Computers and Artificial Intelligence；Vol.2, No.3; 2025
E-ISSN: 3006 -3728 P-ISSN：3006-371X
DOI: https://doi.org/10.70267/ cai.25v2n3.117142
Published by: Zeus Press

Multi Table QA: Evaluating Modern LLM Strategies on Table Understanding

Letian Li*

University of Wisconsin - Madison Madison, WI

**Corresponding author: Letian Li*.*

Abstract

Tables constitute the majority of structured data in enterprise environments. While single-table question answering has received significant attention, research on multi-table reasoning remains limited. Compared to single-table QA, multi-table QA requires schema alignment, relational inference, and scalable context management. We introduce tool-augmented reasoning as a paradigm for multi-table QA, and systematically study two complementary strategies: (1) free-form tool interaction, where models iteratively call exploration and computation tools, and(2) structured agent workflows, which stage tool-use into exploration, preparation, and analysis phases. Using the TQA-Bench dataset, we evaluate GPT-4o-mini and GPT-5-mini across both small (8k) and large (128k) database scales under varying tool context constraints. We show that tool augmentation substantially improves robustness and accuracy over direct prompting, with gains up to +28 percentage points. Structured workflows yield further benefits for weaker models on large databases, but regress for stronger models, revealing a scale – capacity trade-off in the value of structure. These results establish tool-augmented reasoning as a powerful paradigm for multi-table QA. Structure aids weaker models under scale but constrains stronger models, underscoring that tool-use strategies must be adapted jointly to model ability and database complexity.

Keywords

large language models, multi-table question answering, tool-augmented reasoning, structured data understanding, agent-based workflows, tabular data reasoning

1. Introduction

Tables constitute the majority of structured data in enterprise systems, serving as the foundation for business management, analytics, and decision-making processes. As such, understanding tabular data with language models offers significant public benefits Lu et al. (2025). Single-table question answering has received significant attention in recent research, yet in reality data rarely resides in a single flat table. Instead, analysts must reason across multiple interrelated tables, align schemas, perform joins, and aggregate values across relational boundaries. This multi-table reasoning problem remains far less explored, despite being central to real-world analytics.

Compared to single-table QA, multi-table QA introduces distinct challenges. First, it requires schema-alignment, identifying which attributes across tables should be matched. Second, it demands relational-inference, where models must infer connections across heterogeneous data. Third, the problem imposes scalable context management: large databases can easily exceed model context windows, forcing trade-offs

between recall and efficiency. These challenges leave multi-table QA as an open research problem.
Recent advances in tool-augmented LLMs suggest a promising direction for addressing these challenges Yao et al. (2023))Yao et al. (2023). Rather than treating the model as a single end-to-end operation, tool-augmented reasoning allows models to iteratively interact with data via exploration, selective retrieval, and programmatic computation (Chen et al., 2023, Gao et al., 2023). This paradigm has transformed tasks such as arithmetic reasoning Gao et al. (2022), but has not been systematically studied in the domain of multi-table reasoning. Our work demonstrates the value of tool-augmented reasoning for multi-table QA through controlled comparisons.

We evaluate two complementary strategies for tool use. The first, free-form tool interaction, equips models with tools for schema inspection, row sampling, full-table access, and Python-based computation, allowing iterative exploration in a ReAct-style loop (Yao et al., 2023, Chen et al., 2023). The second, structured agent workflows, organize tool usage into staged phases: exploration, data preparation, and analysis. This design enforces systematic coverage of reasoning steps and may mitigate search inefficiency. By comparing these strategies to direct prompting baselines, we test not only whether tools help, but how their organization interacts with model capacity and database scale.
Our contributions are threefold:

• **Paradigm Introduction**: We introduce tool-augmented reasoning as a novel and effective paradigm for multi-table QA, demonstrating substantial gains (+15-28 percentage points) over direct prompting across all models and scales. This paradigm shift transforms multi-table reasoning from a text understanding problem to an interactive data exploration task.

• **Scale-capacity tradeoff study**: We show that structured agent workflows benefit weaker models under large-scale databases (+7. 11pp for GPT-4o-mini) by mitigating exploration inefficiency, but hinder stronger models (-5. 81pp for GPT-5-mini) due to overhead and loss of flexibility. This scale-capacity interaction challenges the assumption that more structure universally improves performance.

• **Deployment insights**: Contemporary models achieve near-ceiling performance (95. 65% with tools) on small-scale multi-table reasoning, but struggle dramatically under deployment constraints. Context limitations cause 4-14pp degradation in sophisticated strategies, while the presence of ignorable junk data reduces performance by 15-28pp. These findings establish deployment robustness—not fundamental reasoning—as the primary research frontier for real-world multi-table QA systems.

Taken together, our findings establish tool augmentation as a powerful paradigm for multi-table QA. More broadly, they show that the value of structure is not uniform but conditional on both model ability and database complexity, reframing the design space of table reasoning systems.

2. Related Work

Our work builds upon recent research in tool-augmented reasoning with large language models and new multi-table question answering benchmarks.

Multi-Table QA. While table QA has been widely studied in the single-table setting, multi-table reasoning has only recently received systematic treatment. The TQA-Bench dataset Qiu et al. (2024))Qiu et al. (2024) provides a controlled benchmark for evaluating multi-table reasoning under scalable context sizes with symbolic verification, and serves as the foundation for our study. By leveraging this benchmark, we move beyond ad-hoc task construction and situate our analysis within a reproducible evaluation framework.

Tool-augmented Reasoning. Prior work has shown that tools can extend LLM capabilities across tasks such as question answering, programming, and reasoning. ReAct Yao et al. (2023) introduced multi-turn reasoning with iterative tool calls; PAL Gao et al. (2022) demonstrated precise delegation of computations to external execution environments; and ChatCoT Chen et al. (2023) highlighted conversational tool use to structure reasoning chains. These paradigms inform our implementations of free-form tool interaction and structured workflows.

While existing work has established the value of tool augmentation for various reasoning tasks, our work is the first to systematically evaluate how tool organization strategies interact with model capacity and database

scale in the multi-table domain, revealing that structure is not universally beneficial but depends critically on model architecture, database size, and internal model assumptions.

3. Methodology

We study the problem of multi-table question answering (QA) with large language models (LLMs) and introduce tool-augmented reasoning as a new paradigm for this task. Our methodology com-pares three approaches: (1) a direct prompting baseline, (2) free-form tool-augmented reasoning, and (3) structured agent workflows. The latter two fall under the tool-augmented paradigm but differ in the degree of structure imposed on tool usage. To test robustness, each approach is evaluated under multiple database scales and context limitation conditions.

3.1 Problem formulation

Given a multi-table database D = {T1 , T2, . . . , Tn } where each Ti represents a table with schema Si , and a natural language question Q, the model must select an answer $\hat{A}$ from a set of multiple choice options C = {C1 , C2, . . . , Ck }. A predicted answer $\hat{A}$ is correct only if it matches the actual answer A $\in$ C. For a model to determine the correct answer, it needs to effectively reason across multiple tables, understand relationships between them, and perform the necessary computations to arrive at the correct answer.
We use the TQA-Bench dataset which provides a test set in the above format with k = 4 multiple choice options per question Q, and use their 8k and 128k db D sizes for evaluation Qiu et al. (2024). We further provide a never-correct answer choice C5 $\neq$ A = ”N/A” to discourage the model from randomly guessing, creating |C| = 5 multiple choice options per question Q.

3.2 Table serialization

For our baseline of direct prompting, we serialize all tables in the database as csv’s preceded by a dataset header (# <dataset>) per-table headers (## <table name>). While TQA-Bench provides a Markdown serialization option, our study was limited to CSV parsing.

For tool-augmented reasoning and structured workflows, we instead pre-load tables as in-memory pandas DataFrames. Models interact with them via explicit tool calls (see below), isolating tool-augmented strategies from raw serialization limits and ensuring the serialization strategy is optimized for the tool in question.

3.3 Answer parsing

All methodologies must output their selection $\hat{A}$ in a standardized format. The model must produce a line of the form Answer: A/B/C/D or N/A, followed on the next line by I AM DONE. We parse the last occurrence of this pattern case-insensitively to obtain the final choice. Invalid or missing choices are marked as empty predictions and incorrect.

3.4 Baselines and tool-augmented strategies

We evaluate three distinct approaches to multi-table reasoning, each representing different strategies for leveraging LLM capabilities and external tools.

