CodeUnify: A Flexible Code-Driven Framework for Reasoning over Multiple Structured Knowledge Sources

Anonymous ACL submission

Abstract

Unified structured data question answering task aims to utilize a unified model to answer natural language questions based on different types of structured data. Existing unified structured data question answering methods usually rely on predefined functionalities, which limits their ability to perform complex reasoning beyond these predefined operations. To overcome this limitation, we propose a flexible code-driven framework CodeUnify, which comprises two core modules: CodeSTEP and CRAFT. The CodeSTEP module is a paradigm that generates a complete executable Python code sequences containing a series of step-by-step code-based reasoning query operations based on the question, and CRAFT module (Code-based Reasoning for Adaptive Function Tailoring) can dynamically generate custom code functions for operations beyond the predefined function set, significantly enhancing the flexibility and capability in handling complex reasoning. Comprehensive empirical experiments on multiple structured datasets demonstrate that CodeUnify exhibits superior flexibility and remarkable improvements in complex reasoning scenarios compared to existing unified methods.

1 Introduction

Structured data, (e.g., tables, relational databases, knowledge graphs (KGs), and temporal KG (TKGs)) organize information in well-defined formats, enabling efficient storage, retrieval, and computation (Tan et al., 2024). In the era of Large Language Models (LLMs), structured data is an essential source of knowledge to improve factual accuracy, reduce hallucinations, and support complex reasoning capabilities (Yang et al., 2024a).

Natural language reasoning over structured data, with growing applications across various domains, is important but challenging. While specialized approaches have been developed for specific data structures like tables or knowledge graphs, real-world scenarios often require reasoning across heterogeneous data sources simultaneously, driving interest in unified approaches that can handle multiple structured data formats. For example, retrieval-based unified methods like StructGPT (Jiang et al., 2023a) and Readi (Cheng et al., 2024) are proposed by accessing the raw data by predefined functions. To further enhance the trustworthiness of the unified method, TrustUQA (Zhang et al., 2025) is proposed, which gets the answer through an unified query language without inputting many raw data into the LLM. The capability of these methods are limited to the predefined function callings. However, tasks with complex computation and advanced logical reasoning often involves functions beyond the predefined ones, raising significant challenges for these methods.

Method such as Program of Thought (PoT) (Chen et al., 2022) have shown that structured code execution can effectively enhance complex reason-

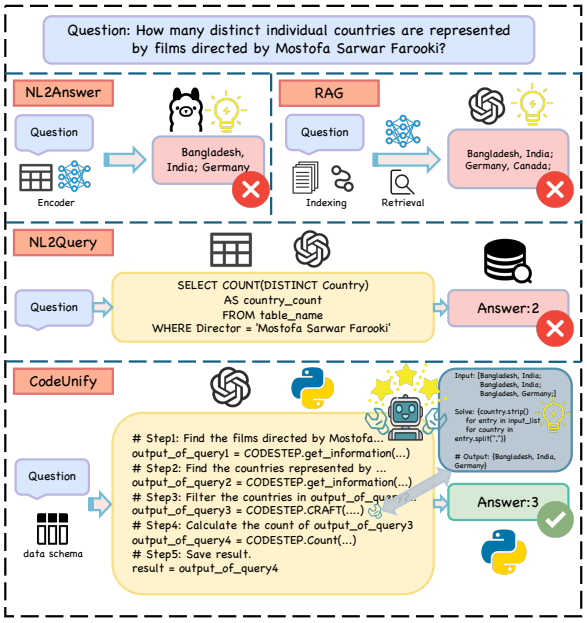

Figure 1: Comparison of NL2Answer, RAG, NL2SQL, and our proposed CodeUnify.

ing capability by decomposing problems into explicit computational steps with code. We believe the capability of code-based approaches to represent both process logic and data manipulation can help LLMs to handle complex tasks better.

Inspired by the Program of Thought (Chen et al., 2022) approach, we explored the feasibility of solving unified structured data question answering task based on executable code. Given the limitations of the current approach, we propose a code-based framework to improve the flexibility of unified method, named **CodeUnify**. Our CodeUnify framework consists of two core modules: Code-based Stepwise Transparent Execution Paradigm (**CodeSTEP**) and Code-based Reasoning for Adaptive Function Tailoring (**CRAFT**).

CodeSTEP is a custom code paradigm that generates complete executable Python code without intermediate parsing. We introduced CodeSTEP to address the limitations of existing methods, which generate natural language functions that require further parsing steps and they are limited to using predefined functions. By directly generating executable code, CodeSTEP provides an explainable problem-solving step, maintaining the trustworthiness of query-based methods while significantly simplifying the QA process and providing a basis for more flexible operations.

CRAFT is an innovative module designed to dynamically handle scenarios beyond the capabilities of predefined functions. We proposed CRAFT to overcome the fundamental limitation of existing unified methods that can only operate within the scope of predefined functions. CRAFT can generate dedicated code for specific reasoning steps and seamlessly integrate with the main CodeSTEP code execution. This design enhances the flexibility of the CodeUnify framework while maintaining its structured and verifiable QA framework, making it more suitable for handling complex reasoning tasks. Figure 1 illustrates a comparison between our proposed framework and other methods.

CodeUnify implements a framework that allows multiple LLMs to collaborate. Multiple LLMs that are the same or different are allowed to collaborate in the same code environment. With this design, individual models can focus on different aspects of the reasoning process, thus allowing for improved reasoning performance through the collaboration of multiple LLMs.

In summary, contributions of this paper are:

- We present the CodeUnify, a flexible and trustworthy code-based framework for unified structured data question answering, which includes CodeSTEP module for code-based step-by-step reasoning and CRAFT module for dynamic function customization.
- To the best of our knowledge, we are the first code-based implementation of unified reasoning across different structured knowledge sources in a custom form.
- We conducted comprehensive empirical studies based on 7 datasets of 2 structured data types. Results prove that our approach significantly outperforms existing unified methods and achieves competitive results with data-specific methods, especially for complex reasoning scenarios that are difficult to solve with predefined functions.

2 Preliminary

Structured Data Representation. Following previous work (Zhang et al., 2025), We consider 2 common types of structured data, **Tables** and **Knowledge Graphs** (KGs). TrustUQA (Zhang et al., 2025) is a trustful framework for unified structured data question answering. It adopts a unified knowledge representation method called Condition Graph (CG) to handle multiple types of structured data simultaneously. A Condition Graph is formally defined as $\mathcal{CG} = \{\mathcal{N}, \mathcal{T}\}$, where $\mathcal{N}$ is the set of nodes representing entities, relationships, properties, or numerical values. $\mathcal{T} = \{(node_1, node_2, condition) \mid node_1 \in \mathcal{N}, node_2 \in \mathcal{N}\}$ is a collection of condition triples, where $condition = \{\, node_k \in \mathcal{N} \mid k = 1, \ldots, |\mathcal{N}| \,\}$ is a list of nodes (possibly empty) specifying the conditions under which $node_1$ connects to $node_2$. TrustUQA uses a two-layer querying approach, An LLM generates simplified functions for composing LLM queries, these queries are translated into execution queries that can be executed on the $\mathcal{CG}$.

