et al., 2016), which evaluates token-level overlap between the prediction and the ground truth. By incorporating these two metrics, we complement the strictness of EM with more flexible and nuanced evaluations.

The MATA framework is implemented using LangChain[3] and leveraged Ollama[4] to support open-source LLMs. This setup enables flexible model switching, provided that the models are supported by Ollama. We defined LLMs with parameter sizes under 10B as *small LLMs* (Meta AI, 2024; Jiang et al., 2023; Qwen et al., 2025), and those with more than 10B parameters or that are closed-source as *large LLMs* (Mistral AI, 2025; Deep Cogito, 2025; Qwen et al., 2025; OpenAI et al., 2024; Anthropic, 2025). We used five models per category, totaling ten models across both small and large LLMs. MATA and all locally executable baseline runs were evaluated under the same inference conditions using a single A100 GPU.

[2]https://pypi.org/project/fuzzywuzzy/

[3]https://www.langchain.com/

[4]https://ollama.com/

4.3 Results

On the easier benchmark **Penguins in a Table** (Table 2), MATA maintained consistently high performance across all 10 models, clearly demonstrating its strong model-agnostic capability. In contrast, despite the simplicity of the tables and questions, the baselines TabLaP and MixSC exhibited significant performance declines when applied to smaller LLMs. This indicates that smaller LLMs possess insufficient capabilities for table analysis, and approaches such as TabLaP and MixSC fail to adequately address this shortfall.

While SynTQA achieved highest performance among baselines for some small LLMs on this benchmark, this advantage was confined to simple, single-table reasoning tasks. We attribute this to its avoidance of excessive reasoning. Whereas TabLaP and MixSC entail more than ten LLM inferences to derive a final answer, SynTQA requires only three. This finding aligns with prior findings (Huang et al., 2025; Chen et al., 2024a), which posits that repeated inference in straightforward tasks can lead to performance degradation.

However, this design choice comes at the expense of generalizability and flexibility in more complex tasks. On **TableBench** (Table 3), which includes larger tables and more challenging questions, MATA achieved the best performance across all models. For example, MATA outperformed the best baseline (SynTQA) by 40.1% (EM), 21.9% (fuzzy), and 33.1% (F1). Notably, SynTQA did not maintain its lead on these harder tasks, and in some cases was outperformed by TabLaP and MixSC, especially when leveraging closed-source models. This suggests that approaches relying on minimal reasoning steps, such as SynTQA, may not generalize well to complex table understanding scenarios, where a more nuanced and adaptive reasoning strategy is necessary. Furthermore, TabLaP and MixSC, originally evaluated only with closed-source GPT series models, are specifically tuned for such environments and do not transfer robustly to open-source or smaller LLMs. In contrast, MATA employs multiple types of reasoning only to the extent necessary, achieving optimal performance across various models. In other words, MATA demonstrates superior generalization and stability, delivering optimal performance across a diverse range of models and task difficulties. This underscores the practical value of MATA's balanced and flexible

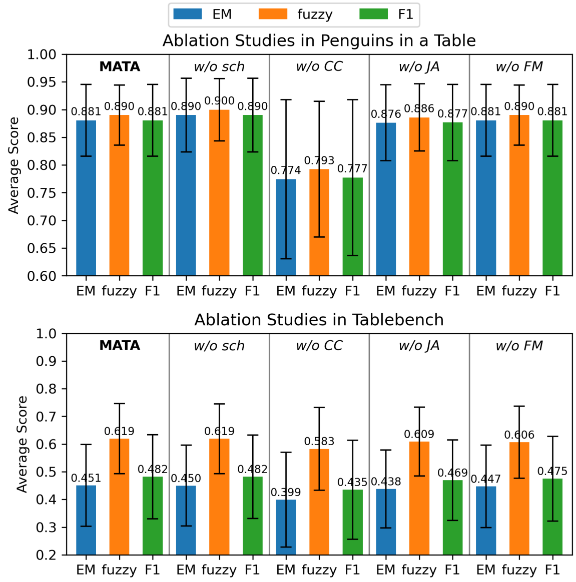

Figure 2: Ablation Studies on the **Penguins in a Table** (top) and **TableBench** (bottom). The scores shown in the graph represent the average across all models.

reasoning framework for diverse TableQA applications, where adaptability and reliability are crucial.

We also compare MATA with additional TableQA and multi-agent baselines in Appendix E.5. With GPT-4o as the shared backbone, MATA substantially outperforms ReAcTable (Zhang et al., 2024c) and Chameleon (Lu et al., 2023a) on both **Penguins in a Table** and **TableBench**. We further compare MATA against AutoPrep (Fan et al., 2025), another standard multi-agent framework, in which a single LLM backbone is tasked with planning, executing, and iteratively refining answers for question-aware data preparation in TableQA, using representative Qwen2.5 backbones. MATA performs better in most settings, suggesting that its gains are not limited to the three main baselines in Tables 2 and 3.

Furthermore, since the original **TableBench** study evaluates performance using the ROUGE-L metric, we provide corresponding ROUGE-L scores in Appendix E.3 to facilitate direct comparison.

4.4 Ablation Study

We also conduct an ablation study to analyze the contribution of each module in MATA. Figure 2 reveals that the Confidence Checker (*CC*) is the most critical component for overall performance. Removing *CC* from the framework results in the largest performance drop in accuracy, as it forces the system to rely solely on the Judge Agent (*JA*) to evaluate all candidate answers, which is a much less efficient strategy.

The main strength of the *CC* lies in its ability to bypass unnecessary reasoning when a candidate's confidence score is sufficiently high, allowing immediate selection of that answer without invoking additional LLM-based inference through the *JA*. This not only improves computational efficiency but also helps avoid performance degradation caused by excessive inference steps or prompts, a phenomenon reported in prior studies (Huang et al., 2025; Chen et al., 2024a).

Empirical results demonstrate that using *CC* significantly reduced the *JA* invocation frequency by 95.8% on the easier **Penguins in a Table** and by 60.6% on the more challenging **TableBench**, all while maintaining or improving final accuracy (see Appendix E for details). This demonstrates that efficient candidate selection, rather than exhaustive verification, yields both better performance and greater efficiency. The roles of the Judge Agent (*JA*) and the Format Matcher (*FM*) become more significant as task complexity increases. On simple benchmarks like **Penguins in a Table**, most correct answers can be identified directly by the *CC*, with little need for further processing. However, on challenging tasks such as **TableBench**, the additional verification and answer formatting provided by *JA* and *FM* consistently result in superior outcomes across all models. This pattern illustrates that modular, adaptive answer selection is especially important for complex TableQA tasks.

We also evaluated the impact of the Scheduler (*Sch*) module. Using the *Sch* led to a 14.6% reduction in LLM agent calls on **Penguins in a Table** and a 7.6% reduction on **TableBench**, aggregated across all 10 models, further improving computational efficiency (see Appendix E for details). However, its contribution to overall accuracy was context-dependent. In high-difficulty settings (e.g., **TableBench**), omitting the *Sch* and using all available reasoning paths sometimes led to marginally better results, likely due to increased answer diversity. Conversely, in easier tasks (e.g., **Penguins in a Table**), reducing the number of reasoning methods occasionally limited answer diversity, slightly diminishing confidence in the final prediction.

These results suggest a nuanced trade-off between efficiency and answer diversity. For complex table understanding problems, using all reasoning paths can, in some cases, increase the risk of spurious or conflicting inferences, potentially hindering the selection of the correct answer. In such scenarios, a moderate and well-chosen degree of reasoning (i.e., not simply maximizing the number of paths) tends to yield the best balance of efficiency and accuracy. Similarly, for low-difficulty tasks involving small tables and straightforward questions, we observed that eliminating certain reasoning methods by using the *Sch* sometimes led to a slight decrease in answer diversity, and thus in confidence or robustness of the final prediction. This underscores the value of generating a sufficiently diverse set of candidate answers, even in seemingly simple settings.

Therefore, our findings indicate that achieving optimal TableQA performance requires a careful balance between ensuring sufficient reasoning diversity and optimizing the number of inference steps. Strategic use of the *Sch* and other modules is essential: too many reasoning paths may introduce noise, while too few may miss important perspectives. These observations reinforce the modular design philosophy of MATA, highlighting that both efficiency and robustness are best served by flexible, context-aware inference orchestration.

4.5 End-to-End Efficiency

Model	TabLaP	SynTQA	MixSC	**MATA**
llama3.2-3b	15.44	5.95	12.58	21.32
mistral-7b	23.80	3.34	20.69	26.63
phi4-mini-3.8b	12.49	10.42	9.66	23.26
qwen2.5-3b	20.47	0.93	18.13	17.77
qwen2.5-7b	18.83	1.65	16.80	20.82
mistral-small-24b	93.88	12.45	84.07	47.31
cogito-32b	127.84	14.75	121.30	31.83
qwen2.5-32b	78.35	5.36	72.61	31.48
Average	48.89	6.86	44.48	27.55

Table 4: Average end-to-end latency in seconds per query for locally hosted open-source backbones. Latency is measured in the same local environment and averaged over **Penguins in a Table** and **TableBench**. Lower is better.

We further evaluate the end-to-end efficiency of MATA by measuring wall-clock response latency. Since closed-source API models introduce provider-side scheduling, network, and service-load overheads that are not directly comparable to local inference, we report latency only for open-source backbones executed in the same local environment.

Table 4 reports the average end-to-end latency per query. In the same local environment, the combined latency of the lightweight *Sch*/*CC*/*FM* tools accounts for only about 0.6% of MATA's total latency, confirming that the dominant cost comes from backbone LLM invocations. SynTQA is the fastest method because it uses a fixed inference budget with minimal LLM calls (only 3). However, this also limits its flexibility when handling more complex questions, resulting in lower performance, as shown in Table 3. In contrast, TabLaP and MixSC require heavier fixed inference budgets (12 and 10 LLM calls, respectively) and therefore incur substantially higher latency. MATA lies between these two extremes: it allocates additional computation when needed, but avoids unnecessary reasoning branches and Judge Agent (*JA*) calls through *Sch*- and *CC*-driven early exits. As a result, MATA achieves lower average latency than the call-heavy baselines while maintaining the strongest overall accuracy, especially on the more challenging **TableBench**.

5 Conclusion

We introduce MATA, a novel multi-agent framework for reliable and flexible Table Question Answering, which leverages diverse reasoning paths including Chain-of-Thought, Program-of-Thought, and text-to-SQL to generate multiple candidate answers, while employing lightweight tools and specialized agents to optimize selection and minimize costly LLM inferences. Overall, MATA advances TableQA capabilities by highlighting model-agnostic design and efficient reasoning with future work potentially integrating additional techniques or agents to broaden its applicability across diverse scenarios and data complexities.

Limitations

While MATA reduces inference cost by selectively executing reasoning methods and utilizing lightweight modules such as the scheduler and selector, it does not address the fundamental cost associated with the LLMs themselves. Each reasoning path still requires full LLM inference, which can be computationally expensive, especially for large-scale models. As a result, the overall efficiency of the system remains constrained by the inherent resource demands of LLM-based reasoning. Future work should explore LLM compression, distillation, or hybrid architectures that offload parts of the reasoning process to smaller or non-LLM components.

Ethical Considerations

Our proposed MATA framework, while designed to improve the reliability and efficiency of TableQA systems, can potentially be misused in ways that raise ethical concerns. Specifically, the model's ability to generate seemingly plausible answers via multiple reasoning paths (e.g., CoT, PoT, text2SQL) may be exploited to fabricate misleading tabular information or circumvent truthfulness checks in sensitive applications such as finance, healthcare, or public policy. Additionally, our model inherits biases from the underlying LLMs and datasets, which could manifest as systematic errors or unfair predictions, particularly when deployed without careful calibration. Although MATA is designed to work with open-source LLMs to encourage accessibility and transparency, this flexibility may also increase the risk of improper use in low-accountability environments. We strongly discourage the application of MATA in domains requiring verifiable factuality and fairness without additional safeguards. We advocate for its use in responsible, research-driven, and low-risk settings and encourage further research into robust hallucination detection and bias mitigation to enhance the ethical deployment of TableQA models. In accordance with the ACL Policy on AI Assistance, we acknowledge the use of Gemini[5] to assist with code debugging and writing polishing. All experimental designs, data analyses, and scientific claims presented in this work were verified by the authors.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) grants (RS-2024-00414981, RS-2025-00560762) and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grants (RS-2024-00397085, RS-2021-II211343). This research was also conducted as part of the Creative-Pioneering Researchers Program and the Bio-Connect Program through the Bio-MAX Institute, Seoul National University. J. Do is with ASRI, Seoul National University.

[5] https://deepmind.google/technologies/gemini/

References

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, and 8 others. 2024. Phi-4 technical report. Preprint, arXiv:2412.08905.

Anthropic. 2025. Claude 3.7 sonnet and claude code. https://www.anthropic.com/news/claude-3-7-sonnet. Accessed on 2025-07-25.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. Preprint, arXiv:2005.14165.

Donald D. Chamberlin and Raymond F. Boyce. 1974. Sequel: A structured english query language. In Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, SIGFIDET '74, page 249–264, New York, NY, USA. Association for Computing Machinery.

Lingjiao Chen, Jared Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. 2024a. Are more llm calls all you need? towards the scaling properties of compound ai systems. In Advances in Neural Information Processing Systems, volume 37, pages 45767–45790. Curran Associates, Inc.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. Transactions on Machine Learning Research.

Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2020. Tabfact: A large-scale dataset for table-based fact verification. In International Conference on Learning Representations.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024b. Teaching large language models to self-debug. In The Twelfth International Conference on Learning Representations.

Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflow-guided retrieval augmentation for repository-level code completion. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 7957–7977, Bangkok, Thailand. Association for Computational Linguistics.

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. ICLR, abs/2210.02875.

Deep Cogito. 2025. Cogito v1 preview: Introducing ida as a path to general superintelligence. https://www.deepcogito.com/research/cogito-v1-preview. Accessed: 2025-08-02.

Meihao Fan, Ju Fan, Nan Tang, Lei Cao, Guoliang Li, and Xiaoyong Du. 2025. Autoprep: Natural language question-aware data preparation with a multi-agent framework. Proc. VLDB Endow., 18(10):3504–3517.