3.4.1 Direct prompting(Baseline)

The baseline approach represents the traditional method of table QA, where the complete database is serialized as text and provided to the LLM in a single prompt. This is the strategy used in the TQA-Bench paper Qiu et al. (2024). This approach treats multi-table reasoning as a text understanding problem, requiring the model to process all available information simultaneously.

Implementation. We serialize the entire database as flat text (Section 3. 2) and pass it to the model in one turn, without any tool calls. Specifically, we construct a single user message consisting of: (1) a brief instruction to “analyze and answer step by step, ” (2) the full database dump, (3) the natural language question, (4) the multiple-choice options, and (5) detailed instructions to ”break down the question, evaluate each option, and explain”, as well as the answer format.

A complete prompt example (with partial tables) can be found in Appendix B.

This setup uses a single API completion per question and no system/tool messages.

3.4.2 Tool-augmented reasoning

The tool-augmented approach provides LLMs with access to computational tools that enable iter- ative database exploration and analysis. This paradigm allows models to dynamically explore the database structure and perform computations as needed.

We designed 4 tools for the model to call at any point in time. Some tool behaviors are modified by experimental configurations; see Section 4. 1. 3.

Available Tools (Function Calls):

• getTableNames(): Returns all available table names in the database

• peekTables(tableNames): Retrieves schema and sample data (first 5 rows) for specified tables

• readTables(tableNames) (conditionally available): Returns complete table con- tents as text for specified tables

• executePython(code): Executes Python code in a sandboxed environment with ac-cess to pandas, numpy, and helper functions to access the table DataFrames, returns stdout and stderr. Times out after 2 minutes.

Implementation: The model is given tool schemas, detailed tool descriptions, the question and multiple choice options, and instructions to use the tools to answer said question. We structure the conversation as a multi-turn exchange where the model can call tools between reasoning steps. On each turn (round), if the model's response has one or more tool calls, the corresponding implementations are executed and tool messages are appended to the model's conversation history. The conversation continues until the model either provides a final answer in the required format (Section 3. 3) without requesting additional tools, or hits the maximum limit of 10 rounds per question.

A complete prompt example can be found in Appendix C.

Tool call availability and behavior is modified by experiments. See section 4. 1. 3for more details.

3.4.3 Structured agent workflow

The structured agent approach implements a systematic three-stage reasoning process that explicitly separates different aspects of multi-table reasoning. This strategy provides a structured alternative to free-form tool usage, ensuring comprehensive coverage of the reasoning process.
Each of the three stages has access to the same tool calls and is given the same tool schema: peekTables(), executePython(), and conditionally based on the experimental configuration readTables().
The tool getTableNames() was removed because stage 1 is directly given the list of table names.

Stage 1 - Exploration: This stage determines which tables and columns are relevant for answering the question Q. The agent is directly given the list of table names, detailed instructions on its role, the question without answer choices, and asked to determine which tables contain information relevant to answering the question. The stage concludes with explicit identification of relevant tables and columns, as well as a free form note to the next stage about its findings.

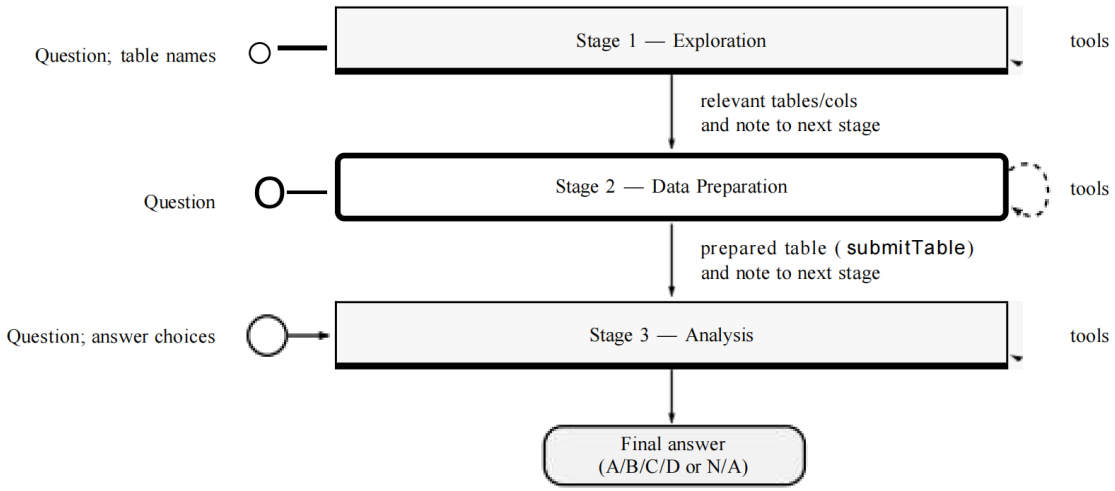
Stage 2 - Data Preparation: This stage consolidates the large set of tables and columns found by Stage 1 into a single table most helpful for answering the question Q. The agent is given a preview of each of the relevant tables found in the previous stage, detailed instructions on its role, the question without answer choices, instructions on how to submit a table to Stage 3, and asked to submit said table to the next stage. While it is given a preview of only the relevant tables, it theoretically still has access to all the other tables. The stage concludes after the agent uses the submitTable() function inside the executePython() tool, and after it explicitly identifies the name of the table it submits and a free form note to the next stage about its constructed table.

Stage 3 - Analysis: This stage uses the consolidated table to answer the question. The agent is given a preview (first five rows) of the consolidated table, the question along with answer choices, and asked to analyze the table as necessary and output the final answer. Just like stage 2, the model theoretically still has access to

all the other tables and is not limited to the consolidated table. In practice, it does not matter as models typically do not explore when already given the minimum data to solve. The stage concludes in the same manner as Tool-Augmented Reasoning.

Implementation: The Structured Agent Workflow can be thought of running Tool-Augmented reasoning in 3 steps. We enforce a maximum of 5 rounds per stage, down from 10 from Tool- Augmented Reasoning. Figure 1illustrates the structured agent workflow.
A complete prompt example can be found in Appendix D.

Figure 1: Structured Agent Workflow



4. Experiments

4.1 Experimental setup

We conduct a comprehensive evaluation of Large Language Models (LLMs) on multi-table question answering using the TQA-Bench test set. Our study systematically compares different interaction paradigms and examines the impact of context limitations across two commercial models: GPT-4o- mini and GPT-5-mini. We evaluate these models across three distinct approaches: direct prompting, tool-augmented reasoning, and structured agent workflows, with varying context limitations to understand scalability and robustness.

4.1.1 Models and interaction paradigms

We evaluate two state-of-the-art language models through the official OpenAI API: GPT-4o-mini, OpenAI's efficient multimodal model, and GPT-5-mini, the recently released latest generation model with improved tool usage and reasoning abilities. We use the "mini" versions to minimize computational costs while maintaining comprehensive evaluation coverage.

Direct Prompting (Baseline): This approach provides models with complete database serialization as text, requiring direct question answering without tool access. This represents the traditional in- context approach to table question answering and serves as our baseline for comparison.

Tool-Augmented Reasoning: This paradigm grants models access to four specialized functions: getTableNames() for retrieving table names, peekTables(tableNames) for examining table structure and sampling the first five rows without overwhelming the con- text window, readTables(tableNames) for reading entire tables (if enabled), and executePython(code) for executing Python code in a sandboxed environment with access to pandas DataFrames containing all table data.

Structured Agent Workflow: This approach implements a three-stage workflow where models systematically explore relevant tables, prepare unified data representations, and perform final analysis. Each stage has specific completion markers and context constraints, with maximum 5 rounds per stage and 15 total

rounds across all stages.

4.1.2 Dataset and scale configurations

Our evaluation utilizes ten real-world databases spanning diverse domains: airline flight and route information, cookbook recipe and ingredient data, food facility inspection records, global biodiversity species and habitat information, movie and actor databases, music tracker song and artist information, restaurant dining establishment data, university academic institution information, and water quality environmental monitoring data.

Database Scale: We evaluate performance across two database scales to understand scalability:

• **8k scale:** Small-scale databases that generate approximately 8, 000 tokens when serialized using markdown format. With CSV serialization (used in our experiments), this corresponds to approximately 4, 000 tokens, representing manageable context sizes for baseline performance evaluation.

• **128k scale:** Large-scale databases that generate approximately 128, 000 tokens when serialized using markdown format. With CSV serialization, this corresponds to approximately 62, 000 tokens, testing context handling capabilities and scalability under realistic data volumes.