3 Methodology

3.1 Overview

CodeUnify framework is build upon the key idea of code-based reasoning, that each reasoning step is expressed as executable Python code, ensuring precision and eliminating ambiguity. Code is expressive, making the framework can handle operations

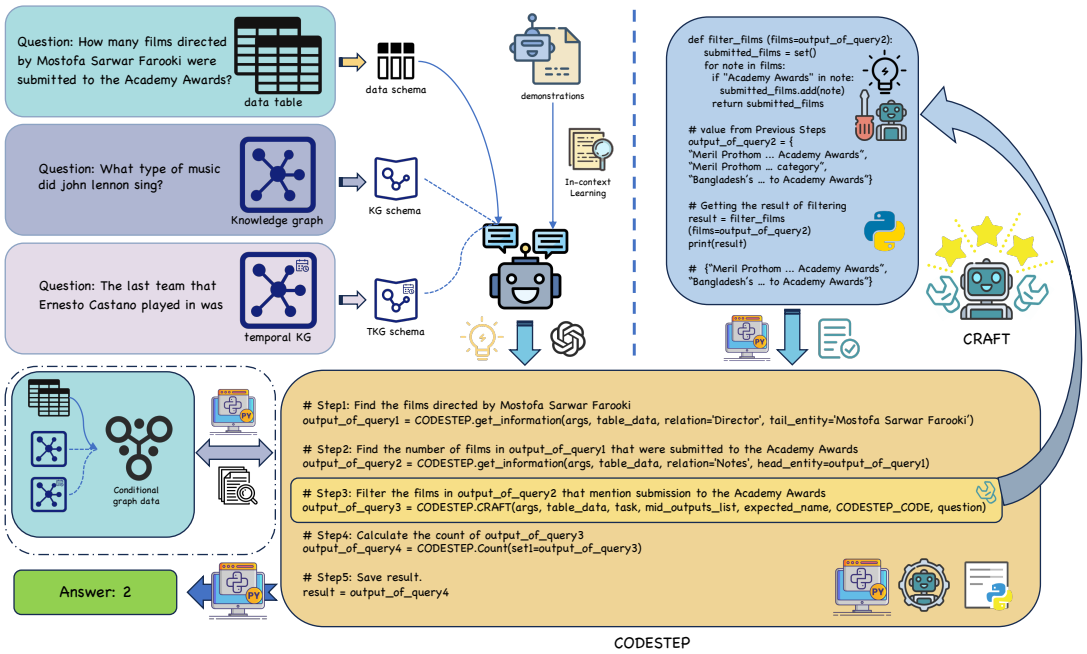

Figure 2: Overview of CodeUnify framework.

beyond predefined functions through dynamically generated code for custom functions. Figure 2 illustrates the reasoning process of the CodeUnify framework.

Specifically, given a structured data source $\mathcal{D}$ and a natural language question q, we firstly transforms $\mathcal{D}$ into a data source schema $\mathcal{D}_{schema}$ and a conditional graph representation $\mathcal{D}_{cg}$. Based on $\mathcal{D}_{schema}$ and q , we utilize an LLM with parameter θ and few-shot query-code sequence prompt p to generate an executable code sequence $\mathcal{C} = \{c_1, c_2, ..., c_n\}$. This code sequence is then executed as a complete program using $\mathcal{D}_{cg}$ to directly obtain the answer a to question q. These two processes can be represented as:

$$f_\theta(\mathcal{D}_{schema}, q, p) \mapsto \mathcal{C} \tag{1}$$

$$\text{Execute}(\mathcal{C}, \mathcal{D}_{cg}) \mapsto a \tag{2}$$

3.2 Code-based Stepwise Transparent Execution Paradigm

3.2.1 Code Generation of CodeSTEP

To achieve code-based reasoning, we propose a **Code**-based **S**tepwise **T**ransparent **E**xecution **P**aradigm (**CodeSTEP**) module, which can be formalized as following two steps: ***(1) Query Analysis.*** For a given natural language question q and data source schema $\mathcal{D}_{schema}$, we firstly using the LLM with parameter θ construct a reasoning path $\mathcal{P} = \{s_1, s_2, ..., s_n\}$. ***(2) Step Construction.*** For each reasoning step s_i, we construct a corresponding code operation c_i for data schema $\mathcal{D}_{schema}$ that implements the reasoning step, thereby constructing a complete code operation sequence $\mathcal{C} = \{c_1, c_2, ..., c_n\}$. The entire process can be expressed as:

$$f_\theta(\mathcal{D}_{schema}, q, p) \mapsto \mathcal{C}, \tag{3}$$

where p is the prompt with few-shot query-code sequence.

3.2.2 Supporting Operations

Conditional Graph Query Operations. The primary tools for data interaction with $\mathcal{D}_{cg}$ in CodeSTEP is the conditional graph query operations. This operation can be formulated as:

$$g(\mathcal{D}_{cg}, \mathcal{R}, \mathcal{E}_h, \mathcal{E}_t, \mathcal{K}, \mathcal{V}, \delta_t, \delta_v) \mapsto \mathcal{S}, \tag{4}$$

where $\mathcal{D}_{cg}$ is the conditional graph data source, $\mathcal{R}$ is the relation (column or edge type), $\mathcal{E}_h$ is the head entity set (row identifiers or source nodes), $\mathcal{E}_t$ is the tail entity (column value or target node), $\mathcal{K}$ is the key (column or attribute), $\mathcal{V}$ is the value of $\mathcal{K}$, δ_t and δ_v are comparison operators, $\mathcal{S}$ is the resulting set. This operation consists of two primary query modes: ***(1) Relation-Tail Entity Mode*** $(\mathcal{R} + \mathcal{E}_t)$: Returns the head entity set $\mathcal{E}_h$ corresponding to tail entity $\mathcal{E}_t$ in relation $\mathcal{R}$. Specifically, when the data source $\mathcal{D}$ is table, the tail entity $\mathcal{E}_t$ is a specific column value, and this mode returns the head entity set

Operation	Definition
Set Operations	
Union	$f_{\text{union}}(\mathcal{S}_1, \ldots, \mathcal{S}_n) \mapsto \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_n$
Intersection	$f_{\text{intersect}}(\mathcal{S}_1, \ldots, \mathcal{S}_n) \mapsto \mathcal{S}_1 \cap \cdots \cap \mathcal{S}_n$
Difference	$f_{\text{diff}}(\mathcal{S}_1, \mathcal{S}_2) \mapsto \mathcal{S}_1 - \mathcal{S}_2$
Negation	$f_{\text{neg}}(\mathcal{D}_{cg}, \mathcal{S}_1) \mapsto \mathcal{D}_{cg} - \mathcal{S}_1$
Calculator Operations	
Min	$f_{\min}(\mathcal{S}) \mapsto \{\min(\mathcal{S})\}$
Max	$f_{\max}(\mathcal{S}) \mapsto \{\max(\mathcal{S})\}$
Mean	$f_{\text{mean}}(\mathcal{S}) \mapsto \{\frac{1}{\lvert\mathcal{S}\rvert} \sum_{x \in \mathcal{S}} x\}$
Count	$f_{\text{count}}(\mathcal{S}) \mapsto \{\lvert\mathcal{S}\rvert\}$
Sum	$f_{\text{sum}}(\mathcal{S}) \mapsto \{\sum_{x \in \mathcal{S}} x\}$