Xi Fang, Weijie Xu, Fiona Anting Tan, Jiani Zhang, Ziqing Hu, Yanjun Qi, Scott Nickleach, Diego Socolinsky, Srinivasan Sengamedu, and Christos Faloutsos. 2024. Large language models (llms) on tabular data: Prediction, generation, and understanding - a survey. Preprint, arXiv:2402.17944.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In Proceedings of the 40th International Conference on Machine Learning, volume 202 of Proceedings of Machine Learning Research, pages 10764–10799. PMLR.

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. In Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 4971–4983, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Pengcheng He, Jianfeng Gao, and Weizhu Chen. 2023. DeBERTav3: Improving deBERTa using ELECTRA-style pre-training with gradient-disentangled embedding sharing. In The Eleventh International Conference on Learning Representations.

Audrey Huang, Adam Block, Qinghua Liu, Nan Jiang, Akshay Krishnamurthy, and Dylan J Foster. 2025. Is best-of-n the best of them? coverage, scaling, and optimality in inference-time alignment. In Forty-second International Conference on Machine Learning.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. Preprint, arXiv:2409.12186.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. Preprint, arXiv:2310.06825.

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

Zhewei Kang, Xuandong Zhao, and Dawn Song. 2025. Scalable best-of-n selection for large language models via self-certainty. Preprint, arXiv:2502.18581.

Brendan King and Jeffrey Flanigan. 2024. Unsupervised end-to-end task-oriented dialogue with LLMs: The power of the noisy channel. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 8283–8300, Miami, Florida, USA. Association for Computational Linguistics.

Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady, 10(8):707–710.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 7871–7880, Online. Association for Computational Linguistics.

Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2024. Table-gpt: Table fine-tuned gpt for diverse table tasks. Proc. ACM Manag. Data, 2(3).

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2022. TAPEX: Table pre-training via learning a neural SQL executor. In International Conference on Learning Representations.

Tianyang Liu, Fei Wang, and Muhao Chen. 2024. 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.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and

Jianfeng Gao. 2023a. Chameleon: Plug-and-play compositional reasoning with large language models. In Advances in Neural Information Processing Systems, volume 36, pages 43447–43478. Curran Associates, Inc.

Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and Ashwin Kalyan. 2023b. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. In The Eleventh International Conference on Learning Representations.

Renjie Lu. 2019. Malware detection with lstm using opcode language. arXiv preprint arXiv:1906.04593.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In Advances in Neural Information Processing Systems, volume 36, pages 46534–46594. Curran Associates, Inc.

Meta AI. 2024. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/. Accessed July 25, 2025.

Mistral AI. 2025. Mistral small 3. https://mistral.ai/news/mistral-small-3. Accessed: 2025-08-02.

Tomáš Nekvinda and Ondřej Dušek. 2021. Shades of BLEU, flavours of success: The case of MultiWOZ. In Proceedings of the 1st Workshop on Natural Language Generation, Evaluation, and Metrics (GEM 2021), pages 34–46, Online. Association for Computational Linguistics.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In Proceedings of the 40th International Conference on Machine Learning (ICML'23).

OpenAI, :, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Mądry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, and 401 others. 2024. Gpt-4o system card. Preprint, arXiv:2410.21276.

OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/. Accessed on 2025-07-25.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 1470–1480, Beijing, China. Association for Computational Linguistics.

Sohan Patnaik, Heril Changwal, Milan Aggarwal, Sumit Bhatia, Yaman Kumar, and Balaji Krishnamurthy. 2024. CABINET: Content relevance-based noise reduction for table question answering. In The Twelfth International Conference on Learning Representations.

Python Software Foundation. 2024. difflib — Helpers for computing deltas. Python 3.13.3 documentation.

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, Junyang Lin, and 24 others. 2025. Qwen2.5 technical report. Preprint, arXiv:2412.15115.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research, 21(140):1–67.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, pages 2383–2392, Austin, Texas. Association for Computational Linguistics.

John W. Ratcliff and David E. Metzener. 1988. Pattern matching: The gestalt approach. Dr. Dobb's Journal, 13(7):46–51.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code llama: Open foundation models for code. Preprint, arXiv:2308.12950.

Yewei Song, Cedric Lothritz, Xunzhu Tang, Tegawendé F. Bissyandé, and Jacques Klein. 2024. Revisiting code similarity evaluation with abstract syntax tree edit distance. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), pages 38–46.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R. Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, Agnieszka Kluska, Aitor Lewkowycz, Akshat Agarwal, Alethea Power, Alex Ray, Alex Warstadt, Alexander W. Kocurek, Ali Safaya, Ali Tazarv, and 431 others. 2023. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. Transactions on Machine Learning Research. Featured Certification.

Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. MobileBERT: a compact task-agnostic BERT for resource-limited devices. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 2158–2170, Online. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in Neural Information Processing Systems, volume 30. Curran Associates, Inc.

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

Yuxiang Wang, Jianzhong Qi, and Junhao Gan. 2025. Accurate and regret-aware numerical problem solver for tabular question answering. Proceedings of the AAAI Conference on Artificial Intelligence, 39(12):12775–12783.

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. 2024. Chain-of-table: Evolving tables in the reasoning chain for table understanding. In The Twelfth International Conference on Learning Representations.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In Advances in Neural Information Processing Systems, volume 35, pages 24824–24837. Curran Associates, Inc.

Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In Proceedings of the 9th Python in Science Conference, pages 56 – 61.

Xianjie Wu, Linzheng Chai, Jian Yang, Ge Zhang, Xeron Du, Jiaheng Liu, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, Tongliang Li, Zhoujun Li, and Guanglin Niu. 2024. Tablellm: Enabling tabular data manipulation by llms in real-world scenarios with complex table structures. arXiv preprint arXiv:2402.17944.

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xeron Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, Tongliang Li, Zhoujun Li, and Guanglin Niu. 2025. Tablebench: A comprehensive and complex benchmark for table question answering. Proceedings of the AAAI Conference on Artificial Intelligence, 39(24):25497–25506.

Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. 2023. Large language models are versatile decomposers: Decomposing evidence and questions for table-based reasoning. In Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '23, page 174–184, New York, NY, USA. Association for Computing Machinery.

Siyue Zhang, Anh Tuan Luu, and Chen Zhao. 2024a. SynTQA: Synergistic table-based question answering via mixture of text-to-SQL and E2E TQA. In Findings of the Association for Computational Linguistics: EMNLP 2024, pages 2352–2364, Miami, Florida, USA. Association for Computational Linguistics.

Tianshu Zhang, Xiang Yue, Yifei Li, and Huan Sun. 2024b. TableLlama: Towards open large generalist models for tables. 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 6024–6044, Mexico City, Mexico. Association for Computational Linguistics.

Weixu Zhang, Yifei Wang, Yuanfeng Song, Victor Junqiu Wei, Yuxing Tian, Yiyan Qi, Jonathan H. Chan, Raymond Chi-Wing Wong, and Haiqin Yang. 2023. Natural language interfaces for tabular data querying and visualization: A survey. IEEE Transactions on Knowledge and Data Engineering.

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

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

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In Findings of the Association for Computational Linguistics: ACL 2024, pages 851–870, Bangkok, Thailand. Association for Computational Linguistics.

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

Appendix

A Model Performance Differences Depending On The Reasoning Path

We analyze the relative strengths of three major reasoning strategies for Table QA—Chain-of-Thought (CoT) (Wei et al., 2022), Program-of-Thought (PoT) (Chen et al., 2023), and text-to-SQL (Text2SQL) (Zhong et al., 2017)—across different reasoning categories. In total, we evaluate 20 models, including 18 open-source models ranging in size from 3B to 32B and two closed-source models (See Figure 3). For a fair comparison, we use only the prompts defined in Appendix F for the CoT Agent (CoTA), PoT Agent (PoTA), and Text2SQL Agent (t2SA), excluding Debug Agents that might bias results in favor of CoTA.

To evaluate under realistic and challenging conditions, we conduct experiments on TableBench (Wu et al., 2025), a recently introduced benchmark that spans 18 subcategories across four reasoning domains, including numerical reasoning and fact checking. TableBench provides a more faithful reflection of real-world scenarios and enables fine-grained performance analysis across diverse question types.

As shown in Figure 3, in several models, and consistent with prior findings (Liu et al., 2024; Zhang et al., 2024a; Wang et al., 2025), CoT tends to perform well on fact-checking and data interpretation tasks, while PoT and text2SQL show better results for complex arithmetic and aggregation-based questions. This trend is particularly evident in both closed-source models. That is, the patterns observed in prior studies appear more frequently in models with 10B+ parameters or closed-source configurations.

However, our results also reveal that such trends vary substantially depending on the model series and size. For instance, among 7B-sized models, text2SQL excels in mistral, PoT dominates in qwen2.5-coder, and CoT performs best in qwen2.5. Within the qwen2.5 family, smaller models like 3B and 7B favor CoT the most, followed by text2SQL, with PoT lagging behind. As the size increases to 14B and 32B, PoT's performance improves dramatically. In the code-specialized qwen2.5-coder series, PoT significantly outperforms its performance in the general-purpose qwen2.5 models.

In conclusion, we find no consistent pattern in the performance of CoT, PoT, or text2SQL across factors such as model size, architecture series, or whether the model is open- or closed-source. Numerous exceptions emerge depending on the specific model, making it difficult to draw universal trends. These findings suggest that the effectiveness of a reasoning strategy is influenced not only by the question type but also by a wide range of other factors, including model capacity and internal design. While previous studies (Liu et al., 2024; Zhang et al., 2024a; Wang et al., 2025) typically categorized reasoning methods based on task types, our analysis shows that such categorizations do not generalize well across different model scales or series. Therefore, robust Table QA requires flexible and dynamic selection of reasoning strategies.

B Details of Training Dataset, Scheduler(*sch*) and Confidence Checker(*CC*)

This section provides detailed information about the training datasets as well as the specific implementation of the scheduler (*sch*) and the Confidence Checker (*CC*).

B.1 Training Dataset for *Sch* and *CC*

We construct a large-scale training dataset specifically designed to support the fine-tuning of lightweight tools—*Sch* and *CC*—within the multi-reasoning TableQA framework MATA.

To build this dataset, we leverage three publicly available TableQA datasets: WikiTQ (Pasupat and Liang, 2015), TabMWP (Lu et al., 2023b), and TabFact (Chen et al., 2020). Detailed statistics for each source dataset are provided in Table 5.

- **WikiTQ** (Pasupat and Liang, 2015) is a widely used benchmark for table-based question answering, consisting of natural language questions over semi-structured Wikipedia tables. Each question is paired with a crowd-annotated answer. The dataset emphasizes compositional reasoning, covering operations such as superlatives, aggregation, and arithmetic. Tables are disjoint across splits to test generalization to unseen schemas.
- **TabMWP** (Lu et al., 2023b) is a dataset of math word problems grounded in semi-structured tables, requiring multi-step numerical reasoning. Questions are mostly free-text with diverse answer types including integers, decimals, and spans. It combines textual and tabular cues, posing challenges in alignment and multi-hop symbolic reasoning.
- **TabFact** (Chen et al., 2020) is a large-scale fact

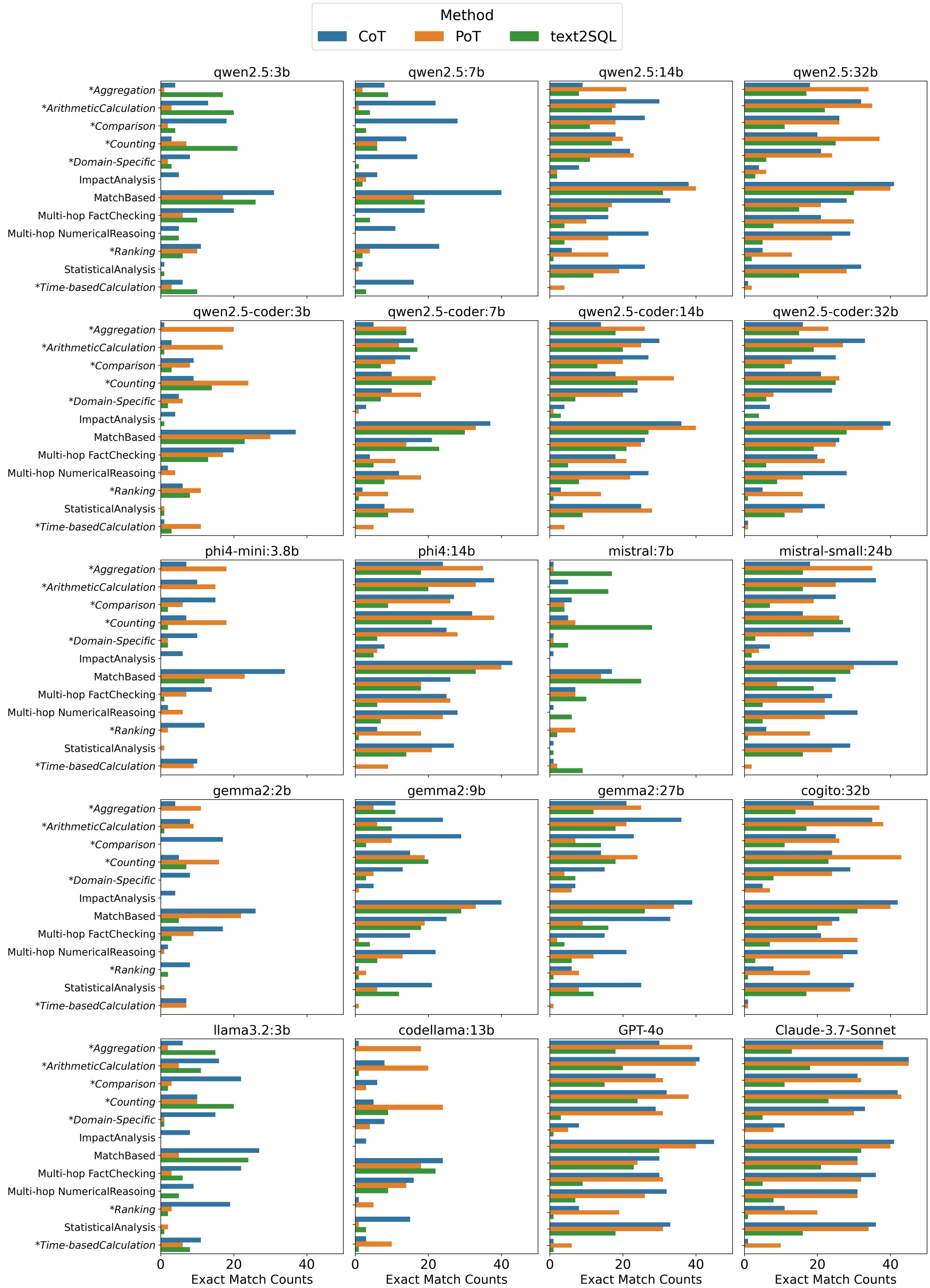

Figure 3: Exact Match (EM) accuracy comparison of CoT, PoT, and text2SQL across LLMs on the TableBench (Wu et al., 2025) dataset. The figure highlights each method's strengths and weaknesses by question category. Asterisks on the y-axis indicates categories related to Numerical Reasoning.