For consistency, we will refer to the 3k token database as 8k as that is how it is labeled in the TQA-Bench dataset.

4.1.3 Context limitation configurations

We define three context limitation configurations to attempt to emulate larger than context window databases and examine the impact of resource constraints. These are not perfect emulations, as ideally the database is actually larger with more data.

Unlimited Context: readTables is available (permitting full table dumps on demand) and executePython captures full stdout without truncation. This configuration represents the ideal scenario with no context limitations and no concerns about the database being too large.

Limited Context: readTables is disabled; peekTables remains available to sample the first five rows of data. Stdout from any executePython call is truncated to 1, 000 characters with a warning. This configuration simulates the model being unable to read the entire database at once.

Limited Context with Junk Data: The constraints from Limited Context are applied, but additionally we append a single garbage row to tables to increase the size of the database past the maximum context window. Two additional columns are added, one boolean column indicating if the row is a garbage row and one column to hold the arbitrary garbage data. This configuration is only tested on the 128k scale with the water quality table to evaluate robustness.

Notably, the garbage data is explicitly labeled with a boolean column indicating which rows contain junk data. This design choice tests a critical aspect of model behavior: whether models proactively explore data quality indicators and utilize this information during their reasoning process. In practice, models often fail to investigate the meaning of the boolean column or utilize this metadata when peeking at tables, leading to performance degradation even when the irrelevant data is explicitly marked. This reveals limitations in models' ability to autonomously handle data quality metadata and explore table schemas for quality indicators.

4.1.4 Experimental matrix

For each model (GPT-4o-mini and GPT-5-mini), we test the following configurations:

• Direct Prompting (unlimited) - 8k and 128k scales

• Direct Prompting (limited) - 8k and 128k scales (equivalent to unlimited for direct prompting)

• Tool Augmented (unlimited) - 8k and 128k scales

• Tool Augmented (limited) - 8k and 128k scales

• Tool Augmented (limited with junk) - 128k scale, water_quality table only

• Agent (unlimited) - 8k and 128k scales

- Agent (limited) - 8k and 128k scales
- Agent (limited with junk) - 128k scale, water_quality table only

4.1.5 Evaluation framework

We implement question validation using the testValid() function to filter out malformed ques-tions in the TQA-Bench dataset. Notably questions where some or all choices were ”nan” or ”” or ”none” were considered invalid. This preprocessing ensures only well-formed questions are included in evaluation.
For each database and scale combination, we evaluate 5 different database instances, 1 sample per database, 14 questions per sample, covering all valid question types within each database.

4.1.6 Implementation details

All experiments use OpenAI’s API with model-specific parameter configurations. GPT-4o-mini uses temperature 0. 6 and top p 0. 95. Because GPT-5-mini dropped support for such parameters, no extra parameters were attached to each request body.

Tool-augmented strategies implement conversation round limits to prevent infinite loops: 10 rounds maximum for free-form tool interaction, and 5 rounds per stage (15 total) for structured agent work- flows. Code execution in executePython() calls is limited to 120 seconds (2 minutes) with timeout handling to prevent hanging processes.

The exact prompts used can be found in Appendices B, C, and D.

These implementation constraints ensure fair competition. As far as we know, no round limits nor timeouts were hit by any model during evaluation.

4.1.7 Evaluation metrics

Our primary evaluation metrics include accuracy (percentage of questions answered correctly), question type performance (accuracy broken down by question complexity and type), and error analysis (classification of failure modes and error types). Secondary metrics include token usage (input and output token consumption patterns), and tool usage patterns (frequency and effectiveness of tool utilization).
All experimental results are stored in SQLite databases with comprehensive metadata including model and configuration information, question validity flags, token usage statistics, and full model responses for error analysis. This experimental design enables systematic comparison of different approaches to multi-table reasoning while controlling for dataset characteristics, model capabilities, and practical implementation constraints.

5. Results

We evaluate two models (GPT-4o-mini, GPT-5-mini) across one baseline (direct prompting) and two tool-augmented strategies (free-form tool interaction and structured agent workflows). Experiments span 10 database schemas, totaling 689 questions at the 128k scale and 668 at the 8k scale after filtering for well-formed multiple-choice format.

5.1 Overall performance

Table 1 reports aggregate results at the 128k context scale with full tool access. Tool-augmented approaches yield substantial gains for both models, improving over direct prompting by +27. 7pp for GPT-4o-mini and +14. 9pp for GPT-5-mini. Structured agent workflows further improve GPT- 4o-mini (+7. 1pp over tools) but regress for GPT-5-mini (-5. 8pp), indicating a capacity-dependent benefit of added structure.

Table 1: Overall performance across strategies at 128k scale (unlimited context)

Model	Baseline	Tools	Agent
GPT-4o-mini	42. 09%	69. 81%	76. 92%
GPT-5-mini	80. 70%	95. 65%	89. 84%

The results demonstrate three primary findings. First, tool-augmented approaches yield substantial

performance improvements of 27. 72 percentage points for GPT-4o-mini and 14. 95 percentage points for GPT-5-mini compared to baseline prompting. Second, agent strategies exhibit model-dependent effectiveness, providing additional benefits of 7. 11 percentage points for GPT-4o-mini while showing reduced performance of 5. 81 percentage points for GPT-5-mini. Third, GPT-5-mini consistently outperforms GPT-4o-mini by as much as 40 percentage points across all strategies.

To visualize these patterns, Figure 2compares performance across each strategy for each model and context scale (8k and 128k) combination, all at unlimited context.

Figure 2: Model and Context Size

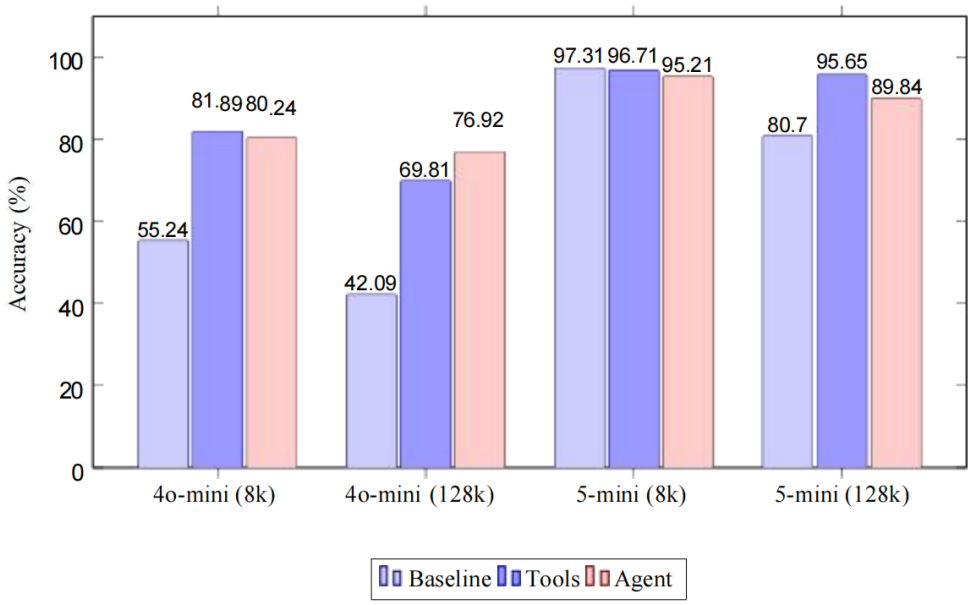


Figure 2: Performance by model and context size, grouped by strategy. This view highlights how each strategy performs across different model architectures and scales, showing the consistent ad- vantage of tool-augmented approaches across all conditions.