Table 1: Details of Predefined Calculation Operations

$\mathcal{E}_h$ corresponding to that value, which are the row identifiers. When the data source $\mathcal{D}$ is KG, $\mathcal{E}_t$ is the tail entity, and this mode returns the head entity set $\mathcal{E}_h$ corresponding to relation $\mathcal{R}$. ***(2) Relation-Head Entity Mode*** ($\mathcal{R} + \mathcal{E}_h$): Returns the tail entity set $\mathcal{E}_t$ corresponding to head entity $\mathcal{E}_h$ in relation $\mathcal{R}$. Specifically, when the data source $\mathcal{D}$ is table, the head entity $\mathcal{E}_h$ is a row identifier, and this mode returns the tail entity set $\mathcal{E}_t$ corresponding to that identifier, which are the column values. When the data source $\mathcal{D}$ is KG, $\mathcal{E}_h$ is the head entity, and this mode returns the tail entity set $\mathcal{E}_t$ corresponding to relation $\mathcal{R}$.

Calculation Operations. In addition to the aforementioned data query operation g, we also offer common set operation functions including f_{union}, $f_{\text{intersect}}$, f_{diff}, f_{neg}, and algebraic calculator operation functions including $f_{\min}$, $f_{\max}$, f_{mean}, f_{count}, f_{sum}. The operational rule of these operations are detailed in Table 1.

Predefined Function Set. The conditional graph query operation and all calculation operations are collectively referred to a predefined function set $\mathcal{F}_{\text{predefined}}$ that can be defined as:

$$\mathcal{F}_{\text{predefined}} = \{g, f_{\text{union}}, f_{\text{intersect}}, f_{\text{diff}}, f_{\text{neg}}, f_{\min}, f_{\max}, f_{\text{mean}}, f_{\text{count}}, f_{\text{sum}}\}. \quad (5)$$

Each step s_i in the reasoning path can be implemented using one of these predefined functions:

$$c_i = f_i(\mathcal{D}_{cg}, \{r_j \mid j \in \cup_{k=0}^{i-1} \mathcal{J}_k\}), \quad \text{where } f_i \in \mathcal{F}_{\text{predefined}}. \quad (6)$$

However, predefined functions may not cover all possible operations required to answer complex questions. This limitation motivates the need for a more flexible method that can dynamically generate custom functions for specific operations.

3.3 Code-based Reasoning for Adaptive Function Tailoring

We propose a **C**ode-based **R**easoning for **A**daptive **F**unction **T**ailoring (**CRAFT**) module, which extends CodeSTEP module to address the inflexibility of predefined operations, through dynamically generating custom code-based functions for operations not covered by predefined functions.

CRAFT Implement. As a specialized code generation system, CRAFT module utilizes the LLM with parameter θ' and few query-code sequence shots prompt p_c to translate current task descriptions into executable operation f_c, denoted as:

$$f_{\theta'}(q, \mathcal{C}, \mathcal{T}_i, \mathcal{R}_{prev}, \mathcal{F}_{expected}, p_c) \mapsto f_c. \quad (7)$$

There are five key input components to create custom functions tailored to the current step i: ***(1) original question*** q providing the overall context and goal of the reasoning step; ***(2) complete code sequence*** $\mathcal{C}$ helping understand the role of current step i; ***(3) current task description*** $\mathcal{T}_i$ providing clear functional requirements; ***(4) previous steps and results*** $\mathcal{R}_{prev} = \{r_1, r_2, ..., r_{i-1}\}$ helping better understand the task background and data characteristics; ***(5) expected function signature*** Leverages the LLM that generates the CodeSTEP code understanding of the current task to convey the expected function signature $\mathcal{F}_{expected}$ to the CRAFT framework, for CRAFT to reference.

Seamlessly Integration with CodeSTEP. Following are key steps of CRAFT: **(1) Delegated Tasks**, while the code sequence $\mathcal{C}$ generated by CodeSTEP module is being executed, if an operation is encountered that lacks predefined operations from $\mathcal{F}_{\text{predefined}}$, it delegates the task to CRAFT module. **(2) Context Analysis and Reasoning**, CRAFT analyzes the current task context based on the original question, overall code framework, previous results, current task description, and expected function signature. **(3) Function Code Generation**, through reasoning about the current step's requirements, CRAFT generates a self-contained Python function to implement the functionality needed for the current step, as shown in Equation 7. Therefore, we can update Equation 6 as:

$$c_i = \begin{cases} f_i(\mathcal{D}_{cg}, \mathcal{R}_i), & f_i \in \mathcal{F}_{\text{predefined}} \\ f_{\theta'}(q, \mathcal{C}, \mathcal{T}_i, \mathcal{R}_{\text{prev}}, \mathcal{F}_{\text{expected}}), & f_i \notin \mathcal{F}_{\text{predefined}} \end{cases} \quad (8)$$

where $\mathcal{R}_i = \{r_j \mid j \in \cup_{k=0}^{i-1} \mathcal{J}_k\}$ represents the previous results that step i depends on. **(4) Function Code Execution**, the generated function is

executed to obtain the results r_{step} needed for the current step, as shown in Equation 9. **(5) Results Return**, after obtaining the result r_{step} for the current step, it is returned to the main CodeSTEP code execution process, can seamlessly integrate with the complete CodeSTEP code execution.

$$r_{step} = \text{Execute}(f_c, \{r_j | j \in \cup_{k=0}^{i-1} \mathcal{J}_k\}) \quad (9)$$

This integration allows CodeUnify to dynamically extend its capabilities beyond predefined functions, addressing complex queries that require custom operations.

3.4 Code Execution

While internally, Each individual code step c_i returns an intermediate result r_i that may serve as input to subsequent steps, can be concisely represented as:

$$r_i = \text{Execute}(c_i, \mathcal{D}_{cg}, \{r_j | j \in \cup_{k=0}^{i-1} \mathcal{J}_k\}) \quad (10)$$

where $\mathcal{J}_k$ is the subset of previous step indices that step k depends on.

This decomposition enables transparent reasoning and facilitates error detection and correction. The complete reasoning process is ultimately executed through the complete code sequence $\mathcal{C}$, returns the final step result r_n, which directly corresponds to the final answer a to the original question:

$$r_n = \text{Execute}(\mathcal{C}, \mathcal{D}_{cg}) \mapsto a \quad (11)$$

4 Experiments

We conduct various experiments to answer the following three key questions: **RQ1**: How effective is CodeUnify in multiple structured data question answering tasks compared to baselines? **RQ2**: How does each component in CodeUnify framework contribute to the overall performance? **RQ3**: Does the CRAFT module effectively handle complex reasoning beyond predefined functions?