Dataset	#Train	#Val	#Test	Main Task	Table Source
WikiTQ	11,321	2,831	4,344	Compositional QA	Wikipedia (HTML tables)
TabMWP	23,059	7,686	7,686	Multi-step Math QA	Curated semi-structured tables
TabFact	92,283	12,792	12,779	Fact Verification	Wikipedia (infobox-style tables)

Table 5: Summary of table QA datasets used for training *CC* and *sch*.

Dataset	# (*T*, *Q*) pairs	# table cells	LLM	# CoT	# PoT	# text2SQL	# Incorrect All
WikiTQ	14,152	162.3	CodeLLaMA:13B	4,832	3,813	3,284	6,378
			phi4:14B	10,034	7,877	5,117	2,578
			Qwen2.5-Coder:14B	8,802	8,290	5,625	2,795
TabMWP	23,007	11.3	CodeLLaMA:13B	11,403	2,988	4,542	9,326
			phi4:14B	21,592	16,806	8,464	1,004
			Qwen2.5-Coder:14B	21,254	17,536	9,055	1,065
TabFact	20,729	85.6	CodeLLaMA:13B	11,044	5,961	289	7,638
			phi4:14B	17,580	12,188	10,249	1,655
			Qwen2.5-Coder:14B	16,313	13,601	10,173	1,863

Table 6: Summary of table QA datasets used for the training *CC* and *sch*. **# table cells** denotes the average number of cells per table in each dataset.**# CoT**, **# PoT**, and **# text2SQL** indicate the number of questions correctly answered by each reasoning method according to the Exact Match metric.

verification dataset containing human-annotated statements labeled as ENTAILED or REFUTED, grounded in Wikipedia tables. It requires both linguistic inference and symbolic reasoning. Statements vary in complexity, from simple row-level facts to multi-step logical compositions involving comparison and aggregation.

Specifically, we use the entire training and validation sets from WikiTQ (14,152 examples), examples with indices 0 through 23,006 from the TabMWP training set (23,007 examples), and examples with indices 0 through 20,728 from the TabFact training set (20,729 examples).

A total of 57,888 unique table-question pairs (*T*, *Q*) are processed using three large language models— CodeLLaMA:13B (Rozière et al., 2024), phi4:14B (Abdin et al., 2024), and Qwen2.5-Coder:14B (Hui et al., 2024)—to yield 173,664 samples. For each (*T*, *Q*) pair, we generate outputs from three reasoning paradigms: Chain-of-Thought (CoT), Program-of-Thought (PoT), and text2SQL, all executed using the same inference pipeline as in the MATA framework. Specifically, CoT reasoning is performed once using the CoT Agent (CoTA), while PoT and text2SQL utilize inference-and-debug loops via the PoT Agent (PoTA) and Python Debug Agent (PDA), and the text2SQL Agent (t2SA) and SQL Debug Agent (SDA), respectively. (For further details on the reasoning and interactions among LLM agents, refer to the Methodology section of the main paper.)

To foster research on improving the efficiency and adaptability of TableQA systems, the dataset is released at `https://github.com/AIDASLab/MATA/tree/main/experiments` for free academic use—along with documentation and licensing. For detailed statistics and structure of the training dataset, see Table 6 and Figure 4.

B.2 Scheduler (*Sch*)

MATA optionally includes a scheduler module to optimize inference efficiency. The scheduler predicts which code-based reasoning path—PoT or text2SQL—is more likely to succeed, allowing MATA to prioritize that path while deferring or skipping the other.

The scheduler (denoted as *Sch*) serves as the initial module, processing the input table (*T*) and question (*Q*). Based on various features extracted from *T* (e.g., table size, schema, data types) and the semantic content of *Q*, *Sch* determines whether to perform PoT or text2SQL reasoning first. It is implemented as a lightweight model combining MobileBERT (Sun et al., 2020) and a two-layer MLP, totaling 24.65M parameters.

The architecture of the scheduler (*Sch*) is illustrated in Figure 5. The scheduler first uses MobileBERT (Sun et al., 2020), a lightweight text encoder, to embed a concatenated text input that includes both the question and the table schema. Additional hand-crafted features are then concatenated to this embedded vector. These features are used as cheap routing cues for deciding whether PoT or text2SQL should be attempted first. Row, column, and cell counts capture table scale, where symbolic querying can be more useful for large tables. Question length and numeric-token counts capture question complexity and arithmetic demand. Schema-question overlap reflects column-localization ambiguity, and value-type flags indicate whether numerical computation, string matching, or missing-value handling is likely to dominate. These features are used only as inputs to *Sch*, not as direct decision rules. This design is motivated by SynTQA's

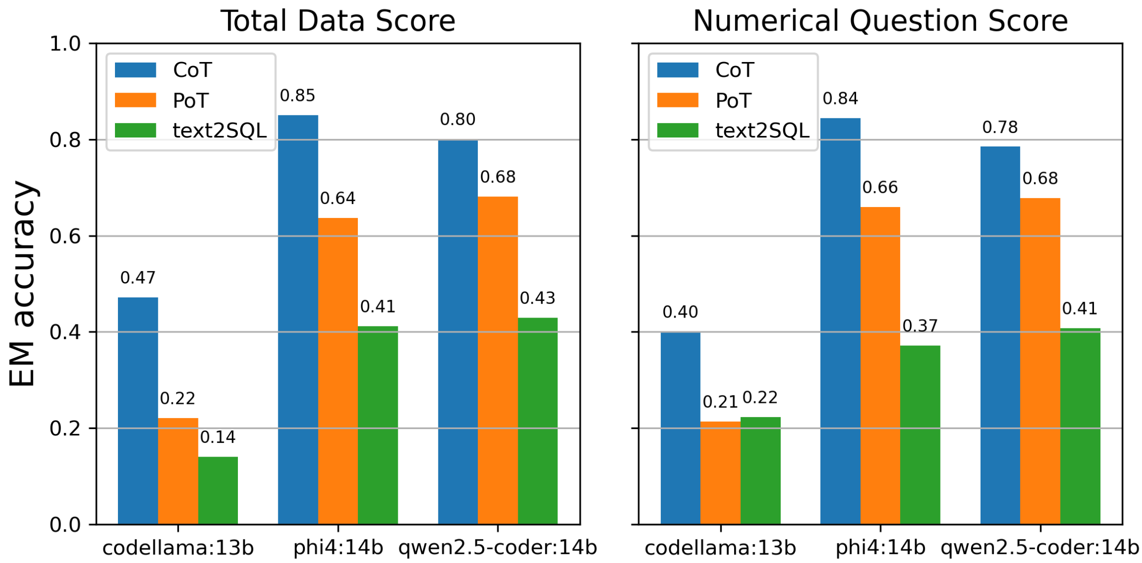

Figure 4: Exact Match (EM) accuracy on total training datasets: overall accuracy (left) and accuracy on numerical questions only (right). The x-axis represents different LLMs.

feature-based selector and MixSC's error analysis, which identify table scale, numeric demand, and schema-question localization as important cues for choosing reliable symbolic reasoning paths.

Numeric features (6):

- Number of rows in the table
- Number of columns in the table
- Overall table size
- Number of unique words in the question
- Number of numeric tokens in the question
- Number of overlapping words between the question and the table schema

Boolean features (4):

- Whether integer-type values exist in table cells
- Whether float-type values exist
- Whether string-type values exist
- Whether NaN values exist

In total, ten additional features are appended to the embedding vector produced by MobileBERT. This combined vector is passed through a two-layer Multi-Layer Perceptron (MLP), which outputs a final one-dimensional vector of length 2.

Training was conducted using the dataset described in Appendix B.1. Specifically, for each Table-Question pair (T, Q), we label whether the PoT and text2SQL reasoning methods produce the correct answer based on the Exact Match metric, assigning binary values (0 or 1). These labels are then used to construct two-dimensional ground truth vectors, which supervise the fine-tuning of the scheduler (*Sch*). The model is trained with Binary Cross Entropy loss for 45 epochs, using a batch size of 128 on a single NVIDIA A100 GPU.

At inference time, the scheduler outputs two logits, and MATA executes the reasoning path corresponding to the higher logit first, alongside CoT (Chain-of-Thought) reasoning. If the two results agree, the remaining reasoning path is skipped. If they disagree, the third path is also executed, and all three outputs are passed to the final answer selection stage. In this way, the scheduler reduces unnecessary LLM calls when confident, while preserving robustness by ensuring full fallback execution when disagreement occurs. When computational cost is not a concern, the scheduler can be disabled entirely to allow all three reasoning paths to run by default.

B.3 Confidence Checker (*CC*)

The Confidence Checker (hereafter *CC*) module takes as input the table T, question Q, and candidate answers generated from CoT, PoT, and text2SQL reasoning methods, assigning a confidence score to each candidate. Prior research (Gu et al., 2022) has reported strong performance of DeBERTaV3 (He et al., 2023) on table understanding tasks. Con-

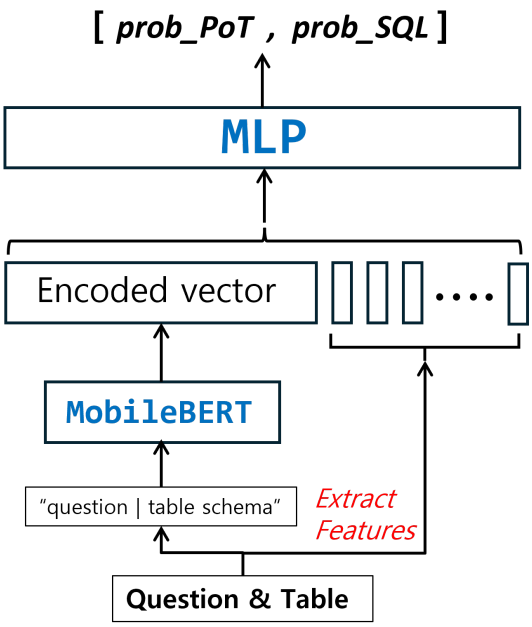

Figure 5: The architecture of the scheduler module in MATA. The scheduler encodes the input question and table schema using MobileBERT, and then concatenates the resulting vector with ten hand-crafted features extracted from the table and question. A two-layer MLP processes this combined representation and outputs the probabilities of success for the PoT and text2SQL reasoning paths, allowing MATA to prioritize the more promising one during inference.

sequently, *CC* was implemented by fine-tuning DeBERTaV3-large, which has 435M parameters.

Key techniques used in implementing the Confidence Checker are as follows:

B.3.1 Soft Labeling

Initially, the CC is fine-tuned on a previously collected training dataset, taking each reasoning method's textual outputs as input and producing a 1-dimensional vector of length 3. Each element of this vector represents the confidence score for the respective PoT, text2SQL, and CoT reasoning paths. To perform supervised fine-tuning of DeBERTaV3-large, we labeled each vector by comparing the final predictions of each reasoning method with the ground truth. Rather than simply labeling using binary results (0 or 1) based on Exact Match, we employed a two-stage labeling process. Figure 6 illustrates this procedure. First, predictions are evaluated using Exact Match; predictions with Exact Match results labeled as True are assigned a score of 1.0. Subsequently, for predictions that failed the Exact Match, an additional partial score was assigned based on F1-score evaluation. This allowed for semantic diversity in labeling, even if predictions did not exactly match the ground truth.

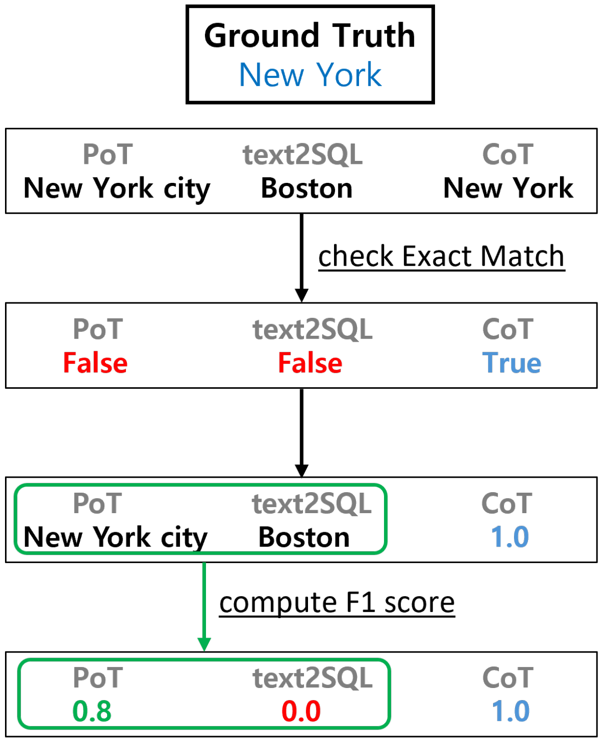

Figure 6: This figure shows the labeling rules for training the Confidence Checker. Soft labeling is performed in two orders: (1) If the final answer obtained from a method exactly matches the ground truth, it is labeled as 1.0. (2) When exact matching fails, we assign the label based on the F1 score (Rajpurkar et al., 2016) between the model's answer and the ground truth. In the case of CoT, exact matching is successful and thus directly used for supervision.