5.2 Scale sensitivity

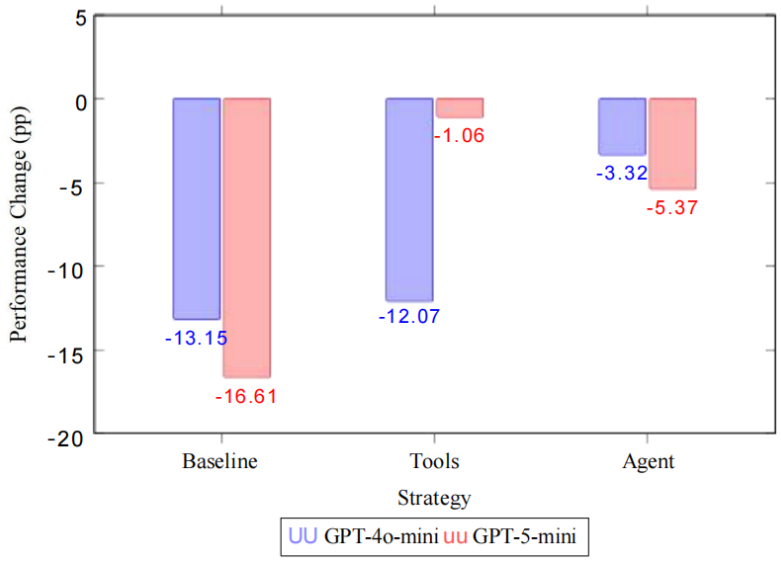
Table 2presents a comparison of performance between 8k and 128k context scales across all model- strategy combinations using unlimited context configurations. Values represent accuracy percent- ages calculated using the same methodology as the main results table. The difference column shows the change in accuracy when moving from 8k to 128k context scale.

Table 2: Scale Comparison: 8k vs 128k Performance (Unlimited Context Configurations)

Configuration	8k Accuracy	128k Accuracy	Difference
GPT-4o-mini Baseline	55. 24%	42. 09%	-13. 15pp
GPT-4o-mini Tools	81. 89%	69. 81%	-12. 07pp
GPT-4o-mini Agent	80. 24%	76. 92%	-3. 32pp
GPT-5-mini Baseline	97. 31%	80. 70%	-16. 61pp
GPT-5-mini Tools	96. 71%	95. 65%	-1. 06pp
GPT-5-mini Agent	95. 21%	89. 84%	-5. 37pp

The analysis reveals distinct patterns in scale sensitivity across models and strategies. GPT-4o- mini exhibits significant performance degradation when transitioning from 8k to 128k context scale, with baseline and tools strategies showing reductions of 13. 15 and 12. 07 percentage points, respectively. In contrast, GPT-5-mini demonstrates greater stability across context scales under the tool- augmented paradigm, with minimal performance changes ranging from -1. 06 to -5. 37 percentage points. The agent strategy appears least affected by scale changes, showing the smallest performance differences across both models.

Figure 3: Scale sensitivity: Performance change from 8k to 128k context scales



GPT-4o-mini shows significant degradation across all strategies, while GPT-5-mini demonstrates greater stability, particularly with tool-augmented approaches.

5.3 Context robustness

We analyze robustness under three context regimes: unlimited, limited (tool truncation), and limited+junk (irrelevant noise). Table 3 shows representative results at 128k scale. Baselines are nearly unaffected (i2pp loss), while tool and agent paradigms degrade substantially when context or relevance is restricted, especially for GPT-4o-mini. GPT-5-mini shows milder degradation, maintaining 90% accuracy even under noisy conditions.

Table 3: Context configuration comparison at 128k scale (full dataset)

Model	Paradigm	Unlimited	Limited	Limited+Junk
GPT-4o-mini	Baseline	42. 09%	40. 64% (-1. 45pp)	-
	Tools	69. 81%	55. 71%(-14. 10pp)	38. 57%(-17. 14pp)
	Agent	76. 92%	65. 71%(-11. 21pp)	37. 14%(-28. 57pp)
GPT-5-mini	Baseline	80. 70%	79. 83% (-0. 87pp)	-
	Tools	95. 65%	90. 00% (-5. 65pp)	84. 29% (-5. 71pp)
	Agent	89. 84%	84. 29% (-5. 55pp)	75. 71% (-8. 58pp)

Table 4: Context Configuration Comparison (128k Water Quality DB Schema Only)

Model	Paradigm	Unlimited	Limited	Limited+Junk
GPT-4o-mini	Baseline	47. 14%	45. 71(-1. 43pp)	-
	Tools	62. 86%	55. 71(-7. 15pp)	38. 57(-17. 14pp)
	Agent	65. 71%	60. 00%(-5. 71pp)	37. 14%(22. 86pp)
GPT-5-mini	Baseline	81. 43%	81. 43% (0. 00pp)	-
	Tools	92. 86%	90. 00% (-2. 86pp)	84. 29%(-5. 71pp)
	Agent	84. 29%	84. 29% (0. 00pp)	75. 71%(-8. 58pp)

The context configuration analysis examines the robustness of different strategies to context limitations by comparing unlimited, limited, and limited+junk configurations across the full dataset. Note that unlimited baseline and limited baseline are technically identical configurations since baseline prompting does not use tools that would be affected by context restrictions, though we observe negligible performance differences likely due to the semi-random nature of LLMs.

Analysis of context configuration sensitivity reveals consistent patterns across both full dataset and water quality-only evaluations. Baseline strategies show negligible performance impact from unlimited to limited

context (4o-mini:-1. 45pp full dataset, -1. 43pp water quality; 5-mini:-0. 87pp full dataset, 0. 00pp water quality), confirming that simple prompting is robust to tool-side context restrictions. Tool strategies show moderate degradation (4o-mini: -14. 10pp full dataset, -7. 15pp water quality; 5-mini: -5. 65pp full dataset, -2. 86pp water quality), with additional performance loss when junk data is added (-17. 14pp and -5. 71pp respectively). Agent strategies show significant degradation (4o-mini: -11. 21pp full dataset, -5. 71pp water quality; 5-mini: -5. 55pp full dataset, 0. 00pp water quality), with dramatic junk impact (-28. 57pp and -8. 58pp respectively).

Across both analyses, adding junk data disproportionately affects agent strategies, with GPT-4o-mini agent showing the largest performance drops in both scenarios. GPT-5-mini demonstrates greater robustness to context limitations than GPT-4o-mini across all strategies, particularly when junk data is added.

5.4 Tool usage behavior

To understand how models operationalize external reasoning, we analyze tool invocation patterns across paradigms, question types, and model sizes.

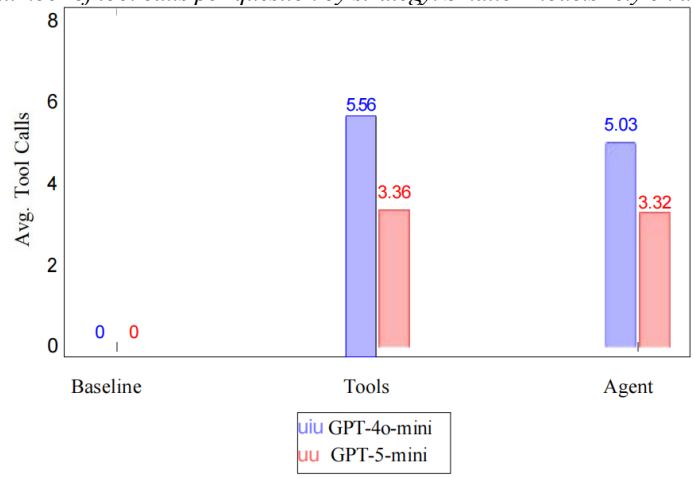
5.4.1 Average calls per question

Table 5 and Figure 4 summarize average tool usage per question. GPT-4o-mini issues roughly 5-6 calls per query, while GPT-5-mini averages 3-3. 5. Aggregation-oriented tasks such as sum, average, and count consistently trigger more invocations, reflecting their multi-stage reasoning demands. Smaller models exhibit denser tool interaction, fragmenting reasoning into more granular steps, whereas larger models consolidate computation into fewer, higher-impact calls.

Table 5: Average number of tool calls per question by type

Type	GPT-4o-mini			GPT-5-mini		
	Agent	Tools	Baseline	Agent	Tools	Baseline
Average	5. 11	5. 89	0. 00	3. 22	3. 44	0. 00
Correlation	5. 10	4. 60	0. 00	3. 40	3. 10	0. 00
Count	4. 20	6. 00	0. 00	3. 00	4. 00	0. 00
Difference	5. 90	5. 80	0. 00	3. 60	3. 20	0. 00
Item-Select	4. 80	5. 10	0. 00	3. 10	3. 50	0. 00
Row-Match	4. 60	4. 00	0. 00	3. 60	3. 00	0. 00
Sum	5. 50	7. 50	0. 00	3. 30	3. 30	0. 00
All	5. 03	5. 56	0. 00	3. 32	3. 36	0. 00

Figure 4: Average number of tool calls per question by strategy. Smaller models rely on denser tool interaction



The overall pattern reinforces the earlier performance trends: smaller models depend on more frequent external operations to reach comparable accuracy, whereas larger models execute fewer but more targeted tool calls—reflecting stronger internal planning and schema inference.