4.1 Experimental Setup

Datasets and Evaluation Metrics. For KGQA, we use WebQSP (Yih et al., 2016) with Hit@1 as the evaluation metric. For TableQA, we use WikiSQL (Zhong et al., 2017), WikiTableQuestions (WTQ) (Pasupat and Liang, 2015) with Denotation Accuracy (DA) (Jiang et al., 2023a) as the metric, and TableBench (Wu et al., 2025) with Exact Match (EM) as the metric. To further evaluate the CRAFT module, we constructed two datasets, named WikiSQL-E and WTQ-E, and used Calling Rate, Calling Denotation Accuracy (CDA), DA, and F1-score as evaluation metrics on the WikiSQL-E, WTQ-E, and TableBench datasets. More details are provided in Appendix A.

Baselines. We evaluate CodeUnify against a comprehensive set of baselines. For the WebQSP (Yih et al., 2016) dataset, we compare with data type specific models including DecAF (Yu et al., 2022), KB-Binder (Li et al., 2023), KB-Coder(Nie et al., 2024)), UniKGQA (Jiang et al., 2022), TIARA (Shu et al., 2022), ReasoningLM (Jiang et al., 2023b)and AgentBench (Liu et al., 2023b). For the WikiSQL (Zhong et al., 2017) and WTQ (Pasupat and Liang, 2015) datasets, we compare with table-specific models including TAPEX (Liu et al., 2021), DATER (Ye et al., 2023), TAPAS (Herzig et al., 2020) and MAPO (Liang et al., 2018). And we compare with unified models on WebQSP (Yih et al., 2016), WikiSQL (Zhong et al., 2017) and WTQ (Pasupat and Liang, 2015), including UnifiedSKG (Xie et al., 2022), StructGPT(Jiang et al., 2023a), Readi (Cheng et al., 2024) and TrustUQA (Zhang et al., 2025). To evaluate the CRAFT module on WikiSQL-E, WTQ-E, and TableBench (Wu et al., 2025) datasets, we use TrustUQA (Zhang et al., 2025) as the primary baseline, implemented with the same LLM as our framework for fair comparison. For TableBench experiments, we adopt various baselines from the original TableBench (Wu et al., 2025), including both open-source and closed-source methods with different prompting strategies, as well as TrustUQA (Zhang et al., 2025) implemented with various LLMs. Detailed descriptions of all baseline methods are provided in Appendix B.

Implementation. We implement our framework using multiple LLMs as the reasoning engine including GPT-3.5-turbo, GPT-4o-mini, GPT-4o, and GPT-4.1. The specific usage of each model will be detailed in the experimental sections. The CodeUnify framework is implemented in a Python environment, with the CRAFT module dynamically generating and executing custom Python functions at runtime. All experiments were conducted using the OpenAI API for LLM access. The prompt templates used in our experiments will be provided in the Appendix D. For in-context learning, we used 10 demonstrations for the CodeSTEP module and 2 demonstrations for the CRAFT module.

Method	WebQSP Hit@1 (%)
Data Type Specific Models	
UniKGQA (Jiang et al., 2022)	75.1
DecAF (Yu et al., 2022)	**78.7**
TIARA (Shu et al., 2022)	76.7
ReasoningLM (Jiang et al., 2023b)	78.5
AgentBench (Liu et al., 2023b)	47.8
KB-Binder (Li et al., 2023)	68.9
KB-Coder (Nie et al., 2024)	77.2
Unified Models	
UnifiedSKG (Xie et al., 2022)	80.7
StructGPT (Jiang et al., 2023a)	69.6
Readi (Cheng et al., 2024)	74.3
TrustUQA (Zhang et al., 2025)	83.5
CodeUnify (Ours)	**85.2**

Table 2: Experimental results on WebQSP dataset.

Method	WikiSQL DA (%)	WTQ DA (%)
Data Type Specific Models		
MAPO (Liang et al., 2018)	72.6	43.8
TAPAS (Herzig et al., 2020)	83.6	48.8
TAPEX (Liu et al., 2021)	**89.5**	57.5
DATER (Ye et al., 2023)	–	**65.9**
Unified Models		
UnifiedSKG (Xie et al., 2022)	86.0	49.3
StructGPT (Jiang et al., 2023a)	65.6	52.2
Readi (Cheng et al., 2024)	66.2	**61.7**
TrustUQA (Zhang et al., 2025)	85.7	46.7
CodeUnify (Ours)	**86.1**	45.8

Table 3: Results on WikiSQL and WTQ datasets.

4.2 KGQA Results (RQ1)

On the WebQSP dataset, we conducted experiments using the CodeUnify framework without the CRAFT module to explore the impact of the code-based formulation itself on structured data question answering tasks. These experiments were implemented based on the GPT-3.5-turbo-0613.

Table 2 shows results on WebQSP. CodeUnify achieves 85.21% Hit@1 precision, outperforming all baselines including specialized models like DecAF (Yu et al., 2022) and unified models like TrustUQA(Zhang et al., 2025). The improvements demonstrate the feasibility and potential of using executable code for structured data question answering tasks, providing an effective new approach for structured data question answering.

4.3 Table QA Results (RQ1)

Table 3 shows the results on WikiSQL and WTQ with the GPT-4o-mini model.

On WikiSQL, CodeUnify achieves 86.1% Denotation Accuracy (DA), slightly outperforming other unified models including UnifiedSKG (Xie et al., 2022) and TrustUQA (Zhang et al., 2025). This demonstrates that our approach maintains competitive performance on table QA tasks without sacrificing generalizability. On WTQ, CodeUnify achieves 45.8% DA, which is lower than some specialized and unified models.

It's important to recognize that methods based on executable programs require high precision throughout the entire solution process (Wu et al., 2025; Chen et al., 2022). For simpler reasoning tasks requiring precision, our approach may not show advantages. We believe our method is more suitable for complex reasoning scenarios and requires certain capabilities from the base model. However, with the rapid development of large language models and their rapidly improving capabilities, we believe our method aligns with development trends and has significant potential. We will validate this in subsequent experiments examining our method's capabilities in complex reasoning scenarios and the impact of base models.

Table 4 shows CodeUnify's performance comparison with TableBench baselines (Wu et al., 2025) and the TrustUQA (Zhang et al., 2025) method on the TableBench dataset. On TableBench, CodeUnify significantly outperforms TrustUQA and shows clear advantages compared to PoT-based baselines. With GPT-4o as the base model, CodeUnify achieves 68.75% and 51.01% accuracy on Fact Checking and Numerical Reasoning, surpassing TrustUQA by 6.25 and 21.46 percentage points respectively. The performance improvements are particularly notable on the Numerical Reasoning task, highlighting the effectiveness of CodeUnify for complex mathematical reasoning.