B.3.2 Special Tokens

XML-style structured special tokens were introduced to stabilize the training of DeBERTaV3-large. The added tokens are listed in Table 7.

These tokens structured all debugging results from *CoTA*, *PoTA*, *t2SA*, *PDA*, and *SDA*, preparing them as inputs for fine-tuning DeBERTaV3-large. The <NOTHING> token serves a special role. Any reasoning path containing this token is guaranteed to be incorrect, thus receiving a training confidence score of 0. The <NOTHING> token is used in two specific scenarios: first, reasoning paths deemed unnecessary by the scheduler are marked with <NOTHING>. For example, if the scheduler decides to execute text2SQL first and the CoT and text2SQL predictions match, PoT inference is skipped, and the corresponding position is

Category	Special Tokens
Table Metadata	<Table_row_size>, </Table_row_size>, <Table_column_size>, </Table_column_size>, <Table_size>, </Table_size>, <Table>, </Table>
Input/Output	<Question>, </Question>, <solution>, </solution>, <answer>, </answer>
Reasoning Types	<PoT>, </PoT>, <text2sql>, </text2sql>, <CoT>, </CoT>
Execution Steps	<N=k_code>, </N=k_code>, <N=k_execution_result>, </N=k_execution_result> (where $k \in \{0, 1, 2, 3\}$)
For *Sch*	<NOTHING>

Table 7: List of structured special tokens added to DeBERTaV3-large tokenizer.

forcibly filled with the <NOTHING> token. Consequently, the skipped PoT path is represented by <NOTHING>, and the CC effectively evaluates only CoT and text2SQL.

Additionally, some poorly performing LLMs occasionally generate irrelevant long token sequences when presented with excessively large tables or difficult questions. Such results hinder training and unnecessarily consume computational resources. Therefore, if the reasoning output exceeds 3000 characters, it is discarded and replaced by the <NOTHING> token, allowing the CC to efficiently assign lower confidence scores.

In the Figure 7 and Figure 8, we present examples of input texts to the Confidence Checker that include the added special tokens.

B.3.3 Segmented Training for Large Tables

Training DeBERTaV3-large with excessively large tables posed challenges due to GPU memory constraints. Therefore, tables exceeding a maximum token limit were segmented for training purposes. Specifically, the table schema, considered critical information, was maintained entirely without omissions. Instead, the table was segmented row-wise, without dividing columns. Additionally, each epoch introduced a sliding window, shifting the rows downward sequentially, resulting in augmented datasets per epoch.

Moreover, prior studies have shown that table size is a critical factor influencing performance in table understanding. Accordingly, we also include the number of rows, columns, and the overall size of the table as input features to the Confidence Checker. A more detailed analysis of table size can be found in Appendix E.6.

Training utilized two 80GB NVIDIA A100 GPUs with a batch size of 18 per device. The initial learning rate was set at 1e-6, employing a cosine learning rate scheduler and weight decay of 0.01 over 10 epochs.

An Example Input Format for the Confidence Checker

```
<Table_row_size>5</Table_row_size>
<Table_column_size>2</Table_column_size>
<Table_size>10</Table_size>
<Table>
| Day        | Boxes of cookies |
|:-----------|-----------------:|
| Tuesday    |               25 |
| Wednesday  |               27 |
| Thursday   |               23 |
| Friday     |               26 |
| Saturday   |               23 |
</Table>
<Question>A Girl Scout troop recorded how many boxes of cookies they sold each day for a week.
According to the table, what was the rate of change between Wednesday and Thursday?</Question>
<PoT>
<N=0_code>
# Calculate the rate of change between Wednesday and Thursday
rate_of_change = df.loc[1, 'Boxes of cookies'] - df.loc[2, 'Boxes of cookies']
ans = rate_of_change</N=0_code>
<N=0_execution_result>4</N=0_execution_result>
<N=1_code>
# Calculate the rate of change between Wednesday and Thursday
rate_of_change = df.loc[1, 'Boxes of cookies'] - df.loc[2, 'Boxes of cookies']
ans = rate_of_change</N=1_code>
<N=1_execution_result>4</N=1_execution_result>
</PoT>
<text2sql>
<N=0_code>SELECT (b.Boxes_of_cookies - a.Boxes_of_cookies) AS answer
FROM dataframe a
JOIN dataframe b ON a.Day = 'Wednesday' AND b.Day = 'Thursday';</N=0_code>
<N=0_execution_result>[[-4]]</N=0_execution_result>
</text2sql>
<CoT>
<solution>To find the rate of change in the number of boxes sold between Wednesday and Thursday,
we need to calculate the difference in the number of boxes sold on these days.
Step 1: Identify the values from the table:
- Boxes sold on Wednesday = 27
- Boxes sold on Thursday = 23
Step 2: Calculate the rate of change by subtracting the number of boxes sold on Thursday from the
number sold on Wednesday:
Rate of Change = Boxes sold on Wednesday - Boxes sold on Thursday = 27 - 23 = 4
The rate of change is a decrease of 4 boxes.</solution>
<answer>-4</answer>
</CoT>
```

Figure 7: An Example Input Format for the Confidence Checker

An Example Input Format for the Confidence Checker with the <NOTHING> Token

```
<Table_row_size>5</Table_row_size>
<Table_column_size>2</Table_column_size>
<Table_size>10</Table_size>
<Table>
| Day        | Boxes of cookies |
|:-----------|-----------------:|
| Tuesday    |               25 |
| Wednesday  |               27 |
| Thursday   |               23 |
| Friday     |               26 |
| Saturday   |               23 |
</Table>
<Question>A Girl Scout troop recorded how many boxes of cookies they sold each day for a week.
According to the table, what was the rate of change between Wednesday and Thursday?</Question>
<PoT>
<N=0_code><NOTHING></N=0_code>
<N=0_execution_result><NOTHING></N=0_execution_result>
<N=1_code><NOTHING></N=1_code>
<N=1_execution_result><NOTHING></N=1_execution_result>
</PoT>
<text2sql>
<N=0_code>SELECT (b.Boxes_of_cookies - a.Boxes_of_cookies) AS answer
FROM dataframe a
JOIN dataframe b ON a.Day = 'Wednesday' AND b.Day = 'Thursday';</N=0_code>
<N=0_execution_result>[[-4]]</N=0_execution_result>
</text2sql>
<CoT>
<solution>To find the rate of change in the number of boxes sold between Wednesday and Thursday,
we need to calculate the difference in the number of boxes sold on these days.
Step 1: Identify the values from the table:
- Boxes sold on Wednesday = 27
- Boxes sold on Thursday = 23
Step 2: Calculate the rate of change by subtracting the number of boxes sold on Thursday from the
number sold on Wednesday:
Rate of Change = Boxes sold on Wednesday - Boxes sold on Thursday = 27 - 23 = 4
The rate of change is a decrease of 4 boxes.</solution>
<answer>-4</answer>
</CoT>
```

Figure 8: An Example Input Format for the Confidence Checker with the <NOTHING> Token

C Details of Benchmarks and Baselines, and Metrics

C.1 Dataset

This section provides a detailed description of the two benchmark datasets we used (see Table 8).

• **TableBench** is a comprehensive benchmark for complex table QA, spanning 18 subcategories across fact checking, numerical reasoning, data analysis, and visualization. Tables are sourced from diverse domains such as finance, sports, and science. The benchmark emphasizes real-world complexity and supports multiple reasoning paradigms including TCoT, SCoT, and PoT. In our experiments, since our primary focus is on entity-type TableQA where the expected output is a concise entity or value, we evaluated only the entity-answer subset of TableBench. Specifically, we excluded examples whose targets are not short entity-style answers, such as those in the Descriptive Analysis, Anomaly Detection, Causal Analysis, and Chart Generation categories. As a result, 693 out of the original 886 test samples are included in our evaluation, using EM, fuzzy matching, and token-level F1 as the main metrics.

• **Penguins in a Table** is a diagnostic dataset from BIG-bench designed to test basic table reasoning. It presents a single table of penguin species with attributes like height and weight, and asks simple factual or comparative questions. This isolates core table understanding without involving complex language or multi-step logic. Since we targeted only TableQA on a single table as our task, we removed cases that perform QA with multiple tables. After refining the samples through a human evaluation check, a final set of 144 evaluation samples was used.

C.2 Baselines

• **TabLaP** (Wang et al., 2025) is a multi-LLM system for table QA that delegates numerical reasoning to Python code execution. It prompts an LLM (NumSolver) to generate a Python script for answering numerical questions, while using a SOTA TableQA model for non-numerical ones. A separate LLM-based module (AnsSelector) selects the more reliable answer between the two branches, and a trustworthiness evaluator (TwEvaluator) estimates answer reliability to support regret-aware usage.

• **SynTQA** (Zhang et al., 2024a) is an ensemble framework that combines Text-to-SQL and end-to-end table QA (E2E TQA) models by selecting the more reliable answer from both. It leverages the complementary strengths of each approach—Text-to-SQL excels at numerical reasoning and long tables, while E2E TQA is better at handling ambiguous questions and complex table content. A lightweight selector (either feature-based or LLM-based) chooses the final output, achieving improved accuracy and robustness over individual models.

• **MixSC** (Liu et al., 2024) is a table QA framework that combines textual and symbolic reasoning via a mix self-consistency mechanism. It uses GPT-3.5 to perform both direct prompting and Python code execution (via a shell agent), then aggregates multiple outputs from each reasoning path to improve answer robustness. A normalization module (NORM) further enhances stability against structural perturbations like table transposition and row shuffling.

C.3 Examples from the Penguins in a Table

In the TableQA example shown in Figure 9a, the question (Q) is "How many penguins are more than 8 years old?", the answer (A) is **1**. In the second TableQA example shown in Figure 9b, the question (Q) is "Which penguin is younger but taller than Gwen?", the answer (A) is **Bernard**.

C.4 Examples from the TableBench

In the TableQA example shown in Figure 10, the question (Q) is "What is the total increase in net assets over the 3-year period from 2005/2006 to 2007/2008?", the answer (A) is **4910084**, and the question type is `Time-basedCalculation`. In the second TableQA example shown in Figure 11, the question (Q) is "What is the average percentage of national votes won by all leaders in the table?", the answer (A) is **37.64%**, and the question type is `aggregation`.

C.5 Metrics

In many previous TableQA studies, Exact Match (EM) accuracy has been the most commonly used evaluation metric. However, EM alone is insufficient for assessing the quality of answers generated by LLMs. For instance, if the ground truth is "John and Andy" but the model outputs "John , Andy" EM would consider it incorrect, even though the answer is semantically accurate.

Therefore, unlike prior work, we employed two additional metrics alongside EM. First, we used

Dataset	#Train	#Val	#Test	Main Task	Table Source
TableBench	–	–	886	Complex Table QA	Domain tables (finance, science, etc.)
penguins in a table	–	–	144	Basic Table Reasoning	Manually created toy table

Table 8: Summary of table QA datasets used in our experiments. Since we targeted only TableQA on a single table as our task, we removed cases that perform QA with multiple tables from Penguins in a Table, and refined 144 evaluation samples through a human evaluation check. Regarding TableBench, it contains heterogeneous target formats, including short entity or numeric answers, sentence-level descriptive outputs, and chart-generation tasks. Since our focus is entity-type TableQA, where the expected output is a concise entity or value, we evaluate the entity-answer subset of TableBench (693 out of 886 examples) and use EM, fuzzy matching, and token-level F1 as the main metrics. We exclude examples whose targets are not short entity-style answers, such as those in the Descriptive Analysis, Anomaly Detection, Causal Analysis, and Chart Generation categories.

	name	age	height_cm	weight_kg
0	Louis	7	50	11
1	Vincent	9	60	11
2	Gwen	8	70	15

(a) Example of **Penguins in a Table** (Index=10).
Q: How many penguins are more than 8 years old?
A: 1

	name	age	height_cm	weight_kg
0	Louis	7	50	11
1	Bernard	5	80	13
2	Vincent	9	60	11
3	Gwen	8	70	15

(b) Example of **Penguins in a Table** (Index=140).
Q: Which penguin is younger but taller than Gwen?
A: Bernard

Figure 9: Two examples from the **Penguins in a Table** dataset.

	year	total support and revenue	total expenses	increase in net assets	net assets at end of year
0	2003 / 2004	80129	23463	56666	56666
1	2004 / 2005	379088	177670	211418	268084
2	2005 / 2006	1508039	791907	736132	1004216
3	2006 / 2007	2734909	2077843	654066	1658282
4	2007 / 2008	5032981	3540724	3519886	5178168
5	2008 / 2009	8658006	5617236	3053599	8231767
6	2009 / 2010	17979312	10266793	6310964	14542731
7	2010 / 2011	24785092	17889794	9649413	24192144
8	2011 / 2012	38479665	29260652	10736914	34929058

Figure 10: Example of **TableBench** dataset (Index=385). The question (Q) is **What is the total increase in net assets over the 3-year period from 2005/2006 to 2007/2008?**, the answer (A) is **4910084**

fuzzy matching[6], a metric widely adopted in various studies (King and Flanigan, 2024; Cheng et al., 2024; Nekvinda and Dušek, 2021) to measure textual similarity based on Levenshtein distance (Levenshtein, 1966). Second, we adopted the SQuAD-style token-level F1 score (Rajpurkar et al., 2016), which evaluates token-level overlap between the prediction and the ground truth. By incorporating these two metrics, we complement the strictness of EM with more flexible and nuanced evaluations.