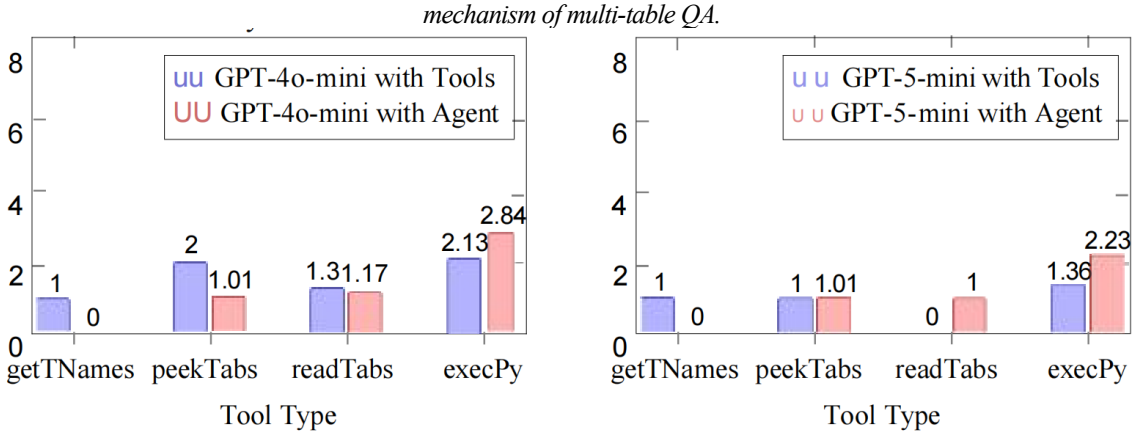
5.4.2 Tool type utilization

Figure 5 shows mean tool call frequency by tool type and model. Across all settings, executePython() dominates, accounting for roughly half of all calls. This confirms that computational reasoning—rather than simple retrieval or inspection—is the core driver of multi-table question answering.

The auxiliary tools follow predictable patterns. GetTableNames() appears exactly once per question in free-form tool interaction runs, where models must first discover available tables. It is intentionally disabled for structured agent workflows because Stage 1 already receives the list of table names. PeekTables() is consistently used for schema exploration and data preview, typically once or twice per question. This does not imply that only one or two tables are relevant; rather, each call may return multiple table samples in a single response.

Differences between models emerge most clearly in how readTables() is used. GPT-4o-mini frequently invokes this function immediately after peeking, loading entire tables into memory toreason explicitly over their content. GPT-5-mini, in contrast, rarely uses it—even when available—preferring selective inspection through peeks and computation via executePython(). This suggests a shift in strategy: smaller models rely on full-table access to compensate for weaker abstraction, while larger models retrieve only what they need to compute results efficiently.

Figure 5: Tool usage by type and model. executePython() dominates, indicating that computational reasoning is the core mechanism of multi-table QA.



(a) GPT-4o-mini (b) GPT-5-mini

5.4.3 Token consumption analysis

Token consumption analysis reveals sharp contrasts in how models manage computational resources. Direct prompting produces high token consumption despite being a single-turn process, since it requires full database serialization—averaging roughly 62, 000 input tokens per question at the 128k scale. Tool-based methods distribute reasoning across multiple turns but avoid embedding the entire database, leading to lower or comparable total input usage depending on model and strategy.

Table 6: Average Input Token Consumption by Question Type (128k Scale)

Type	GPT-4o-mini			GPT-5-mini		
	Agent	Tools	Baseline	Agent	Tools	Baseline
Average	75, 117	46, 673	62, 470	13, 079	7, 614	62, 469
Correlation	76, 216	44, 605	62, 552	13, 242	5, 941	62, 552
Count	73, 802	140, 632	62, 538	13, 841	9, 226	62, 537
Difference	78, 504	118, 729	62, 554	18, 311	5, 814	62, 553
Item-Select	75, 265	112, 643	62, 552	13, 276	7, 507	62, 551
Row-Match	74, 485	40, 699	62, 542	14, 858	5, 384	62, 541
Sum	75, 424	67, 349	62, 552	19, 920	7, 366	62, 551
All	75, 551	82, 125	62, 538	15, 249	6, 970	62, 537

Table 7: Average Output Token Consumption by Question Type (128k Scale)

Type	GPT-4o-mini			GPT-5-mini		
	Agent	Tools	Baseline	Agent	Tools	Baseline
Average	836	734	662	3, 015	1, 243	6, 687
Correlation	850	530	534	3, 060	1, 063	2, 016
Count	884	730	612	2, 921	1, 701	9, 496
Difference	1, 430	620	411	3, 007	825	589
Item-Select	942	2, 349	377	2, 938	1, 533	938
Row-Match	933	398	408	2, 839	684	599
Sum	854	1, 062	699	2, 973	1, 255	8, 915
All	963	920	527	2, 964	1, 186	4, 141

For GPT-4o-mini, both the tool-augmented and structured-agent strategies consume slightly more input tokens overall (≈75-82k vs. 62k). The smaller model compensates for limited context reasoning by calling tools repeatedly and expanding its intermediate reasoning chains. Output tokens also increase, with agent completions averaging 963 tokens—around 80% longer than baseline answers. These patterns indicate that GPT-4o-mini achieves higher accuracy through more verbose and iterative reasoning, trading efficiency for reliability.

For GPT-5-mini, the pattern reverses. Tool-augmented reasoning and structured workflows sharply reduce input consumption (≈7-15k vs. 62k). The stronger model retrieves only relevant data through selective tool calls rather than serializing entire databases. Although its responses are somewhat longer (1-3k output tokens), overall token usage remains far lower than the baseline.

In summary, weaker models increase token usage when using tools because they externalize reasoning across multiple steps. Stronger models use tools for targeted data access, substantially reducing redundant context. Tool-augmented reasoning thus transitions from compensatory to compressive as model capacity increases—reducing computational cost while maintaining or improving accuracy.

5.5 Key insights and implications

The comprehensive evaluation reveals several critical patterns that inform the design and selection of approaches for multi-table question answering systems.

Model capability emerges as the primary performance driver, with GPT-5-mini consistently outperforming GPT-4o-mini by 35-40 percentage points across all paradigms and configurations. Tool-augmented prompting provides substantial benefits for both models (15-28pp improvements over baseline), though agent paradigms show model-dependent effectiveness—beneficial for smaller models but counterproductive for larger ones.

Performance varies significantly by question type, with simple lookup operations achieving near- perfect accuracy (90%+) across all paradigms, while complex analytical queries reveal the largest capability gaps. Context scale sensitivity also varies substantially by model, with GPT-4o-mini showing significant degradation at 8k scale while GPT-5-mini maintains stable performance across different context sizes.

These findings collectively suggest that tool-augmented prompting represents the most robust default choice for multi-table question answering, while agent paradigms require careful evaluation based on specific model capabilities and may not provide additional benefits in all cases.

5.6 Analysis

Our comprehensive evaluation of multi-table question answering reveals critical insights about paradigm effectiveness and model capabilities that have significant implications for practical deployment.

5.6.1 Paradigm effectiveness and model dependency

The results demonstrate a clear performance hierarchy where tool-augmented prompting provides substantial improvements over baseline approaches (15-28 percentage points across both models), establishing structured tool access as the most reliable strategy for enhancing multi-table reasoning capabilities. However, agent paradigms exhibit striking model-dependent effectiveness: GPT-4o- mini benefits from agent approaches (+7. 11pp over tools), while GPT-5-mini experiences performance degradation (-5. 81pp from tools to agent).

This model-dependent pattern suggests that agent paradigms introduce reasoning complexity that benefits smaller models requiring structured guidance but proves counterproductive for larger models already possessing sophisticated reasoning capabilities. The findings indicate that optimal paradigm selection must consider model size and inherent capabilities, with tool-augmented prompting emerging as the most robust default choice across different model architectures.

5.6.2 Question type sensitivity and reasoning complexity

Performance analysis by question type reveals that paradigm effectiveness varies substantially with reasoning complexity. Simple operations (row matching, item selection) achieve near-perfect accuracy (90

However, complex analytical queries expose significant capability gaps, with the largest performance differences occurring in aggregation operations (sum, count, average) and correlation analysis. These results suggest that while current approaches handle basic table navigation effectively, sophisticated analytical reasoning remains a fundamental challenge that requires structured interaction paradigms to address effectively.