4.4 Ablation Study (RQ2)

Table 5 presents an ablation study on TableBench FactChecking and Numerical Reasoning tasks, with GPT-4o as the base model. Removing CRAFT module reduces performance on Fact Checking from 68.75% to 65.26% and on Numerical Rea-

Methods	TableBench	
	FC	NR
Open-source based		
TableLLM-Qwen2-7B+PoT	10.59	10.34
TableLLM-Llama3.1-8B+PoT	25.67	28.64
Qwen2-72B+PoT	56.37	**41.33**
Llama3.1-70B+PoT	**59.05**	34.04
Close-source based		
Qwen-Max+PoT	50.42	32.80
Deepseek-Chat-V2+PoT	57.48	45.96
gpt-3.5-turbo+PoT	60.92	42.09
gpt-3.5-turbo+TCoT	59.95	23.45
gpt-4o+PoT	**62.31**	**47.83**
TrustUQA+GPT-3.5-turbo	50.00	20.20
TrustUQA+GPT-4o-mini	55.21	21.72
TrustUQA+GPT-4o	62.50	29.55
Ours		
CodeUnify+GPT-3.5-turbo	61.46	38.63
CodeUnify+GPT-4o-mini	64.58	40.66
CodeUnify+GPT-4o	**68.75**	**51.01**

Table 4: Experimental results on the TableBench dataset. 'FC' and 'NR' represents Fact Checking subset and Numerical Reasoning subset, respectively.

Methods	Fact Checking		Num-Reasoning	
	EM (%)	F1 (%)	EM (%)	F1 (%)
CodeUnify (ours)	**68.75**	**71.60**	**51.01**	**51.80**
w/o CRAFT	65.26	67.71	45.85	46.29
w/o CodeSTEP	59.38	63.16	18.43	19.69

Table 5: The results of ablation study.

soning from 51.01% to 45.85%. This confirms that CRAFT contributes significantly to the framework's ability to handle complex reasoning tasks. Removing CodeSTEP module causes performance to drop to 59.38% on Fact Checking and 18.43% on Numerical Reasoning. The particularly severe degradation on Numerical Reasoning (32.58 percentage points) highlights that CodeSTEP's structured code-based reasoning approach is essential for complex mathematical operations.

These validate that both components are crucial, with CodeSTEP providing the foundational reasoning structure and CRAFT offering critical flexibility for complex cases.

4.5 CRAFT for Complex Reasoning (RQ3)

Table 6 provides a detailed comparison between CodeUnify and TrustUQA across WikiSQL-E, WTQ-E, and TableBench datasets using different base models, evaluated with metrics including calling rate, Calling Denotation Accuracy (CDA), Denotation Accuracy (DA), and F1 score.

A striking observation is that CodeUnify consistently maintains significantly higher CDA across all datasets and models. For instance, on the Numerical Reasoning dataset using GPT-4o, CodeUnify achieves a CDA of 50.0% compared to TrustUQA's mere 6.17%, representing a remarkable improvement of 43.83 percentage points. Similarly, on the WTQ-E dataset using GPT-4.1, CodeUnify reaches a CDA of 57.44% versus TrustUQA's 20.10%, an improvement of 37.34 percentage points. These demonstrate CRAFT can effectively handling cases requiring reasoning beyond predefined functions.

CodeUnify generally exhibits lower calling rates than TrustUQA (e.g., 2.36% vs. 55.52% on WikiSQL-E with GPT-4.1), indicating that within the CodeUnify framework, the model can more precisely determine when custom functions are needed and implement them more effectively. Improvements in overall DA and F1 metrics show notable enhancements on majority datasets, with particularly significant gains on complex reasoning tasks. On TableBench Numerical Reasoning using GPT-4o, CodeUnify achieves a DA of 51.01% compared to TrustUQA's 29.55%, representing a 21.46 percentage point improvement.

5 Related Work

Structured Data Question Answering is increasingly important in human-computer interaction scenarios across healthcare (Yang et al., 2024b; Huang et al., 2021) , finance (Liu et al., 2023a; Zhu et al., 2021), and information retrieval (Zhang et al., 2022). Structured data reasoning refers to the task of answering natural language questions by leveraging structured data sources (Huang et al., 2024). Research in this field has evolved along two primary directions. Single data-type specific methods focus on reasoning over a specific data structure, such as tables (Zha et al., 2023) or KGs (Song et al., 2023). Recent advancements include KB-Coder (Nie et al., 2024), which utilizes a code-based paradigm for KG reasoning with in-context learning, and DATER (Ye et al., 2023), which leverages demonstrations to enhance

Models	Datasets	calling rate (%) (TrustUQA/Ours)	CDA (%) (TrustUQA/Ours)	DA (%) (TrustUQA/Ours)	F1 (%) (TrustUQA/Ours)
gpt-3.5-turbo	WikiSQL-E	70.92/7.15	3.32/**68.24** ↑↑	70.00/67.79	70.40/68.28
	WTQ-E	87.35/50.94	11.12/**39.87** ↑↑	32.02/**34.40** ↑	33.45/**36.17** ↑
	FactChecking	16.67/16.67	0.00/**18.75** ↑	50.00/**61.46** ↑	56.61/**65.44** ↑
	Numerical Reasoning	65.66/50.47	4.23/**33.33** ↑↑	20.20/**38.63** ↑↑	20.87/**40.10** ↑↑
gpt-4o-mini	WikiSQL-E	66.55/2.86	8.33/**61.29** ↑↑	74.54/**79.04** ↑	75.06/**79.66** ↑
	WTQ-E	84.63/56.48	14.10/**41.15** ↑↑	36.57/**38.85** ↑	38.21/**40.71** ↑
	FactChecking	22.92/25.00	0.00/**33.33** ↑↑	55.21/**64.58** ↑	58.39/**70.19** ↑
	Numerical Reasoning	62.63/50.51	2.82/**33.00** ↑↑	21.72/**40.66** ↑↑	22.73/**42.57** ↑↑
gpt-4o	FactChecking	22.92/15.62	4.55/**20.00** ↑	62.50/**68.75** ↑	64.72/**71.60** ↑
	Numerical Reasoning	61.36/51.01	6.17/**50.00** ↑↑	29.55/**51.01** ↑↑	31.18/**51.80** ↑↑
gpt-4.1	WikiSQL-E	55.52/2.36	6.98/**53.57** ↑↑	79.44/**87.34** ↑	79.52/**87.58** ↑
	WTQ-E	76.79/60.16	20.10/**57.44** ↑↑	41.14/**55.02** ↑	42.08/**56.34** ↑

Table 6: Experimental results on the WikiSQL-E, WTQ-E, Fact Checking and Numerical Reasoning datasets. "↑" shows the improvement compared with TrustUQA.

table reasoning in large language models. In contrast, unified-type approaches aim to support reasoning across multiple structured data types simultaneously (Khashabi et al., 2020). Notable examples include UnifiedSKG (Xie et al., 2022), which integrates multiple structured knowledge formats through a seq2seq framework. Such unified frameworks are crucial for real-world applications where information is distributed across heterogeneous data sources (Chen et al., 2020). Single-type and unified methods typically adopt one of the following three paradigms: NL2Answer, NL2Query and RAG. More related work is described in Appendix C.