[6]https://pypi.org/project/fuzzywuzzy/

	election	leader	of seats won	of national votes	% of national vote	of prefectural votes	% of prefectural vote
0	1956	ichirō hatoyama	61	11356874	39.7%	14353960	48.4%
1	1959	nobusuke kishi	71	12120598	41.2%	15667022	52.0%
2	1962	hayato ikeda	69	16581637	46.4%	17112986	47.1%
3	1965	eisaku satō	71	17583490	47.2%	16651284	44.2%
4	1968	eisaku satō	69	20120089	46.7%	19405546	44.9%
5	1971	eisaku satō	62	17759395	44.5%	17727263	44.0%
6	1974	kakuei tanaka	62	23332773	44.3%	21132372	39.5%
7	1977	takeo fukuda	63	18160061	35.8%	20440157	39.5%
8	1980	masayoshi ōhira	69	23778190	43.3%	24533083	42.5%
9	1983	yasuhiro nakasone	68	16441437	35.3%	19975034	43.2%
10	1986	yasuhiro nakasone	72	22132573	38.58%	26111258	45.07%
11	1989	sōsuke uno	36	17466406	30.70%	15343455	27.32%
12	1992	kiichi miyazawa	68	20528293	45.23%	14961199	33.29%
13	1995	yōhei kōno	46	10557547	25.40%	11096972	27.29%
14	1998	keizō obuchi	44	17033851	30.45%	14128719	25.17%
15	2001	junichiro koizumi	64	22299825	41.04%	21114727	38.57%
16	2004	junichiro koizumi	49	16797686	30.03%	19687954	35.08%
17	2007	shinzō abe	37	16544696	28.1%	18606193	31.35%
18	2010	sadakazu tanigaki	51	14071671	24.07%	19496083	33.38%
19	2013	shinzō abe	65	18460404	34.7%	22681192	42.7%

Figure 11: Example of **TableBench** dataset (Index=8). The question (Q) is **What is the average percentage of national votes won by all leaders in the table?**, the answer (A) is **37.64%**

C.5.1 Fuzzy Matching Score for "`John , Andy`" vs. "`John and Andy`"

Given two candidate answers[7]

$$s_1 = \text{"}\texttt{john , andy}\text{"} \quad (\text{length} = 11)$$
$$s_2 = \text{"}\texttt{john and andy}\text{"} \quad (\text{length} = 13)$$

The similarity score is computed by `fuzz.ratio()`, which defaults to `difflib.SequenceMatcher` when `python-Levenshtein` is not installed. The score is

$$\mathrm{SM}(s_1, s_2) = \frac{2M}{|s_1| + |s_2|} \times 100,$$

where M is the total length of all matching blocks returned by `SequenceMatcher`. For the strings above, the matching blocks are "`john `" (length 5) and "` andy`" (length 5) with whitespace characters counted in the length. Thus, $M = 5 + 5 = 10$, and the final similarity score is

$$\mathrm{SM}(s_1, s_2) = \frac{2 \times 10}{11 + 13} \times 100 = 83.33\%.$$

`fuzz.ratio()` rounds this to 83; dividing by 100 in our post-processing step yields the final value 0.83.

C.5.2 Token-level F1 Score for "`John , Andy`" vs. "`John and Andy`"

Following the SQuAD v1.1 definition, each answer is first lower-cased and split on whitespace to obtain tokens:

$$\text{prediction tokens} := \{\text{"john"}, \text{","}, \text{"andy"}\},$$
$$\text{ground-truth tokens} := \{\text{"john"}, \text{"and"}, \text{"andy"}\}.$$

Overlap. The set intersection contains two common tokens, "`john`" and "`andy`", so

$$n_{\text{same}} = 2.$$

Precision and recall.

$$\text{precision} = \frac{n_{\text{same}}}{|\text{prediction tokens}|} = \frac{2}{3},$$

$$\text{recall} = \frac{n_{\text{same}}}{|\text{ground-truth tokens}|} = \frac{2}{3}.$$

[7] All strings are lower-cased and stripped of leading/trailing whitespace before comparison.

F1 score.

$$F_1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$
$$= \frac{2 \times \frac{2}{3} \times \frac{2}{3}}{\frac{2}{3} + \frac{2}{3}} = \frac{2}{3} \approx 0.67.$$

Thus, the SQuAD-style token-level F1 score for the two example strings is

$$\boxed{F_1 = 0.67}.$$

D Hyper-Parameter Setting

This section reports the experimental results that guided the selection of MATA's hyperparameters, N and θ. Based on these results, MATA sets $N = 3$ and $\theta = 0.1$.

D.1 Debugging Iteration Stop Condition

Unlike CoT, PoT and text2SQL often suffer from execution failures, particularly when encountering atypical table structures or inconsistent value types. To address this issue, we apply a lightweight self-refinement strategy (Madaan et al., 2023) for PoT and text2SQL (rather than a full-fledged LLM-based code debugging(Chen et al., 2024b; Zhong et al., 2024)). Each generated code snippet, along with its execution result, is re-injected into the LLM for iterative correction, up to a maximum of N rounds. Early stopping is applied when refinements show minimal change or execution outcomes remain unchanged, thereby reducing unnecessary computation.

For Python code generated via PoT, refinement termination is determined by computing four similarity metrics including Levenshtein distance (Levenshtein, 1966), difflib similarity (Python Software Foundation, 2024; Ratcliff and Metzener, 1988), Abstract Syntax Tree (AST) similarity (Song et al., 2024), and Opcode-based similarity (Lu, 2019). Refinement is considered complete when the average of these metrics exceeds 0.9, indicating convergence.

The self-refinement process for text2SQL follows a distinct stopping criterion. Unlike Python code, SQL queries tend to undergo minimal changes during refinement and often resist correction when semantic or structural errors are present. Accordingly, we stop the refinement loop for text2SQL either when the generated query remains unchanged across iterations or when previously failing queries execute successfully. Further details are described in Algorithm 4.

Across both PoT and text2SQL, up to N self-refinement rounds may be applied depending on the stability of intermediate outputs. CoT, on the other hand, is excluded from refinement. Empirically, we observe that CoT solutions are stable across repeated generations and rarely benefit from additional passes. To avoid unnecessary LLM calls, we retain a single-step inference for CoT (See Appendix D.3).

At the end of this process, MATA obtains one text-based answer from CoT, along with up to (1+N) candidate outputs each from PoT and text2SQL, including code executions. These are then aggregated and passed to the Confidence Checker for final decision-making. We set $N = 3$, by default, in all experiments.

D.2 The Maximum Number Of Debugging Iteration (N)

Tables 16, 17, 18 and 19 present the iteration at which the debug loop terminates early when the maximum number of iterations is set to $N = 7$. The results show that for nearly all models, debugging for both PoT and text2SQL typically concludes within just one or two iterations. Additional iterations beyond that point generally yield little benefit. An exception is Mistral-7B, which we attribute to its limited debugging capability via self-refinement (Madaan et al., 2023). Based on these findings, we set the default maximum number of iterations to $N = 3$ in order to minimize unnecessary LLM agent calls while still enabling high-quality PoT and text2SQL reasoning through effective debugging.

D.3 CoT Self-Refinement Behavior and Early Stopping Analysis

To assess the inefficiency of the debug iteration loop for CoT, we implemented a CoT Debug Agent (*CDA*) using the same prompt structure as the Python Debug Agent (*PDA*) and SQL Debug Agent (*SDA*) (see Appendix F for prompt details). Like the others, *CDA* employs self-refinement, and we set $N = 7$ as in Appendix D.2 to observe how the CoT solution and answer are revised during the process. The experimental results are shown in Table 20 and Table 21. The results show that in most cases, the iteration stops at $N = 1$, indicating that the initial CoT reasoning generated by *CoTA* is rarely altered by *CDA*. Therefore, to maximize the efficiency of MATA, we chose not to introduce *CDA* alongside *CoTA*.

D.4 Effect of Confidence Threshold (θ) on MATA's Performance

We input the solutions, answers, code, and code execution results generated by CoT, PoT, and text2SQL reasoning into the Confidence Checker (*CC*) to obtain confidence scores for each reasoning path. If the confidence score is sufficiently high, the system selects the reasoning path with the highest score as the final answer without invoking the additional Judge Agent (*JA*). This allows us to obtain high-quality answers while avoiding extra LLM agent calls. The threshold θ, which determines whether a confidence score is deemed sufficiently high, is a tunable hyperparameter. We investigated the effect of varying this hyperparameter.

Figure 12 and Figure 13 show that increasing θ generally leads to a decrease in overall performance. This aligns with the analysis in the main Ablation Study: as θ increases, the final answer is more often determined by JA rather than *CC*. Given that removing *CC* significantly reduced performance in the ablation study, we infer that *CC* is more effective than JA in selecting the correct answer. Thus, increasing θ leads to more frequent (and less effective) reliance on JA, resulting in lower performance. When θ is set below 0.1, we observed no significant change in performance, suggesting that confidence scores below 0.1 are not meaningful. Therefore, we conclude that setting $\theta = 0.1$ is an appropriate choice.

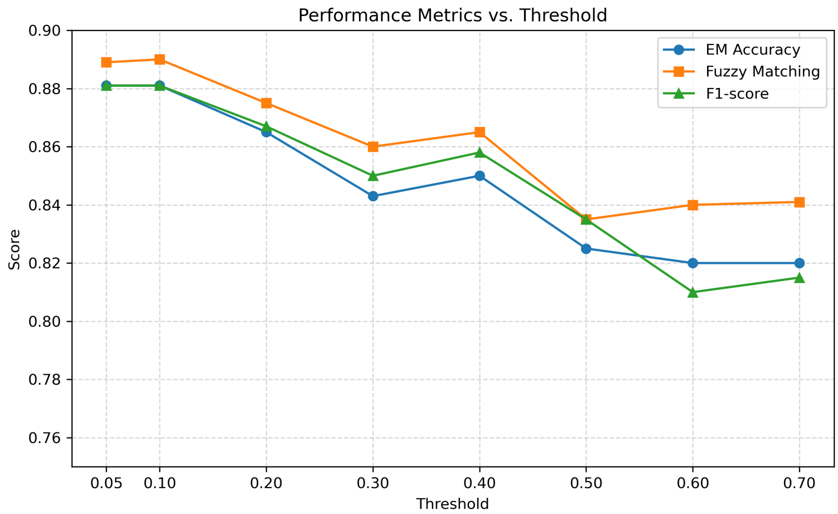

Figure 12: Performance scores (EM, fuzzy, F1) averaged over 10 models on the **Penguins in a Table**. As the confidence threshold θ increases, performance gradually decreases.

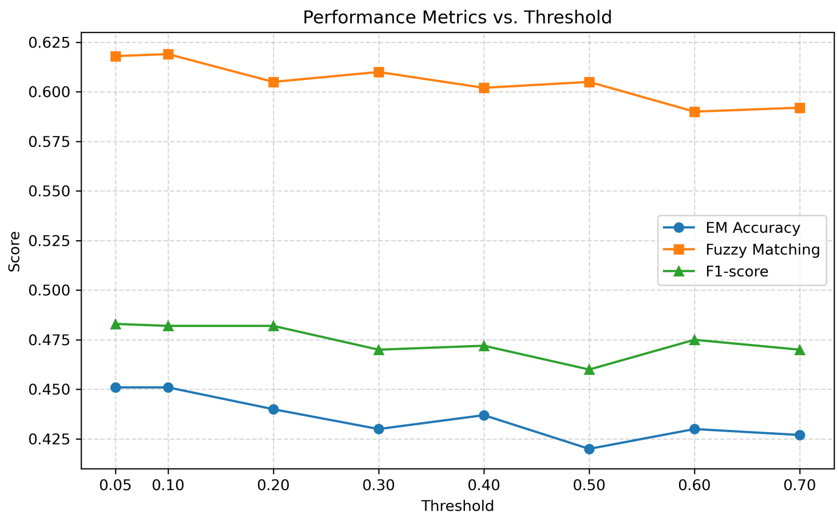

Figure 13: Performance scores (EM, fuzzy, F1) averaged over 10 models on the **TableBench**. Higher values of θ result in more frequent fallback to the Judge Agent, leading to degraded performance.

D.5 Confidence Threshold Analysis

We further analyze whether the Confidence Checker (*CC*) provides a reliable signal for cost-aware early exiting. For each query, *CC* assigns three confidence scores to the candidate answers produced by CoT, PoT, and text2SQL, denoted as C, P, and TS, respectively. We define $s = \max(\mathrm{C}, \mathrm{P}, \mathrm{TS})$ as the confidence of the answer that would be selected by *CC*. If $s > \theta$, MATA accepts the highest-scoring candidate and skips the *JA*; otherwise, the query is treated as uncertain and passed to *JA*.

The goal of *CC* is not to produce perfectly calibrated probabilities, but to serve as a lightweight gating and ranking module that decides when *JA* can be safely skipped. We therefore evaluate whether s separates cases where the *CC*-selected answer is likely to be correct from cases where additional verification is needed. Following the threshold sweep in Appendix D.4, we use $\theta = 0.1$: thresholds below 0.1 showed little effect, while larger thresholds increased fallback to *JA* and generally degraded final performance.

Metric	llama3.2-3b	qwen2.5-32b	GPT-4o
ROC-AUC ↑	0.875	0.886	0.869
ECE_{10} ↓	0.163	0.154	0.162
JA-skip@$\theta = 0.1$ ↑	0.577	0.698	0.730
EM($s > \theta$) ↑	0.545	0.698	0.700
EM($s \leq \theta$) ↑	0.092	0.177	0.267

Table 9: Confidence-score analysis on **TableBench** ($n = 693$). *CC* outputs confidence scores C, P, and TS for CoT, PoT, and text2SQL, and we define $s = \max(\mathrm{C}, \mathrm{P}, \mathrm{TS})$. ROC-AUC evaluates whether s ranks correct *CC*-selected answers above incorrect ones. ECE_{10} is the 10-bin expected calibration error. *JA*-skip@$\theta = 0.1$ is the fraction of queries where $s > \theta$, so *JA* is skipped. EM($s > \theta$) and EM($s \leq \theta$) report the EM of the *CC*-selected answer in the high- and low-confidence partitions.

Table 9 reports this analysis on **TableBench**. ROC-AUC measures how well s ranks correct *CC*-selected answers above incorrect ones. ECE_{10} is the expected calibration error computed with 10 bins, where lower values indicate better calibration. *JA*-skip@$\theta = 0.1$ denotes the fraction of examples for which $s > \theta$, meaning that MATA accepts the *CC*-selected answer without invoking *JA*. Finally, EM($s > \theta$) and EM($s \leq \theta$) report the exact-match accuracy of the *CC*-selected answer in the high- and low-confidence partitions, respectively.