5.6.3 Context scale sensitivity and model robustness

Context scale analysis reveals critical differences in model robustness. GPT-4o-mini exhibits significant performance degradation when transitioning from 8k to 128k context scales (12-13pp drops across paradigms), while GPT-5-mini maintains relatively stable performance (-1. 06 to -5. 37pp changes). This pattern indicates that model selection may be more critical than paradigm optimization for handling longer context requirements, with larger models demonstrating inherent robustness to context scale changes.

The performance losses at shorter context scales suggest that tool-augmented approaches become less effective when context windows are constrained, as the overhead of tool usage cannot be adequately accommodated. This finding has important implications for deployment scenarios with limited context windows, where baseline prompting may prove more effective than tool-augmented approaches.

5.6.4 Context configuration robustness

Analysis of context limiting strategies reveals that baseline paradigms show minimal sensitivity to context restrictions (typically ¡2pp degradation), confirming that simple prompting approaches are robust to context limitations since they do not rely on tools affected by context constraints.

In contrast, tool and agent paradigms show significant degradation under context limitations (4- 14pp drops), with agent paradigms being most affected by both context restrictions and irrelevant junk data. These findings suggest that while sophisticated paradigms offer superior performance in unconstrained environments, they become increasingly vulnerable when deployed in resource constrained settings with context limitations or noisy data environments.

5.6.5 Model capability as primary performance driver

The evaluation clearly demonstrates that raw model capability remains the primary driver of multi- table question answering performance, with GPT-5-mini consistently outperforming GPT-4o-mini by 35-40 percentage points across all strategies and configurations. This substantial capability gap persists regardless of interaction strategy, suggesting that model architecture and training improvements provide the most significant performance gains.

The consistent model ordering across all experimental conditions indicates that strategy selection plays a secondary role to fundamental model capabilities. This finding suggests that investment in model development should be prioritized over interaction strategy optimization for achieving substantial performance improvements in multi-table question answering tasks.

5.6.6 Practical deployment implications

These findings provide clear guidance for practitioners deploying multi-table question answering systems. Tool-augmented prompting represents the most robust default choice, offering substantial performance improvements while maintaining reliability across different deployment constraints. Agent strategies require careful evaluation based on specific model capabilities and may not provide additional benefits in all cases.

For applications with constrained resources or noisy data environments, baseline prompting approaches

may prove more reliable than sophisticated strategies that become vulnerable under context limitations. The substantial performance gap between model architectures suggests that upgrading to more capable models provides greater returns than optimizing interaction strategies for existing models.

These empirical findings demonstrate that tool-augmented reasoning is essential for robust multi- table QA, that structure benefits depend critically on model capacity, and that deployment robustness—not core reasoning—represents the primary challenge for real-world systems.

6. Conclusion and future work

We have presented a comprehensive evaluation of interaction paradigms for multi-table question answering across 10 diverse database schemas, covering 689 questions on the 128k context scale and 668 questions on the 8k context scale (filtered from 700 total questions to ensure well-formed multiple-choice format). Our systematic analysis of two models (GPT-4o-mini, GPT-5-mini) across three paradigms (baseline, tools, agent) on both context scales reveals critical insights about paradigm effectiveness and practical deployment considerations.

6.1 Key findings

Our work makes three key conceptual contributions that advance the understanding of multi-table reasoning systems:

Tool augmentation is necessary for robustness. Our results demonstrate that tool-augmented reasoning is not merely beneficial but essential for robust multi-table QA performance. Without tool access, even state-of-the-art models like GPT-5-mini achieve only 80. 70% accuracy on complex multi-table questions. Tool augmentation provides 15-28 percentage point improvements across all models and scales, establishing it as a fundamental requirement rather than an optional enhancement for multi-table reasoning tasks.

Structure interacts with model scale and capacity. We reveal a critical interaction between reasoning structure and model capabilities: structured agent workflows benefit weaker models (+7. 11pp for GPT-4o-mini) by mitigating exploration inefficiency, but hinder stronger models (-5. 81pp for GPT-5-mini) due to overhead and loss of flexibility. This scale-capacity trade-off demonstrates that optimal strategy selection must consider both model architecture and database complexity, challenging the assumption that more structure universally improves performance.

Deployment robustness is now the frontier, not core reasoning. Contemporary models achieve near-ceiling performance (95. 65% with tools) on core multi-table reasoning, but struggle dramatically under deployment constraints. Context limitations cause 4-14pp degradation in sophisticated strategies, while irrelevant junk data—even when explicitly labeled—reduces performance by 15- 28pp. These findings shift the research frontier from fundamental reasoning capability to deployment robustness, highlighting that real-world multi-table QA success depends more on handling practical constraints than on core algorithmic improvements.

6.1.1 Future research directions

Looking ahead, multi-table QA benchmarks may need to evolve beyond current evaluation paradigms to address emerging challenges. Schema discovery tasks—where models must autonomously identify table relationships and attribute mappings—could test more realistic scenarios where database structure is not predefined. Hybrid unstructured-structured QA tasks would bridge the gap between natural language understanding and structured reasoning, requiring models to handle mixed-format data sources. Finally, noise robustness tasks specifically designed to evaluate model resilience to irrelevant data, missing values, and schema variations would better reflect real-world deployment challenges where data quality cannot be guaranteed.

6.1.2 Failure model analysis

While overall accuracy provides a high-level view of strategy effectiveness, analyzing specific failure modes reveals systematic weaknesses that inform strategy design and deployment decisions.

One prevalent failure mode across all strategies involves metadata column neglect. In the limited+junk configuration, models frequently fail to investigate boolean columns explicitly marking irrelevant ”junk” data

rows, leading to performance degradation of 15-28 percentage points even when the distinguishing metadata is readily available through simple peekTables() calls. This systematic oversight suggests that current strategies lack proactive schema exploration mechanisms, particularly for data quality indicators.

6.2 Practical implications

Our findings provide clear guidance for practitioners deploying multi-table question answering systems. Tool-augmented prompting represents the most robust default choice, offering substantial performance improvements while maintaining reliability across different deployment constraints. This approach balances effectiveness with practical deployment considerations, making it suitable for a wide range of applications.

For applications with constrained resources or noisy data environments, baseline prompting approaches may prove more reliable than sophisticated paradigms that become vulnerable under context limitations. However, the substantial performance gap between model architectures (35-40pp) suggests that upgrading to more capable models like GPT-5-mini provides greater returns than optimizing interaction paradigms for existing models.

Agent paradigms require careful evaluation based on specific model capabilities and may not pro- vide additional benefits in all cases. While beneficial for smaller models like GPT-4o-mini, these approaches can be counterproductive for larger models like GPT-5-mini that already possess sophisticated reasoning capabilities.

6.3 Limitations

Our evaluation has several limitations that should be considered when interpreting the results. First, while we evaluate performance across 10 diverse database schemas, these represent a subset of possible database structures and relationships. Performance patterns may vary with different database designs, table relationships, or data distributions not covered in our evaluation. The complexity within each TQA-Bench database schema may not match real-world scenarios.

Second, our analysis focuses on multiple-choice question answering, which may not fully capture the complexity of real-world multi-table reasoning scenarios that often involve open-ended questions requiring natural language generation. The performance characteristics of different strategies may differ substantially in open-ended settings.

Third, our evaluation is limited to specific model architectures (GPT-4o-mini and GPT-5-mini) and may not generalize to other model families or architectures. The model-dependent effectiveness patterns we observe may vary across different LLM architectures or training approaches. The recency of GPT-5-mini means it may have been exposed to the test set, which may also explain why it performs so well in direct prompt on 8k scale questions.

Finally, while we examine context scale effects and context limiting strategies, we do not evaluate all possible deployment constraints, such as different truncation strategies, rate limiting, or computational resource limitations that may affect practical deployment scenarios.

6.4 Future work

Several promising directions for future research emerge from our findings. First, developing adaptive paradigm selection mechanisms that can dynamically choose the optimal interaction approach based on question characteristics, database properties, and model capabilities could significantly improve practical deployment effectiveness. This could involve learning-based approaches that predict paradigm effectiveness or dynamic switching strategies based on intermediate results.

Second, exploring hybrid interaction paradigms that combine elements of different approaches could yield additional performance improvements while maintaining robustness. For instance, combining tool-augmented prompting with adaptive reasoning structures could provide benefits of both approaches while mitigating their respective limitations.

Third, extending evaluation to include open-ended question types and natural language generation tasks would provide a more comprehensive understanding of paradigm effectiveness across different multi-table

reasoning scenarios. This would help determine whether the patterns we observe in multiple-choice settings generalize to more complex reasoning tasks.