LLM-based Unified Frameworks. With the rapid advancement of large language models, an increasing number of works have attempted to leverage LLMs to implement unified structured data question answering task, offering new possibilities for handling diverse data formats within a single framework. StructGPT (Jiang et al., 2023a) is an iterative reading-then-reasoning framework that uses LLMs to generate answers or next reasoning steps based on collected evidence. Readi (Cheng et al., 2024) is a reasoning-path-editing framework that collects KG evidence based on edited reasoning paths and generates answers based on the evidence and questions using an LLM. TrustUQA (Zhang et al., 2025) presents a trustworthy framework that uses Conditional Graph and a two-layer query approach to uniformly support task scenarios for tables, KGs, and TKGs.

Code-based Reasoning. Recent research has shown that code-based approaches can effectively enhance reasoning capabilities in LLMs (Yang et al., 2025). Program of Thought (PoT) (Chen et al., 2022) demonstrates that executable code can represent complex problems into manageable computational steps. However, current code-based methods are primarily applied to mathematical reasoning and other domains requiring procedural thinking, while our CodeUnify framework extends this paradigm to unified structured data QA.

6 Conclusion

In this paper, we introduced an effective and flexible code-based framework for unified structured data question answering, called CodeUnify. Our framework includes two core modules, CodeSTEP and CRAFT. It generates and executes code sequences that directly answer natural language questions over various structured data types. Through experiments across diverse datasets, we demonstrated our framework's effectiveness, particularly on complex reasoning tasks. CodeUnify offers a new effective solution to unified structured data question answering. The performance improvements with stronger base models suggest our approach will benefit from continued LLM advancements. As large language models continue to evolve, we believe our code-based method aligns well with future AI development trends. Looking forward, we plan to extend our framework to more structured data formats, and further enhance its reasoning capabilities.

Limitations

While our CodeUnify framework shows promising results, we acknowledge several limitations of our approach: (1) due to the high precision requirements of executable code methods (Wu et al., 2025; Chen et al., 2022), the code-based execution paradigm requires maintaining high precision throughout the entire solution process with strong LLMs. (2) our experimental results on WikiSQL and WTQ indicate that CodeUnify may not show obvious advantages on relatively simpler tasks compared to specialized models. The performance gains of our approach are more pronounced on complex reasoning scenarios that require functionality beyond predefined operations. These limitations present opportunities for future research to enhance the robustness, efficiency, and generalization of code-based approaches for unified structured data question answering.

References

Patrice Béchard and Orlando Marquez Ayala. 2024. Reducing hallucination in structured outputs via retrieval-augmented generation. *arXiv preprint arXiv:2404.08189*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Wenhu Chen, Hanwen Zha, Zhiyu Chen, Wenhan Xiong, Hong Wang, and William Wang. 2020. Hybridqa: A dataset of multi-hop question answering over tabular and textual data. *arXiv preprint arXiv:2004.07347*.

Sitao Cheng, Ziyuan Zhuang, Yong Xu, Fangkai Yang, Chaoyun Zhang, Xiaoting Qin, Xiang Huang, Ling Chen, Qingwei Lin, Dongmei Zhang, and 1 others. 2024. Call me when necessary: Llms can efficiently and faithfully reason over structured environments. *arXiv preprint arXiv:2403.08593*.

Jonathan Herzig, Paweł Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. Tapas: Weakly supervised table parsing via pre-training. *arXiv preprint arXiv:2004.02349*.

Sirui Huang, Yanggan Gu, Xuming Hu, Zhonghao Li, Qing Li, and Guandong Xu. 2024. Reasoning factual knowledge in structured data with large language models. *arXiv preprint arXiv:2408.12188*.

Xiaofeng Huang, Jixin Zhang, Zisang Xu, Lu Ou, and Jianbin Tong. 2021. A knowledge graph based question answering method for medical domain. *PeerJ Computer Science*, 7:e667.

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Wayne Xin Zhao, and Ji-Rong Wen. 2023a. Structgpt: A general framework for large language model to reason over structured data. *arXiv preprint arXiv:2305.09645*.

Jinhao Jiang, Kun Zhou, Wayne Xin Zhao, Yaliang Li, and Ji-Rong Wen. 2023b. Reasoninglm: Enabling structural subgraph reasoning in pre-trained language models for question answering over knowledge graph. *arXiv preprint arXiv:2401.00158*.

Jinhao Jiang, Kun Zhou, Wayne Xin Zhao, and Ji-Rong Wen. 2022. Unikgqa: Unified retrieval and reasoning for solving multi-hop question answering over knowledge graph. *arXiv preprint arXiv:2212.00959*.

Haemin Jung and Wooju Kim. 2020. Automated conversion from natural language query to sparql query. *Journal of Intelligent Information Systems*, 55(3):501–520.

Daniel Khashabi, Sewon Min, Tushar Khot, Ashish Sabharwal, Oyvind Tafjord, Peter Clark, and Hannaneh Hajishirzi. 2020. Unifiedqa: Crossing format boundaries with a single qa system. *arXiv preprint arXiv:2005.00700*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

Tianle Li, Xueguang Ma, Alex Zhuang, Yu Gu, Yu Su, and Wenhu Chen. 2023. Few-shot in-context learning for knowledge base question answering. *arXiv preprint arXiv:2305.01750*.

Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. 2018. Memory augmented policy optimization for program synthesis and semantic parsing. *Advances in Neural Information Processing Systems*, 31.

Shiqi Liang, Kurt Stockinger, Tarcisio Mendes De Farias, Maria Anisimova, and Manuel Gil. 2021. Querying knowledge graphs in natural language. *Journal of big data*, 8(1):3.

Chuang Liu, Junzhuo Li, and Deyi Xiong. 2023a. Tabcqa: A tabular conversational question answering dataset on financial reports. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, pages 196–207.

Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. 2021. Tapex: Table pre-training via learning a neural sql executor. *arXiv preprint arXiv:2107.07653*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, and 1 others. 2023b. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*.

Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2024. A survey of nl2sql with large language models: Where are we, and where are we going? *arXiv preprint arXiv:2408.05109*.

Zhijie Nie, Richong Zhang, Zhongyuan Wang, and Xudong Liu. 2024. Code-style in-context learning for knowledge-based question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 18833–18841.

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

Yiheng Shu, Zhiwei Yu, Yuhan Li, Börje F Karlsson, Tingting Ma, Yuzhong Qu, and Chin-Yew Lin. 2022. Tiara: Multi-grained retrieval for robust question answering over large knowledge bases. *arXiv preprint arXiv:2210.12925*.

Yiqing Song, Wenfa Li, Guiren Dai, and Xinna Shang. 2023. Advancements in complex knowledge graph question answering: a survey. *Electronics*, 12(21):4395.