Across representative small, large open-source, and closed-source backbones, *CC* achieves strong ROC-AUC scores of 0.869–0.886. More importantly, the high-confidence group consistently obtains much higher EM than the low-confidence group: for example, 0.698 vs. 0.177 with qwen2.5-32B and 0.700 vs. 0.267 with GPT-4o. At the same time, MATA skips *JA* for 57.7–73.0% of **TableBench** queries. These results support the use of $\theta = 0.1$ as a practical early-exit threshold: it identifies many cases where *CC* alone is reliable, while reserving *JA* for genuinely uncertain cases.

E Additional Experiments Results

E.1 Detailed Results of the Ablation Study

Table 12 and Table 13 provide the exact numerical data corresponding to Figure 2 in the main text.

E.2 Reduction in LLM Agent Calls via Confidence Checker (*CC*) and Scheduler (*Sch*)

This section reports the experimental results on the reduction in LLM agent calls achieved by the *CC* and *Sch* modules, as discussed in the Ablation Study section of the main text. See Tables 22 and 23 for detailed results. On the **Penguins in a Table** dataset, applying the *CC* results in approximately a 95.8% reduction in LLM agent calls across all 10 models, while using *Sch* yields a reduction of about 14.59%. On the **TableBench** dataset, the CC reduces LLM agent calls by approximately 60.6%, and Sch by about 7.62%, again averaged across all 10 models. Additionally, both *CC* and *Sch* were found to be more effective with large LLMs than with smaller ones. This suggests that these modules become even more beneficial in settings where large LLMs—incurring higher inference costs—are used as backbones.

E.3 Additional TableBench Results with ROUGE-L and Protocol Clarification

Model	TabLaP	SynTQA	MixSC	MATA
llama3.2-3b	0.207	0.138	0.218	**0.417**
mistral-7b	0.219	0.281	0.234	**0.353**
phi4-mini-3.8b	0.226	0.288	0.284	**0.334**
qwen2.5-3b	0.244	0.263	0.241	**0.346**
qwen2.5-7b	0.138	0.342	0.234	**0.428**
mistral-small-24b	0.363	0.447	0.421	**0.637**
cogito-32b	0.521	0.484	0.514	**0.633**
qwen2.5-32b	0.380	0.446	0.394	**0.634**
GPT-4o	0.630	0.514	0.580	**0.654**
Claude-3.7-Sonnet	0.666	0.545	**0.690**	0.684
Average	0.359	0.375	0.381	**0.512**

Table 10: ROUGE-L results on the TableBench entity-answer subset used for the main evaluation. Higher is better.

TableBench contains heterogeneous output formats, including short entity/numeric answers, sentence-level descriptive outputs, and chart-generation tasks. Since our target task is entity-type TableQA, the main paper evaluates on the entity-answer subset (693/886 examples), where discrete-answer comparison is meaningful. The excluded subsets correspond to Descriptive Analysis, Anomaly Detection, Causal Analysis, and Chart Generation. To improve comparability with prior work, we additionally report ROUGE-L on the same subset in Table 10.

E.4 Additional In-Distribution Evaluation on WikiTQ, TabMWP, and TabFact

While the main paper emphasizes out-of-distribution evaluation on **TableBench** and **Penguins in a Table**, we additionally report results on WikiTQ, TabMWP, and TabFact to verify that the gains of MATA are not limited to distribution-shift settings. Table 11 reports full-test Exact Match results with Llama3.2-3B. In this setting, MATA remains competitive and generally strong on all three benchmark families.

llama3.2-3b	WikiTQ (4,344)	TabMWP (7,686)	TabFact (12,779)
TabLaP	0.220	0.171	0.599
SynTQA	0.144	0.228	0.451
MixSC	0.232	0.216	0.612
MATA	**0.535**	**0.713**	**0.688**

Table 11: Exact Match results on the full test splits of WikiTQ, TabMWP, and TabFact using llama3.2-3b.

E.5 Additional Baseline Comparisons

We provide additional comparisons with related TableQA and multi-agent baselines that were not included in the main result tables due to differences in implementation assumptions and supported backbones. We consider two groups of baselines. First, we compare with ReAcTable and Chameleon, whose official implementations are designed around GPT-series APIs. Second, we compare with AutoPrep, a multi-agent framework for question-aware data preparation in TableQA.

GPT-based tool-use baselines. ReAcTable (Zhang et al., 2024c) extends the ReAct paradigm for TableQA by integrating LLM reasoning with external SQL and Python execution. It decomposes complex questions into iterative reasoning steps, executes code to obtain intermediate results, and uses these results to guide subsequent reasoning. Chameleon (Lu et al., 2023a) is a plug-and-play compositional reasoning framework that uses an LLM-based planner to orchestrate external tools such as Python programs and other modules.

	w/o sch			*w/o CC*			*w/o JA*			*w/o FM*			**MATA**		
Models	EM	fuzzy	F1	EM	fuzzy	F1	EM	fuzzy	F1	EM	fuzzy	F1	EM	fuzzy	F1
llama3.2-3b	**0.736**	**0.766**	**0.736**	0.549	0.591	0.557	0.715	0.737	0.715	**0.736**	**0.766**	**0.736**	**0.736**	**0.766**	**0.736**
mistral-7b	**0.875**	**0.893**	**0.875**	0.625	0.667	0.643	0.861	0.879	0.861	0.861	0.880	0.861	0.861	0.880	0.861
phi4-mini-3.8b	0.826	0.852	0.826	0.590	0.648	0.590	**0.833**	**0.863**	**0.833**	0.819	0.847	0.819	0.819	0.847	0.819
qwen2.5-3b	**0.882**	**0.896**	**0.882**	0.688	0.715	0.688	0.854	0.872	0.854	0.868	0.883	0.868	0.868	0.883	0.868
qwen2.5-7b	0.931	0.939	0.931	0.854	0.862	0.854	**0.951**	**0.955**	**0.951**	**0.951**	**0.955**	**0.951**	**0.951**	**0.955**	**0.951**
mistral-small-24b	**0.917**	**0.917**	**0.917**	0.875	0.875	0.875	0.896	0.896	0.896	0.896	0.896	0.896	0.896	0.896	0.896
cogito-32b	**0.903**	**0.903**	**0.903**	0.875	0.875	0.875	**0.903**	**0.903**	**0.903**	**0.903**	**0.903**	**0.903**	**0.903**	**0.903**	**0.903**
qwen2.5-32b	**0.931**	**0.931**	**0.931**	0.882	0.882	0.882	0.917	0.917	0.917	0.917	0.917	0.917	0.917	0.917	0.917
GPT-4o	**0.944**	**0.944**	**0.944**	0.882	0.886	0.887	0.903	0.903	0.903	0.903	0.903	0.903	0.903	0.903	0.903
Claude-3.7-Sonnet	**0.958**	**0.958**	**0.958**	0.903	0.907	0.907	0.951	0.951	0.951	0.951	0.951	0.951	0.951	0.951	0.951
Average	**0.890**	**0.900**	**0.890**	0.774	0.793	0.777	0.876	0.886	0.877	0.881	0.890	0.881	0.881	0.890	0.881

Table 12: Ablation study results on the **Penguins in a Table** benchmark. We report Exact Match(EM) accuracy, fuzzy matching, and F1 scores for each model. Bold indicates the best performance; underlined scores are the second best.

	w/o sch			*w/o CC*			*w/o JA*			*w/o FM*			**MATA**		
Models	EM	fuzzy	F1	EM	fuzzy	F1	EM	fuzzy	F1	EM	fuzzy	F1	EM	fuzzy	F1
llama3.2-3b	**0.359**	**0.571**	**0.388**	0.235	0.471	0.265	0.354	0.554	0.374	0.352	0.557	0.379	0.354	0.563	0.381
mistral-7b	0.293	0.471	**0.321**	0.224	0.421	0.253	0.289	0.455	0.314	0.281	0.454	0.306	**0.294**	**0.473**	**0.321**
phi4-mini-3.8b	0.271	0.458	**0.295**	0.201	0.398	0.221	0.268	**0.460**	0.294	0.270	0.440	0.288	**0.273**	0.457	**0.295**
qwen2.5-3b	**0.293**	0.469	**0.317**	0.208	0.395	0.237	0.284	0.465	0.309	0.291	0.458	0.312	0.291	**0.471**	**0.317**
qwen2.5-7b	**0.354**	**0.557**	**0.394**	0.343	0.553	0.386	0.349	0.553	0.387	0.349	0.530	0.379	**0.354**	**0.557**	0.393
mistral-small-24b	**0.573**	0.722	0.604	0.540	0.702	0.576	0.532	0.697	0.568	**0.573**	0.715	0.601	**0.573**	**0.723**	**0.605**
cogito-32b	**0.577**	**0.723**	**0.609**	0.514	0.691	0.552	0.564	0.713	0.596	0.571	0.711	0.600	**0.577**	**0.723**	**0.609**
qwen2.5-32b	0.573	0.718	0.603	0.554	0.710	0.585	0.544	0.699	0.577	0.576	0.712	0.601	**0.577**	**0.721**	**0.607**
GPT-4o	0.587	0.735	0.622	0.586	0.733	0.624	0.584	0.731	0.616	0.593	0.735	0.625	**0.595**	**0.740**	**0.629**
Claude-3.7-Sonnet	**0.620**	**0.767**	**0.666**	0.587	0.752	0.651	0.612	0.763	0.656	0.618	0.750	0.657	**0.620**	0.764	0.664
Average	0.450	**0.619**	**0.482**	0.399	0.583	0.435	0.438	0.609	0.469	0.447	0.606	0.475	**0.451**	**0.619**	**0.482**

Table 13: Ablation study results on the **TableBench** benchmark. Bold and underline follow Table 12.

The official implementations of both baselines are designed to work with GPT-series models. Accordingly, we evaluate ReAcTable, Chameleon, and MATA using GPT-4o as the shared backbone. As shown in Table 14, MATA substantially outperforms both baselines on TableBench, where tables and questions are more complex. On Penguins in a Table, Chameleon remains competitive and obtains a slightly higher F1 score, while MATA achieves the best EM and fuzzy scores.

AutoPrep comparison. AutoPrep (Fan et al., 2025) studies a complementary problem to MATA: question-aware data preparation for TQA. Given a table and a natural language question, AutoPrep first plans high-level data-preparation operations, then generates low-level executable code for these operations, and finally executes and debugs the code to produce a prepared table before answering. Thus, unlike MATA, which focuses on adaptive reasoning-path orchestration and answer selection, AutoPrep focuses on preparing the table so that downstream TQA methods can answer more reliably.

Because AutoPrep was originally evaluated under different benchmark and backbone settings, we treat this comparison as an additional adaptation study rather than a primary baseline comparison. We evaluate AutoPrep and MATA on our benchmarks using representative Qwen2.5 backbones and the same evaluation metrics. As shown in Table 15, MATA performs better in three of the four benchmark–backbone settings. The exception is TableBench with qwen2.5-3B, where AutoPrep achieves higher EM and F1, while MATA obtains slightly higher fuzzy matching. Overall, these additional results suggest that MATA's gains are not limited to the three main baselines, while also showing that question-aware data preparation can be a competitive alternative in some low-resource backbone settings.

E.6 Analysis about Table Size

We also aim to examine how each method is affected by the size of the input table. As shown in Figure 14, CoT's EM accuracy gradually de-

	Penguins in a Table			*TableBench*		
Method	**EM**	**fuzzy**	**F1**	**EM**	**fuzzy**	**F1**
ReAcTable	0.653	0.705	0.738	<u>0.338</u>	<u>0.471</u>	<u>0.402</u>
Chameleon	<u>0.882</u>	<u>0.902</u>	**0.905**	0.267	0.328	0.273
MATA	**0.903**	**0.903**	<u>0.903</u>	**0.595**	**0.740**	**0.629**

Table 14: Evaluation results on the **Penguins in a Table** (left) and **TableBench** (right) datasets. All methods use GPT-4o as the backbone. Bold indicates the best performance; underlined scores are the second best.

Benchmark	Model	AutoPrep			MATA		
		EM	fuzzy	F1	EM	fuzzy	F1
Penguins in a Table	qwen2.5-3b	0.785	0.823	0.785	**0.868**	**0.883**	**0.868**
Penguins in a Table	qwen2.5-7b	0.896	0.898	0.896	**0.951**	**0.955**	**0.951**
TableBench	qwen2.5-3b	**0.323**	0.467	**0.346**	0.291	**0.471**	0.317
TableBench	qwen2.5-7b	0.323	0.459	0.349	**0.354**	**0.557**	**0.393**

Table 15: Additional comparison with AutoPrep, a question-aware data-preparation framework for TQA, on representative Qwen2.5 backbones.

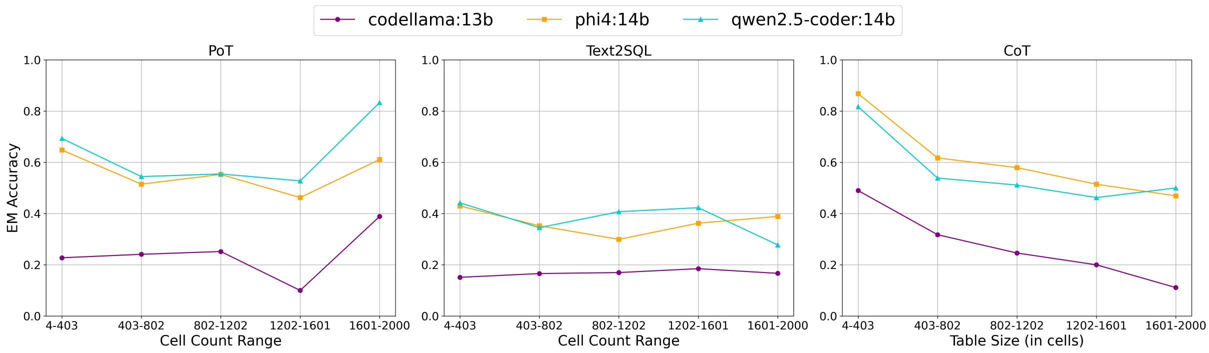

Figure 14: EM accuracy trends across varying table sizes. The horizontal axis indicates cell count ranges, and the vertical axis shows the average EM accuracy achieved by each method, illustrating how table size affects results.

clines as the table size increases, whereas PoT and text2SQL are less sensitive to the table size. These results confirm that the table size significantly influences accuracy across methods, as shown in previous studies (Liu et al., 2024; Zhang et al., 2024a).