Fourth, investigating the impact of different database characteristics (schema complexity, table relationships, data distributions, larger scales) on paradigm effectiveness could provide additional insights for practical deployment. Understanding how paradigm performance varies across different types of multi-table scenarios would enable more targeted paradigm selection.

Finally, developing methods to automatically optimize interaction paradigms for specific deployment constraints (context limitations, computational resources, latency requirements) could enable more efficient multi-table reasoning systems. This could involve automated prompt engineering, dynamic tool selection, or resource-aware reasoning strategies.

6.5 Conclusion

Our work makes three fundamental contributions to multi-table question answering:

1. Introduced tool-augmented reasoning as paradigm for multi-table QA. We establish tool-augmented reasoning as a new paradigm for multi-table QA, demonstrating substantial gains (+15-28 percentage points) over direct prompting across all models and scales. This paradigm shift transforms multi-table reasoning from a text understanding problem to an interactive data exploration task.

2. Demonstrated scale–capacity tradeoff in structured workflows. We reveal that struc- tured agent workflows benefit weaker models under large-scale databases (+7. 11pp for GPT-4o-mini) by mitigating exploration inefficiency, but hinder stronger models (-5. 81pp for GPT-5-mini) due to overhead and loss of flexibility. This scale-capacity interaction challenges the assumption that more structure universally improves performance.

3. **Identified deployment robustness as the research bottleneck.** Contemporary models achieve near-ceiling performance (95. 65% with tools) on small-scale multi-table reasoning, but struggle dramatically under deployment constraints. Context limitations cause 4-14pp degradation in sophisticated strategies, while irrelevant junk data reduces performance by 15-28pp. These findings establish deployment robustness—not fundamental reasoning—as the primary research frontier for real-world multi-table QA systems.

Taken together, our contributions establish tool-augmented reasoning as essential for robust multi-table QA, reveal critical interactions between structure and model capacity, and shift research focus toward deployment robustness challenges.

References

Chen, Z., Zhou, K., Zhang, B., Gong, Z., Zhao, W. X. and Wen, J.-R., (2023). Published. Chatcot: Tool-augmented chain-of-thought reasoning on chat-based large language models. EMNLP 2023 Conference, 2023 Singapore. pp. 14777-14790.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J. and Neubig, G., (2023). Published. Pal: Program-aided language models. International Conference on Machine Learning, 2023 Honolulu, Hawaii, USA. PMLR, pp. 10764-10799.

Lu, W., Zhang, J., Fan, J., Fu, Z., Chen, Y. and Du, X., (2025). Large language model for table processing: A survey. *Frontiers of Computer Science,* vol. 19**,** no. 2, p. 192350.

Qiu, Z., Peng, Y., He, G., Yuan, B. and Wang, C., (2024). Tqa-bench: Evaluating llms for multi-table question answering with scalable context and symbolic extension. *arXiv preprint arXiv:2411.19504*.

Wang, X., (2016). A brief discussion on implementing national plans to enhance China's soybean competitiveness. *Heilongjiang Grain***,** no. 11, pp. 30-33.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R. and Cao, Y., (2023). Published. React: Synergizing reasoning and acting in language models. The eleventh international conference on learning representations, 2023 Kigali, Rwanda. Proceedings of the 11th International Conference on Learning Representations (ICLR).

Funding

This research received no external funding.

Conflicts of Interest

The authors declare no conflict of interest.

Acknowledgment

This paper is an output of the science project.

A Appendix

B Direct prompting example

Figure 6 shows a complete example of the direct prompting approach for a restaurant database question. Table rows beyond the second row are hidden with ”. . . ” .

C Tool-augmented reasoning prompting example

Figure 7 shows a complete example of the tool augmented reasoning prompting approach for a restaurant database question. Separate ChatCompletions messages are denoted using ”— role — ” in the figure. Sections labeled with ”— tool —” are not messages, but instead part of the tool definitions passed into the OpenAI ChatCompletions API.

D Structured agent workflow example

Figure 8shows a complete example of the structured agent workflow for a restaurant database question. The agent proceeds through three distinct stages with explicit completion markers.

Figure 6: Complete Direct Prompting Example

```
Please carefully analyze and answer the following single choice
question step by step.
        #restaurant
        ## geographic
        city, county, region
        alameda, alameda county, bay area
        antioch, contra costa county, bay area
        ...
        ## generalinfo
        id_restaurant, label, food_type, city, review
        209, mcneils bar & grill, american, santa clara, 2.0
        342, cinnabar, american, calistoga, 2.0
        ...
        ## location
id_restaurant, street_num, street_name, city
209, 2886.0, sycamore way, santa clara
        342, 1440.0, lincoln ave., calistoga
        Which street is tcby yogurt located in?
        A) n santa cruz ave
        B) sharon park dr
        C) moraga way
        D) solano ave
This question has only one correct answer. Please break down the
    question,
        evaluate each option, and explain why it is correct or incorrect.
        Conclude with your final choice on a new line formatted as
        'Answer: A/B/C/D or N/A', then put on the following line 'I AM DONE'.
```

Figure 7: Complete Tool-Augmented Reasoning Example

(a) System and User Messages

```
        --- message 1: system ---
You have access to tools to analyze database tables. Use them when
needed to answer questions accurately.
        --- message 2: user ---
Please carefully analyze and answer the following single choice question step
by step. You are provided with tools to see what tables and data
you have available, peek and read said data, and use simple python to read
and write said data.
You should avoid using your own knowledge and instead err for knowledge
retrieved by the tools or augmented by the tools.
        You should always check to see what data is available for your use.
        Which street is tcby yogurt located in?
        A) n santa cruz ave
        B) sharon park dr
        C) moraga way
        D) solano ave
This question has only one correct answer. Please break down the
    question,
        evaluate each option, and explain why it is correct or incorrect.
        Conclude with your final choice on a new line formatted as
        'Answer: A/B/C/D or N/A', then put on the following line 'I AM DONE'.
```

Figure 7: Complete Tool-Augmented Reasoning Example (continued)

(b) Tool Definitions

```
        --- tool 1: getTableNames (function) ---
        Retrieve all table names in the database.
        --- tool 2: peekTables (function) ---
        Peek the first 5 rows of each of the given table names.
        Args:
            tableNames (list[str]): A list of table names to peek.
        --- tool 3: readTables (function) ---
        Read the given table names and return the entire data as a string.
        Args:
            tableNames (list[str]): A list of table names to read.
        --- tool 4: executePython (function) ---
        Execute the given Python code in a sandboxed environment.
        You may create variables, modify variables, modify the database through
            the tables variable, and read them back through the other tools.
        You must print the results to stdout in order to see them.
A tables variable is already set up for you, so do not create a new
    variable by hardcoding values into the environment.
        You should never need to manually create a new variable for the
database.
        Args:
            code (str): Python code to execute. Must be a valid Python string.
        Available Variables (These are prepopulated for you in the environment):
            - tables: A dictionary mapping table names to pandas DataFrames
            - tables[table_name]: pandas DataFrame for the specified table
            - tables.keys(): List of all table names in the database
        Available Libraries:
            - pandas: For data manipulation and analysis
            - numpy: For numerical computations
            - Python builtins: All standard Python built-in functions
Note: You must use print() statements to output results. The output will be
    captured and returned.
        Security:
    - Only Python builtins, pandas, and numpy are available (no os, sys,
         subprocess, etc.)
            - No file system access
            - No network access
            - No ability to import dangerous modules
            - Execution times out after 2 minutes
```

Figure 8: Complete Structured Agent Workflow Example

(a) Stage 1: Database Exploration

```
        --- message 1: user ---
        # Stage 1: Database Exploration
        ## Available Tools
        - ‘peekTables‘ - Examine sample data from tables
        - ‘readTables‘ - Read full table contents
        - ‘executePython‘ - Run custom Python code
        ## Task
Explore the database to identify which tables and columns are relevant to
    answering the question.
        ## Available Tables
        Available tables: geographic, generalinfo, location
        ## Question
        Which street is tcby yogurt located in?
        ## What You Need to Do
1. **Examine the tables** that seem relevant to the question using
    ‘peekTables‘ and ‘readTables‘
2. **Identify the specific columns** that contain useful information for
    answering the question
        3. **Understand the data structure** and relationships between tables
        4. **Output your findings** in the exact format specified below
        ## Output Format
        You must output your findings in this exact format:
        ‘‘‘
        TABLES RELEVANT: table1, table2, table3
        TABLE table1 RELEVANT COLUMNS: col1, col2, col3
        TABLE table2 RELEVANT COLUMNS: col1, col4, col5
        TABLE table3 RELEVANT COLUMNS: col2, col6
NOTE TO NEXT STAGE: Brief summary of what you found relevant‘‘‘
        ## Important Notes
- **TABLES RELEVANT**: List only the table names that are actually
    relevant to answering the question
- **RELEVANT COLUMNS**: List only the columns that contain useful
    information for the question
- **NOTE TO NEXT STAGE**: This is a brief explanation of what you found and
    how it relates to the question. The next stage will use this
            information to understand what data to work with.
```