Xiaoyu Tan, Haoyu Wang, Xihe Qiu, Yuan Cheng, Yinghui Xu, Wei Chu, and Yuan Qi. 2024. Structx: Enhancing large language models reasoning with structured data. *arXiv preprint arXiv:2407.12522*.

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xeron Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, and 1 others. 2025. Tablebench: A comprehensive and complex benchmark for table question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25497–25506.

Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I Wang, and 1 others. 2022. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. *arXiv preprint arXiv:2201.05966*.

Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, and 1 others. 2025. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms. *arXiv preprint arXiv:2502.19411*.

Linyao Yang, Hongyang Chen, Zhao Li, Xiao Ding, and Xindong Wu. 2024a. Give us the facts: Enhancing large language models with knowledge graphs for fact-aware language modeling. *IEEE Transactions on Knowledge and Data Engineering*, 36(7):3091–3110.

Rui Yang, Haoran Liu, Edison Marrese-Taylor, Qingcheng Zeng, Yu He Ke, Wanxin Li, Lechao Cheng, Qingyu Chen, James Caverlee, Yutaka Matsuo, and 1 others. 2024b. Kg-rank: Enhancing large language models for medical qa with knowledge graphs and ranking techniques. *arXiv preprint arXiv:2403.05881*.

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*, pages 174–184.

Wen-tau Yih, Matthew Richardson, Christopher Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 201–206.

Donghan Yu, Sheng Zhang, Patrick Ng, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Yiqun Hu, William Wang, Zhiguo Wang, and Bing Xiang. 2022. Decaf: Joint decoding of answers and logical forms for question answering over knowledge bases. *arXiv preprint arXiv:2210.00063*.

Liangyu Zha, Junlin Zhou, Liyao Li, Rui Wang, Qingyi Huang, Saisai Yang, Jing Yuan, Changbao Su, Xiang Li, Aofeng Su, and 1 others. 2023. Tablegpt: Towards unifying tables, nature language and commands into one gpt. *arXiv preprint arXiv:2307.08674*.

Jinhao Zhang, Lizong Zhang, Bei Hui, and Ling Tian. 2022. Improving complex knowledge base question answering via structural information learning. *Knowledge-Based Systems*, 242:108252.

Wen Zhang, Long Jin, Yushan Zhu, Jiaoyan Chen, Zhiwei Huang, Junjie Wang, Yin Hua, Lei Liang, and Huajun Chen. 2025. Trustuqa: A trustful framework for unified structured data question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25931–25939.

Xiaokang Zhang, Sijia Luo, Bohan Zhang, Zeyao Ma, Jing Zhang, Yang Li, Guanlin Li, Zijun Yao, Kangli Xu, Jinchang Zhou, and 1 others. 2024. Tablellm: Enabling tabular data manipulation by llms in real office usage scenarios. *arXiv preprint arXiv:2403.19318*.

Siyun Zhao, Yuqing Yang, Zilong Wang, Zhiyuan He, Luna K Qiu, and Lili Qiu. 2024. Retrieval augmented generation (rag) and beyond: A comprehensive survey on how to make your llms use external data more wisely. *arXiv preprint arXiv:2409.14924*.

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

Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. 2021. Tat-qa: A question answering benchmark on a hybrid of tabular and textual content in finance. *arXiv preprint arXiv:2105.07624*.

A Datasets and Evaluation Metrics

We evaluate on multiple standard QA datasets spanning different structured data types, **WebQSP** (Yih et al., 2016) is a KGQA dataset with complex questions requiring multi-hop reasoning over Freebase. We use Hit@1 as the evaluation metric. **WikiSQL** (Zhong et al., 2017) is a table QA dataset requiring SQL generation to answer questions over Wikipedia tables. We use Denotation Accuracy (DA)(Jiang et al., 2023a) as the evaluation metric. WIKITABLEQUESTIONS (**WTQ**)(Pasupat and Liang, 2015) is a more challenging table QA dataset with complex questions that often require numerical reasoning and multiple operations. We use Denotation Accuracy (DA) as the evaluation metric. **TableBench** (Wu et al., 2025) is a comprehensive benchmark for table reasoning. We focus on two challenging subtasks, Fact Checking (FC) and Numerical Reasoning (NR), using Exact Match (EM) as the evaluation metric. Statistics of datasets are shows in Table 7.

WikiSQL-E and WTQ-E, To better evaluate the CRAFT module, we constructed two specialized datasets by extracting instances from WikiSQL and WTQ where TrustUQA(Zhang et al., 2025) could not solve them using only predefined functions. These datasets are named **WikiSQL-E** and **WTQ-E**. We primarily use Denotation Accuracy (DA) and F1 metrics to evaluate model performance. Additionally, we introduce two new metrics, **Calling Rate** defined as the percentage of questions requiring functions beyond the predefined function list, and **Calling Denotation Accuracy** (CDA) defined as the accuracy within the subset of questions requiring functions beyond predefined functions. These metrics help validate the effectiveness of the CRAFT module.

Experiments on these constructed datasets effectively explore our framework's improvements and capabilities on complex reasoning problems and provide a valuable dataset and baseline for future research in this area.

B Baselines

We compare CodeUnify with various baseline methods, categorized into data-type specific models and unified models.

Dataset	#Test QA	Others
WTQ	4 344	421 tables
WikiSQL	15 878	5 230 tables
WebQSP	1 639	retrieved version
TableBench-FC	96	96 tables
TableBench-NR	396	396 tables
WikiSQL-E	1189	925 tables
WTQ-E	1801	403 tables

Table 7: Statistics of Experimental Datasets

B.1 Data-Type Specific Models

For the KGQA task on WebQSP (Yih et al., 2016), we compare with several KG specific models: DecAF (Yu et al., 2022) which combines logical form parsing with direct answer generation for KGs, KB-Binder (Li et al., 2023) which incorporates retrieval techniques for improved KGQA, KB-Coder (Nie et al., 2024) which utilizes a code-based paradigm for KG reasoning with in-context learning.

For the TableQA tasks on WikiSQL (Zhong et al., 2017) and WTQ (Pasupat and Liang, 2015), we compare with table-specific models: TAPEX (Liu et al., 2021) which pre-trains language models on tables with SQL execution capabilities, DATER (Ye et al., 2023) which leverages demonstrations to enhance table reasoning in large language models, etc. For experiments on the TableBench dataset, we use baselines directly from the TableBench study (Wu et al., 2025), which include both open-source models (e.g., TableLLM, Qwen2, Llama3.1) and closed-source models (e.g., GPT variants). These baselines implement various prompting strategies such as Program-of-Thought (PoT) and Textual chain-of-thought (TCoT) (Wu et al., 2025) approaches. The baseline also includes TrustUQA (Zhang et al., 2025) methods based on different LLMs for a more complete comparison.