F Prompts for Experiments and Training Data Generation

F.1 CoT Prompt

CoT Agent (*CoTA*) Prompt

Read the following table and then write solution texts to answer a question:
Name	Number of coins
—–	——
Braden	76
Camilla	94
Rick	86
Mary	84
Hector	80
Devin	83
Emily	82
Avery	87
Question: Some friends discussed the sizes of their coin collections. What is the mean of the numbers?
Return a query for the solution and answer with two keys: solution and answer. Respond using JSON only.

{{'solution': "To find the mean, we sum up the numbers of coins for all the individuals and then divide the total by the number of individuals. Step 1: Add all the numbers of coins: 76 + 94 + 86 + 84 + 80 + 83 + 82 + 87 = 672 Step 2: Count the number of individuals: There are 8 individuals. Step 3: Calculate the mean: Mean = Total sum of coins / Number of individuals = 672/8 = 84", 'answer': 84}}

Read the following table and then write solution texts to answer a question:
Price	Quantity demanded	Quantity supplied
—	——-	——-
$155	22,600	5,800
$275	20,500	9,400
$395	18,400	13,000
$515	16,300	16,600
$635	14,200	20,200
Question: Look at the table. Then answer the question. At a price of $155, is there a shortage or a surplus?
Return a query for the solution and answer with two keys: solution and answer. Respond using JSON only.

{{'solution': "To determine if there is a 'shortage' or a 'surplus', we compare the 'quantity demanded' and the 'quantity supplied' at the given price. Step 1: Identify the values from the table for $155: - Quantity demanded = 22,600 - Quantity supplied = 5,800 Step 2: Calculate the difference: Shortage or Surplus = Quantity demanded - Quantity supplied = 22,600 - 5,800 = 16,800 Step 3: Determine the situation: Since the quantity demanded is greater than the quantity supplied, there is a 'shortage' of 16,800 units.", 'answer': 'shortage'}}

Read the following table and then write solution texts to answer a question:
Name	Number
—–	—
Samir	7
Kristen	4
Dakota	7
Jamie	8
Maggie	9
Question: Samir's class recorded how many cans of food each student collected for their canned food drive. What is the median of the numbers?
Return a query for the solution and answer with two keys: solution and answer. Respond using JSON only.

{{'solution': "To find the median, we need to arrange the numbers in order from smallest to largest and then identify the middle number. Step 1: Arrange the numbers: 4, 7, 7, 8, 9 Step 2: Identify the middle number. Since there are 5 numbers (an odd count), the middle number is the third number. So the median is 7.", 'answer': 7}}

Read the following table and then write solution texts to answer a question:
Toy	Price
—–	—
toy boat	$5.54
toy guitar	$8.23
set of juggling balls	$5.01
trivia game	$8.18

| jigsaw puzzle | $5.30 |
| toy dinosaur | $3.00 |
Question: Lorenzo has $13.50. Does he have enough to buy a toy guitar and a set of juggling balls?
Return a query for the solution and answer with two keys: solution and answer. Respond using JSON only.

{{'solution': "To determine if Lorenzo can afford both the toy guitar and the set of juggling balls, we need to calculate their combined cost and compare it to Lorenzo's available money. Step 1: Identify the prices: - Toy guitar = $8.23 - Set of juggling balls = $5.01 Step 2: Calculate the total cost: Total cost = 8.23 + 5.01 = 13.24 Step 3: Compare the total cost to Lorenzo's money. Lorenzo has $13.50, and the total cost is $13.24. Since $13.24 is less than $13.50, Lorenzo does have enough money.", 'answer': 'Yes'}}

{***query***}

F.2 PoT Prompt

PoT Agent (*PoTA*) Prompt

Read the following table and then write Python code with pandas to answer a question:
import pandas as pd
data = {
'Name': ['Braden', 'Camilla', 'Rick', 'Mary', 'Hector', 'Devin', 'Emily', 'Avery'],
'Number of coins': [76, 94, 86, 84, 80, 83, 82, 87]
}
df = pd.DataFrame(data)
Question: Some friends discussed the sizes of their coin collections. What is the mean of the numbers?
You don't need to reprint pre-written code like 'import pandas as pd', 'data = {...}', or 'df = pd.DataFrame(data)'. That code will be provided separately, so just give me the code that processes 'data' and 'df'.
Return a query for the 'python code with pandas which return ans' with one key: code. Respond using JSON only.

{{'code' : "'# Calculate the mean of the 'Number of coins' column
mean_coins = df['Number of coins'].mean()
ans = mean_coins"'}}

Read the following table and then write Python code with pandas to answer a question:
import pandas as pd
data = {
'Price': [155, 275, 395, 515, 635],
'Quantity demanded': [22600, 20500, 18400, 16300, 14200],
'Quantity supplied': [5800, 9400, 13000, 16600, 20200]
}
df = pd.DataFrame(data)
Question: Look at the table. Then answer the question. At a price of $155, is there a shortage or a surplus?
You don't need to reprint pre-written code like 'import pandas as pd', 'data = {...}', or 'df = pd.DataFrame(data)'. That code will be provided separately, so just give me the code that processes 'data' and 'df'.
Return a query for the 'python code with pandas which return ans' with one key: code. Respond using JSON only.

{{'code' : "'# Filter the row where the price is $155
price_155 = df[df['Price'] == 155]
# Calculate shortage or surplus
quantity_demanded = price_155['Quantity demanded'].values[0]
quantity_supplied = price_155['Quantity supplied'].values[0]
if quantity_demanded > quantity_supplied:
ans = 'shortage'
else:
ans = 'surplus' "'}}

Read the following table and then write Python code with pandas to answer a question:

import pandas as pd
data = {
'Name': ['Samir', 'Kristen', 'Dakota', 'Jamie', 'Maggie'],
'Cans collected': [7, 4, 7, 8, 9]
}
df = pd.DataFrame(data)
Question: Samir's class recorded how many cans of food each student collected for their canned food drive. What is the median of the numbers?
You don't need to reprint pre-written code like 'import pandas as pd', 'data = {...}', or 'df = pd.DataFrame(data)'. That code will be provided separately, so just give me the code that processes 'data' and 'df'.
Return a query for the 'python code with pandas which return ans' with one key: code. Respond using JSON only.

{{'code' : "'# Calculate the median of the 'Cans collected' column
median_cans = df['Cans collected'].median()
ans = median_cans"'}}

Read the following table and then write Python code with pandas to answer a question:
import pandas as pd
data = {
'Toy': ['toy boat', 'toy guitar', 'set of juggling balls', 'trivia game', 'jigsaw puzzle', 'toy dinosaur'],
'Price': [5.54, 8.23, 5.01, 8.18, 5.30, 3.00]
}
df = pd.DataFrame(data)
Question: Lorenzo has $13.50. Does he have enough to buy a toy guitar and a set of juggling balls?
You don't need to reprint pre-written code like 'import pandas as pd', 'data = {...}', or 'df = pd.DataFrame(data)'. That code will be provided separately, so just give me the code that processes 'data' and 'df'.
Return a query for the 'python code with pandas which return ans' with one key: code. Respond using JSON only.

{{'code' : "'# Lorenzo's total money
total_money = 13.50

# Filter the prices of 'toy guitar' and 'set of juggling balls'
selected_items = df[df['Toy'].isin(['toy guitar', 'set of juggling balls'])]

# Calculate the total cost
total_cost = selected_items['Price'].sum()

# Determine if Lorenzo has enough money
if total_money >= total_cost:
ans = "yes"
else:
ans = "no"
"'}}

{***query***}

Python Debug Agent (*PDA*) Prompt

You are an expert in reviewing and correcting Python code designed to solve questions about tables.
Review the query, the previous pandas code written to address it, and its execution results to identify any parts that need correction.

query
Read the following table and then write Python code with pandas to answer a question:
import pandas as pd
data = {
'Toy': ['toy boat', 'toy guitar', 'set of juggling balls', 'trivia game', 'jigsaw puzzle', 'toy dinosaur'],
'Price': [5.54, 8.23, 5.01, 8.18, 5.30, 3.00]
}
df = pd.DataFrame(data)
Question: What is the average price of toys that cost more than $5?
You don't need to reprint pre-written code like 'import pandas as pd', 'data = {...}', or 'df = pd.DataFrame(data)'. That code will be provided separately, so just

give me the code that processes 'data' and 'df'.
Return a query for the python code with pandas which return ans with one key: code. Respond using JSON only. (You must return the value with 'ans')

Previous Code:
# The following 4 toys are included :
df_previous = df[df['Toy'].isin([
'toy boat',
'toy guitar',
'set of juggling balls',
'toy dinosaur'
])]
# Summing the prices
total_previous = df_previous['Price'].sum()
# Counting the number of toys
count_previous = len(df_previous)
# Calculating the average
ans = total_previous / count_previous

Previous Execution Result:
5.445

Return a query for 'corrected python code with pandas which return ans' with one key: code. Respond using JSON only. (You must return the value with 'ans')
{{'code' : "'
# The toys that will actually be included among those priced greater than $5 are the toy boat, toy guitar, set of juggling balls, trivia game, and jigsaw puzzle.
df_corrected = df[df['Toy'].isin([
'toy boat',
'toy guitar',
'set of juggling balls',
'trivia game',
'jigsaw puzzle'
])]
# Summing the prices
total_corrected = df_corrected['Price'].sum()
# Counting the number of toys
count_corrected = len(df_corrected)
# Calculating the average
ans = total_corrected / count_corrected
"'}}

query
{***query***}

Previous Code:
{***code***}

Previous Execution Result:
{***execution_result***}

Return a query for 'corrected python code with pandas which return ans' with one key: code. Respond using JSON only. (You must return the value with 'ans')

F.3 Text2SQL Prompt

text2SQL Agent (*t2SA*) Prompt

Read the following table and then write SQL code to answer the question:
– Table: coin_collection
– Columns:
– Name (TEXT)
– Number_of_coins (INTEGER)
–
– Rows:
– Braden | 76
– Camilla | 94
– Rick | 86
– Mary | 84
– Hector | 80
– Devin | 83
– Emily | 82
– Avery | 87
Question: Some friends discussed the sizes of their coin collections. What is the mean of the numbers?
Return a query for the 'SQL code' with one key: code. Respond using JSON only.
{{'code' : "'
SELECT AVG(Number_of_coins) AS answer
FROM coin_collection;
"'}}

Read the following table and then write SQL code to answer the question:
– Table: market
– Columns:
– Price (INTEGER)
– Quantity_demanded (INTEGER)

–   Quantity_supplied (INTEGER)
–
– Rows:
–   155 | 22600 | 5800
–   275 | 20500 | 9400
–   395 | 18400 | 13000
–   515 | 16300 | 16600
–   635 | 14200 | 20200
Question: Look at the table. Then answer the question. At a price of $155, is there a shortage or a surplus?
Return a query for the 'SQL code' with one key: code. Respond using JSON only.
{{'code' : "'
SELECT
CASE
WHEN Quantity_demanded > Quantity_supplied THEN 'shortage'
ELSE 'surplus'
END AS answer
FROM market
WHERE Price = 155;
"'}}

Read the following table and then write SQL code to answer the question:
– Table: can_collection
– Columns:
–   Name (TEXT)
–   Cans_collected (INTEGER)
–
– Rows:
–   Samir | 7
–   Kristen | 4
–   Dakota | 7
–   Jamie | 8
–   Maggie | 9
Question: Samir's class recorded how many cans of food each student collected for their canned food drive. What is the median of the numbers?
Return a query for the 'SQL code' with one key: code. Respond using JSON only.
{{'code' : "'
SELECT PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY Cans_collected) AS answer
FROM can_collection;
"'}}

Read the following table and then write SQL code to answer the question:
– Table: toys
– Columns:
–   Toy (TEXT)
–   Price (DECIMAL)
–
– Rows:
–   toy boat | 5.54
–   toy guitar | 8.23
–   set of juggling balls | 5.01
–   trivia game | 8.18
–   jigsaw puzzle | 5.30
–   toy dinosaur | 3.00
Question: Lorenzo has $13.50. Does he have enough to buy a toy guitar and a set of juggling balls?
Return a query for the 'SQL code' with one key: code. Respond using JSON only.
{{'code' : "'
SELECT
CASE
WHEN SUM(Price) <= 13.50 THEN 'yes'
ELSE 'no'
END AS answer
FROM toys
WHERE Toy IN ('toy guitar', 'set of juggling balls');
"'}}

{***query***}

SQL Debug Agent (*SDA*) Prompt

You are an expert in reviewing and correcting SQL code designed to solve questions about tables.
Review the query, the previous SQL code written to address it, and its execution results to identify any parts that need correction.

query
Read the following table and then write SQL code to answer a question:
– Table: toys
– Columns:
–   Toy (TEXT)
–   Price (DECIMAL)

– Rows:
– toy boat | 5.54
– toy guitar | 8.23
– set of juggling balls | 5.01
– trivia game | 8.18
– jigsaw puzzle | 5.30
– toy dinosaur | 3.00
Question: What is the average price of toys that cost more than $5?
Return a query for the 'SQL code' with one key: code. Respond using JSON only.

Previous Code:
– The following 4 toys are included:
SELECT AVG(Price) AS ans
FROM toys
WHERE Toy IN ('toy boat', 'toy guitar', 'set of juggling balls', 'toy dinosaur');

Previous Execution Result:
5.445

Return a query for 'corrected SQL code' with one key: code. Respond using JSON only.
{{'code' : "'
– The toys that will actually be included among those priced greater than $5 are:
– toy boat, toy guitar, set of juggling balls, trivia game, jigsaw puzzle.
SELECT AVG(Price) AS ans
FROM toys
WHERE Price > 5;
"'}}

query
{***query***}

Previous Code:
{***code***}

Previous Execution Result:
{***execution_result***}

Return a query for 'corrected SQL code' with one key: code. Respond using JSON only.