Figure 8: Complete Structured Agent Workflow Example (continued)

(b) Stage 2: Data Preparation

```
--- message 1: user ---
# Stage 2: Data Preparation
## Available Tools
- 'executePython' - Run custom Python code for data manipulation and
    verification
## Task
Create a combined table that contains all the information needed to
    answer the question.
## Question
Which street is tcby yogurt located in?
## Exploration Results from Stage 1
Relevant tables from exploration:
generalinfo:
    Rows: 96
    Columns: id_restaurant, label, food_type, city, review
    Sample data:
    Row 1: {'id_restaurant': 209, 'label': 'mcneils bar & grill',
        'food_type': 'american', 'city': 'santa clara', 'review': 2.0}
Row 2: {'id_restaurant':
    342, 'label': 'cinnabar', 'food_type':'american', 'city': 'cali
    stoga', 'review': 2.0}
location:
    Rows: 96
    Columns: id_restaurant, street_num, street_name, city
    Sample data:
Row 1: {'id_restaurant': 209, 'street_num': 2886.
    0, 'street_name':'sycamore way', 'city': 'santa clara'}
Row 2: {'id_restaurant': 342, 'street_num': 1440.
    0, 'street_name':'lincoln ave.', 'city': 'calistoga'}


## Stage 1's Note to You
NOTE TO NEXT STAGE: The 'generalinfo' table provides the restaurant names
    and their respective cities, while the 'location' table
        contains the street names and numbers associated with each
        restaurant. We can match the restaurant "tcby yogurt" from the
        'generalinfo' table with its corresponding street details in
    the 'location' table to identify its location.
```

Figure 8: Complete Structured Agent Workflow Example (continued)

(c) Stage 2: Data Preparation (continued)

```
        ## What You Need to Do
1. **Analyze the exploration results** to understand what data you have
    available
2. **Write Python code** that creates a combined table from the relevant
    tables (cleaned, filtered, and augmented)
    - **IMPORTANT**: Try to do all your data manipulation in a single
        executePython call
    - Each executePython call is independent - variables don't persist
        between calls
       3. **Use 'submitTable(dataframe, table_name)'** to store your new table
    - This is REQUIRED - it creates a table that Stage 3 will use to do final
        analysis on
    - Without using 'submitTable', Stage 3 won't have access to your
        prepared data
4. **Verify your generated table** using 'executePython' to check its
    structure and content
       5. **Output the table name** in the exact format specified below
       ## How the Tools Work Together
- **'executePython'**: Use this to write code that manipulates data and
    creates your combined table
- **'submitTable(dataframe, table_name)'**: This function provided in the
    executePython tool stores your DataFrame as a new table in the database
- **Important**: Stage 3 will only be able to access tables created with
    'submitTable'
- **Verification**: Use 'executePython' after creating the table to
    verify it looks correct
       ## Output Format
       You must output your results in this exact format:
       '''
       PREPARED TABLE NAME: your_table_name
NOTE TO NEXT STAGE: Brief description of what the prepared table contains'''
       ## Important Notes
- **PREPARED TABLE NAME**: The exact name of the table you created with
    'submitTable'
- **NOTE TO NEXT STAGE**: Explain what your prepared table contains and how it
    helps answer the question
- **Table Creation**: You MUST use 'submitTable' for Stage 3 to access your
    data
- **Verification**: Always verify your table with 'executePython' before
    marking completion
```

Figure 8: Complete Structured Agent Workflow Example (continued)

(d) Stage 3: Question Analysis

```
        --- message 1: user ---
        # Stage 3: Question Analysis
        ## Available Tools
        - ‘executePython‘ - Run custom Python code for analysis and calculations
        ## Task
        Analyze the prepared table to answer the question.
        ## Question
        Which street is tcby yogurt located in?
        A) n santa cruz ave
        B) sharon park dr
        C) moraga way
        D) solano ave
        ## Prepared Table from Stage 2
        Prepared table: tcby_yogurt_location
        Structure: 1 rows, columns: label, street_num, street_name
        Preview:
          label  street_num       street_name
tcby yogurt         13.0 n santa cruz ave
        ## Stage 2’s Note to You
         NOTE TO NEXT STAGE: The prepared table contains the location details
         (street number and street name) for the restaurant "tcby yogurt".
    This information helps answer the question of which street tcby
    yogurt is located on.
        ## What You Need to Do
        1. **Understand the prepared table** - examine its structure and
content
2. **Use ‘executePython‘** to perform any necessary calculations or
    analysis
        3. **Analyze the data** to find the answer to the question
        4. **Provide clear reasoning** for your conclusions
        5. **Output your final answer** in the exact format specified below
```

Figure 8: Complete Structured Agent Workflow Example (continued)

(e) Stage 3: Question Analysis (continued)

```
        ## How to Use the Tools
- **‘executePython‘**: Use this to write Python code that analyzes the
    prepared table
- **Data Access**: The prepared table is already available in the
    database
- **Analysis**: Write code to filter, aggregate, or calculate what you
    need
        - **Verification**: Use ‘executePython‘ to check intermediate results
        ## Output Format
        You must output your final answer in this exact format:
        ‘‘‘
        Answer: A/B/C/D or N/A
I AM DONE
‘‘‘
        ## Important Notes
        - **Answer**: Choose the correct option (A, B, C, D) or N/A if none
apply
        - **I AM DONE**: This marks the completion of all stages
- **Reasoning**: Explain your analysis process before giving the final
    answer
- **Use the Data**: Base your answer on the prepared table, not external
    knowledge
```

Figure 8: Complete Structured Agent Workflow Example (continued)

(f) Tool Definitions (Shared Across All Stages)

```
        --- tool 1: peekTables (function) ---
        Peek the first 5 rows of each of the given table names.
        Args:
            tableNames (list[str]): A list of table names to peek.
        --- tool 2: readTables (function) ---
        Read the given table names and return the entire data as a string.
        Args:
            tableNames (list[str]): A list of table names to read.
        --- tool 3: executePython (function) ---
        Execute the given Python code in a freshly created sandboxed
environment.
        You may create variables, modify variables, modify the database through
            the tables variable, and read them back through the other tools.
        You must print the results to stdout in order to see them.
A tables variable is already set up for you, so do not create a new
    variable by hardcoding values into the environment.
        You should never need to manually create a new variable for the
database.
        Args:
            code (str): Python code to execute. Must be a valid Python string.
        Available Variables (These are prepopulated for you in the environment):
            - tables: A dictionary mapping table names to pandas DataFrames
            - tables[table_name]: pandas DataFrame for the specified table
            - tables.keys(): List of all table names in the database
        Available Functions:
    - submitTable(dataframe, table_name): Store a DataFrame as a new
        table
                - dataframe: pandas DataFrame to store
                - table_name: string name for the new table
                - Returns: Success message with table info
                - Use this to persist your results for use in later stages
            - getAllDataframes(): Get all tables as a dictionary
            - getTableDataframe(table_name): Get a specific table as a DataFrame
        Available Libraries:
            - pandas: For data manipulation and analysis
            - numpy: For numerical computations
            - Python builtins: All standard Python built-in functions
        Recommended Usage:
            - 'tables: dict[str, pd.DataFrame] = getAllDataframes()'
            - 'df: pd.DataFrame = getTableDataframe('table_name')'
    - 'submitTable(df, 'my_table')' available in Stage 2 to save your
        results for Stage 3
    - Do all related work in a single executePython call when possible;
        environment does not persist between calls
Note: You must use print() statements to output results. The output will be
    captured and returned.
        Security:
    - Only Python builtins, pandas, and numpy are available (no os, sys,
        subprocess, etc.)
            - No file system access
            - No network access
            - No ability to import dangerous modules
            - Execution times out after 2 minutes
```