Format Matcher (*FM*) Prompt

You are a helpful AI that extracts the key entities from a specific sentence.
For a given question, someone has provided an answer.
However, the answer is too long and verbose, so you need to condense it into a few short entities. Summarize the answer into a few words as entities.

Return a query for the solution and answer with two keys: Justification and Answer. Respond using JSON only.
- Question: Which country had the highest British exports in 1950, and how does it compare to its British exports in 1942?
- Answer: Sweden had the highest British exports in 1950 with 165.5 million Pounds, which was 72.3 million Pounds higher than its 1942 value of 93.2 million Pounds.
Return the final output strictly in the following JSON format.
{{'Extracted_Answer' : 'Sweden, 72.3'}}

Return a query for the solution and answer with two keys: Justification and Answer. Respond using JSON only.
- Question : {***Question***}
- Answer : {***Answer***}
Return the final output strictly in the following JSON format.

F.4 Judge Agent Prompt

The Judge Agent (*JA*) is implemented as a tool-calling LLM agent. It receives the table, the question, and the JSON-formatted reasoning outputs from the CoT, PoT, and text2SQL branches. *JA* can also call the confidence-score tool, which returns the *CC* scores for the three reasoning paths. The prompt used for *JA* is shown below.

Judge Agent (*JA*) Prompt

System:
You are a helpful AI that provides appropriate answers to users' questions.

Your task is to output both the Answer to a Question about a given Table and the Justification for that Answer.

The provided Table is as follows:
{***table***}
The Question about the Table is as follows:
{***question***}
The user will also provide 'the results of three different reasoning approaches performed by another AI to answer the Question about the Table.' You should use these as references to produce the optimal Answer.
These reasoning results will be provided in JSON format. The three reasoning approaches are detailed below:
1. reasoning using pandas (PoT): This refers to reasoning about the Question using Python's pandas library, including the generated code and its execution results. Here, N indicates the number of times the code was refined. A larger N means multiple refinements were attempted. For example, code with N=1 was obtained by refining the code with N=0.
2. reasoning using SQL (textSQL): This refers to reasoning about the Question using SQL, including the generated code and its execution results. As with PoT, N indicates the number of refinement iterations.
3. Text-based reasoning (CoT): This refers to reasoning about the Question purely in natural language and its resulting answer. CoT does not involve any refinements; therefore, N is always 0.
You must use the provided reasoning results to determine the best possible final answer. Make sure to use the 'check_confidence_scores' tool for obtaining confidence scores for each approach. If these results alone are insufficient or ambiguous, you may optionally use the 'check_confidence_scores' tool to obtain confidence scores for each reasoning result. However, do not rely on these confidence scores as 100% accurate–they are only meant for reference. You may choose whether or not to use this tool.
Return the final output strictly in the following JSON format: {{'Justification': 'To calculate the rate of change, subtract the number of boxes sold on Wednesday (27) from Thursday (23): 23 - 27 = -4. Since the change occurred over 1 day, the rate of change is -4 boxes per day.', 'Answer': '-4'}}
Human: {***results_all***}

Agent scratchpad: {***agent_scratchpad***}

-	Model	N = 1	N = 2	N = 3	N = 4	N = 5	N = 6	N = 7
small LLM	llama3.2-3b	54	63	15	5	2	2	3
	mistral-7b	24	16	28	22	19	13	22
	phi4-mini-3.8b	107	14	7	3	0	0	13
	qwen2.5-3b	78	35	14	6	2	2	7
	qwen2.5-7b	79	28	11	2	0	1	23
large LLM	mistral-small-24b	0	122	9	2	2	0	9
	cogito-32b	116	9	5	0	0	0	14
	qwen2.5-32b	105	31	4	2	0	0	2
	GPT-4o	82	46	11	3	2	0	0
	Claude-3.7-Sonnet	127	13	4	0	0	0	0

Table 16: Early stopping iteration distribution for PoT debugging on the **Penguins in a Table** dataset with a maximum of $N = 7$ iterations. Most models converge within 1–2 iterations, demonstrating limited benefit from further refinement.

-	Model	N = 1	N = 2	N = 3	N = 4	N = 5	N = 6	N = 7
small LLM	llama3.2-3b	60	247	160	77	37	6	106
	mistral-7b	54	69	54	30	19	12	455
	phi4-mini-3.8b	203	237	78	30	12	5	128
	qwen2.5-3b	248	192	95	39	15	10	94
	qwen2.5-7b	380	132	58	24	7	5	87
large LLM	mistral-small-24b	12	492	122	41	8	3	15
	cogito-32b	449	190	30	7	3	0	14
	qwen2.5-32b	356	223	80	8	7	2	17
	GPT-4o	504	131	15	13	10	8	12
	Claude-3.7-Sonnet	530	102	16	14	11	8	12

Table 17: Early stopping iteration distribution for PoT debugging on the **TableBench** dataset. Compared to Table 16, more iterations are sometimes required due to the increased complexity of questions and tables.

-	Model	N = 0	N = 1	N = 2	N = 3	N = 4	N = 5	N = 6	N = 7
small LLM	llama3.2-3b	133	5	1	0	0	0	0	5
	mistral-7b	131	8	0	5	0	0	0	0
	phi4-mini-3.8b	128	8	4	0	0	0	0	4
	qwen2.5-3b	131	8	1	0	1	0	0	3
	qwen2.5-7b	140	3	0	1	0	0	0	0
large LLM	mistral-small-24b	141	2	1	0	0	0	0	0
	cogito-32b	142	2	0	0	0	0	0	0
	qwen2.5-32b	143	1	0	0	0	0	0	0
	GPT-4o	144	0	0	0	0	0	0	0
	Claude-3.7-Sonnet	144	0	0	0	0	0	0	0

Table 18: Early stopping iteration distribution for text2SQL debugging on the **Penguins in a Table** dataset. Most models terminate at iteration 0 or 1, indicating that debugging typically halts early for simple tasks. Unlike PoT, text2SQL begins counting from $N = 0$ because, if no execution error occurs at the initial step, the loop stops without further debugging.

-	Model	N = 0	N = 1	N = 2	N = 3	N = 4	N = 5	N = 6	N = 7
small LLM	llama3.2-3b	252	173	139	43	23	6	2	55
	mistral-7b	249	42	30	25	25	10	11	301
	phi4-mini-3.8b	259	71	71	62	43	23	14	150
	qwen2.5-3b	284	78	88	71	39	19	10	104
	qwen2.5-7b	119	366	96	14	5	3	2	88
large LLM	mistral-small-24b	343	145	77	30	19	5	3	71
	cogito-32b	422	107	70	33	10	5	1	45
	qwen2.5-32b	370	119	56	40	24	6	2	76
	GPT-4o	460	110	61	16	14	12	8	12
	Claude-3.7-Sonnet	464	176	28	7	6	5	3	4

Table 19: Early stopping iteration distribution for text2SQL debugging on the **TableBench** dataset. While slightly more refinement is needed than in Table 6, most models still resolve within the first few iterations. Unlike PoT, text2SQL begins counting from $N = 0$ because, if no execution error occurs at the initial step, the loop stops without further debugging.

-	Model	N = 1	N = 2	N = 3	N = 4	N = 5	N = 6	N = 7
small LLM	llama3.2-3b	140	4	0	0	0	0	0
	mistral-7b	117	13	1	3	0	0	10
	phi4-mini-3.8b	141	3	0	0	0	0	0
	qwen2.5-3b	120	17	7	0	0	0	0
	qwen2.5-7b	131	9	4	0	0	0	0
large LLM	mistral-small-24b	144	0	0	0	0	0	0
	cogito-32b	135	9	0	0	0	0	0
	qwen2.5-32b	142	2	0	0	0	0	0
	GPT-4o	138	6	0	0	0	0	0
	Claude-3.7-Sonnet	144	0	0	0	0	0	0

Table 20: Early stopping iteration distribution for CoT self-refinement on the **Penguins in a Table** dataset. The majority of models terminate at iteration $N = 1$, showing that CoT reasoning rarely benefits from multiple rounds of refinement.

-	Model	N = 1	N = 2	N = 3	N = 4	N = 5	N = 6	N = 7
small LLM	llama3.2-3b	669	20	4	0	0	0	0
	mistral-7b	497	172	5	2	0	0	17
	phi4-mini-3.8b	582	109	2	0	0	0	0
	qwen2.5-3b	652	41	0	0	0	0	0
	qwen2.5-7b	621	70	2	0	0	0	0
large LLM	mistral-small-24b	558	130	5	0	0	0	0
	cogito-32b	685	8	0	0	0	0	0
	qwen2.5-32b	652	40	1	0	0	0	0
	GPT-4o	684	8	1	0	0	0	0
	Claude-3.7-Sonnet	690	3	0	0	0	0	0

Table 21: Early stopping iteration distribution for CoT self-refinement on the **TableBench** dataset. Most models also stop at iteration $N = 1$, indicating that even in more complex scenarios, CoT outputs are generally stable and do not require iterative correction.

-	Model	*w/o CC*	MATA	%↓	*w/o sch*	MATA	%↓
small LLM	llama3.2-3b	144	18	87.5%	714	695	2.66%
	mistral-7b	144	5	96.5%	823	786	4.50%
	phi4-mini-3.8b	144	7	95.1%	659	635	3.64%
	qwen2.5-3b	144	6	95.8%	695	641	7.77%
	qwen2.5-7b	144	1	99.3%	616	527	14.44%
large LLM	mistral-small-24b	144	10	93.1%	746	584	21.72%
	cogito-32b	144	1	99.3%	625	475	24.00%
	qwen2.5-32b	144	2	98.6%	624	483	22.60%
	GPT-4o	144	8	94.4%	651	495	23.96%
	Claude-3.7-Sonnet	144	2	98.6%	597	444	25.63%
	total	1440	60	**95.8%**	6750	5765	**14.59%**

Table 22: LLM agent call reduction analysis on the **Penguins in a Table** dataset (144 examples per model). Left (Confidence Checker Effect): In the w/o *CC* setting, the Judge Agent (*JA*) is mandatorily invoked for every query to select the final answer, resulting in 1,440 total calls. MATA utilizes *CC* to validate candidates and bypasses *JA* when confidence is high, drastically reducing *JA* calls by 95.8%. Right (Scheduler Effect): w/o *sch* executes both PoT and text2SQL agents for all queries. The Scheduler prioritizes one path and skips the other if it aligns with CoT, achieving a 14.59% reduction.

-	Model	*w/o CC*	MATA	%↓	*w/o sch*	MATA	%↓
small LLM	llama3.2-3b	693	293	57.7%	4600	4427	3.76%
	mistral-7b	693	362	47.8%	5345	5287	1.09%
	phi4-mini-3.8b	693	388	44.0%	4455	4342	2.54%
	qwen2.5-3b	693	362	47.8%	4452	4202	5.62%
	qwen2.5-7b	693	350	49.5%	3717	3627	2.42%
large LLM	mistral-small-24b	693	210	69.7%	4323	3863	10.64%
	cogito-32b	693	188	72.9%	3599	3147	12.56%
	qwen2.5-32b	693	209	69.8%	3897	3419	12.27%
	GPT-4o	693	187	73.0%	3436	2915	15.16%
	Claude-3.7-Sonnet	693	178	74.3%	3300	2762	16.30%
	total	6930	2727	**60.6%**	41124	37991	**7.62%**

Table 23: LLM agent call reduction on the **TableBench** dataset (693 examples per model). Left (Confidence Checker Effect): Without *CC* (w/o *CC*), the system forces the Judge Agent (*JA*) to evaluate all 6,930 instances. MATA leverages *CC* to assess answer confidence and invokes JA only for uncertain cases, thereby cutting expensive JA inference by 60.6%. Right (Scheduler Effect): Similarly, w/o *sch* runs all reasoning paths, whereas the Scheduler selectively skips the redundant path (either PoT or text2SQL) when consistency is found, reducing total agent calls by 7.62%.

Algorithm 4 The entire inference process of our proposed MATA

Require: Table (T), Question (Q), Refinement Count (N), Threshold (θ)
Ensure: Final Answer (A_f)

```
function Code&Debug(Agent, T, Q, N)
    Initialize 𝒞, A ← ∅
    if Agent == PoTA then
        Debug ← PDA
    else
        Debug ← SDA
    code^0, A^0 ← Agent(T, Q)
    Append (code^0, A^0) to 𝒞, A
    for i = 0 to N − 1 do
        code^{i+1}, A^{i+1} ← Debug(T, Q, code^i, A^i)
        Append (code^{i+1}, A^{i+1}) to 𝒞, A
        if Stop_condition == True then
            break
    return 𝒞, A

sol_cot, A_cot ← CoTA(T, Q)
if Use_Scheduler == True then
    prob_pot, prob_sql ← Sch(T, Q)
    if prob_pot >= prob_sql then
        𝒞_pot, A_pot ← Code&Debug(PoTA, T, Q, N)
        if A_cot ≠ Last Answer in 𝒜_pot then
            𝒞_sql, A_sql ← Code&Debug(t2SA, T, Q, N)
        else
            𝒞_sql, A_sql ← ∅
    else
        𝒞_sql, A_sql ← Code&Debug(t2SA, T, Q, N)
        if A_cot ≠ Last Answer in A_sql then
            𝒞_pot, A_pot ← Code&Debug(PoTA, T, Q, N)
        else
            𝒞_pot, A_pot ← ∅
else
    𝒞_pot, A_pot ← Code&Debug(PoTA, T, Q, N)
    𝒞_sql, A_sql ← Code&Debug(t2SA, T, Q, N)

C, P, TS ← CC(T, Q, sol_cot, A_cot, 𝒞_pot, A_pot, 𝒞_sql, A_sql)
if max(C, P, TS) > θ then
    A_f ← arg max_{A∈{A_cot, A_pot, A_sql}} {C, P, TS}
else
    A_f ← JA(T, Q, sol_cot, A_cot, 𝒞_pot, A_pot, 𝒞_sql, A_sql)
if len(A_f) > 100 then
    A_f ← FM(A_f)
return A_f
```