B.2 Unified Models

We also compare with models designed to handle multiple types of structured data: UnifiedSKG (Xie et al., 2022) which integrates multiple structured knowledge formats through a seq2seq framework, StructGPT (Jiang et al., 2023a) which employs an iterative evidence collection and reasoning process across structured data types, Readi (Cheng et al., 2024) which progressively refines reasoning paths for comprehending various structured data formats,

TrustUQA (Zhang et al., 2025) which creates a unified graph representation to generate explainable queries across data types.

C Other Related Works

NL2Query. These methods transform a natural language question into a formal query language that can be executed directly against structured data (Zhang et al., 2025). For single-type approaches, specialized variants such as NL2SQL (Liu et al., 2024) and NL2SPARQL (Jung and Kim, 2020) have been developed, but they usually require custom design for each data type. For unified approaches, frameworks like TrustUQA (Zhang et al., 2025) construct an intermediate conditional graph to bridge disparate data types. NL2Query methods can provide trustworthy, interpretable results via formal verification (Liang et al., 2021) .

NL2Answer. These methods directly generate answers without formal queries, usually using models trained on specific datasets (Zhang et al., 2025). For example, TableGPT (Zha et al., 2023) is a unified framework that enables LLMs to understand and manipulate tables described in natural language. Although NL2Answer methods may generalize across multiple datasets, their reasoning processes often lack transparency and do not easily extend to diverse structured data types (Zhang et al., 2024) .

Retrieval-Augmented Generation. Retrieval-Augmented Generation (RAG) methods retrieve relevant evidence from structured data sources and then generate answers using LLMs (Lewis et al., 2020). While the RAG approach is more flexible across data types (Zhao et al., 2024), RAG approaches are prone to hallucinations and lack the formal validation mechanisms inherent to NL2Query methods (Béchard and Ayala, 2024).

D Prompt Template

D.1 Prompt for CodeSTEP Generation

Figure 3 shows a prompt template for CodeSTEP Generation.

D.2 Prompt for CRAFT module

Figure 4 shows a prompt template for CRAFT module.

"role": "system",
"content": "You are an advanced data analyst proficient in Python, specialized in conditional graph queries for table-based question answering. Your task is to write executable Python code that queries tables to extract relevant information and answer questions based on conditional graph queries.
The conditional graph query function is defined as: cgcodeTest.get_information(args, table_data=table_data, relation=None, head_entity=None, tail_entity=None, key=None, value=None, tail_entity_cmp='=', value_cmp='=', target_type=target_type_ms, is_first=False):This function retrieves information by querying a data source using the given relation and tail entity as search criteria.
Args:
args: The args parameter is fixed as args and cannot be modified.
table_data: The table_data parameter is fixed as table_data and cannot be modified.
relation (str): The relation to the query that matches the tail_entity or contains the head_entity.
tail_entity (str): The tail entity associated with the relation.
head_entity (str): The head entity that belongs to the relation.
key (str): The key to query that matches the tail_entity or head_entity.
value (str): The value associated with or belonging to the key.
tail_entity_cmp (str): Comparison operator ('=', '>', '<', '>=', '<='), default is '='.
value_cmp (str): Comparison operator ('=', '>', '<', '>=', '<='), default is '='.
target_type: Fixed as target_type_ms and cannot be modified.
is_first (bool): Set to True for the first query.
Returns: A set of query results.
Usage Notes:
1) relation + tail_entity: 'relation' is a column name, and 'tail_entity' is a specific value in that column. This mode returns a set of row identifiers where that column matches the value, e.g. {'[line_2]', '[line_7]', '[line_1]'}.2) relation + head_entity: 'relation' is a column name, and 'head_entity' is one or more row identifier(s) in the '[line_id]' format. This mode returns a set of values from the specified column for those rows.
[Note 1]: The first call to the get_information function requires is_first=True.
[Note 2 - Strict Constraint]: In the get_information function, tail_entity and head_entity must never be used together in a single query. Please follow these guidelines:
1. Try to use the functions in the provided preset function list to solve the query at each step.
2. If the preset functions are insufficient, you may use cgcodeTest.LLM_function() to process the query.
3. Use Set and Calculator functions as necessary to complete the task.
4. Record whether cgcodeTest.LLM_function() was used by setting the \"use_LLM_function\" variable to True or False.
[preset function list] Conditional Graph Query functions:
- cgcodeTest.get_information(args, table_data=table_data, relation=None, head_entity=None, tail_entity=None, key=None, value=None, tail_entity_cmp='=', value_cmp='=', target_type=target_type_ms, is_first=False)
- Set functions:
- cgcodeTest.set_union(set1, set2, set3=None, set4=None, set5=None): Get the union of multiple sets. Returns a set.
- cgcodeTest.set_intersection(set1, set2, set3=None, set4=None, set5=None): Get the intersection of multiple sets. Returns a set.
- cgcodeTest.set_difference(set1, set2): Get the difference between two sets. Returns a set.
- cgcodeTest.set_negation(table_data, set1): Get all rows except those in set1. Returns a set.
- Calculator functions:
- cgcodeTest.Min(args=args, set1): Get the smallest element in set1. Returns a set of a numeric value.
- cgcodeTest.Max(args=args, set1): Get the largest element in set1. Returns a set of a numeric value.
- cgcodeTest.Mean(set1): Get the average value of all elements in set1. Returns a set of a numeric value.
- cgcodeTest.Count(set1): Get the number of elements in set1. Returns a set of a numeric value.
- cgcodeTest.Sum(set1): Get the sum of elements in set1. Returns a set of a numeric value.
- If further assistance is needed, use the cgcodeTest.LLM_function() function:
- cgcodeTest.LLM_function(args, table_data=table_data, task, step, mid_outputs_list, expected_name, CODE_file_name=CODE_file_name, question_file_name=question_file_name):
Args: args: Fixed as args without modification.
task: The description of the current task step.
step: The current step number.
mid_outputs_list: The intermediate results prior to the current step.
expected_name: Expected function name with parameters.
CODE_file_name: Fixed as CODE_file_name.
question_file_name: Fixed as question_file_name.
Returns:
mid_result: A set or string of results.
[Guidelines]:
- Think step-by-step and decompose the problem.
- Only use functions from the provided [preset function list] to complete the task.
- Use the provided functions to generate Python code directly.
- Only generate Python code; any additional content must be commented with '#'.
- If cgcodeTest.LLM_function() is used, record it in the 'use_LLM_function' variable.

Figure 3: Prompt template for CodeSTEP Generation.

"role": "system",
"content": '''
Task Context
You are an intelligent code generator that produces step-by-step solutions based on multi-stage task descriptions. Focus exclusively on handling the current step task.
Input Modules
[Final Question]
- represents the final question that needs to be solved in [Complete Code].
- It is for reference. You need to provide the processing code and results required for the current step task in [Current Task].
[Complete Code] (Code Framework)
- Complete code representation, so that you can better understand the overall processing logic.
- You need to implement the code representation of the corresponding cgcodeTest.LLM_function() function in [Complete Code] and get the result.
[Current Task]
- The task description that needs to be processed in the current step.
- You need to write the code based on the task description in [Current Task] with the information and constraints provided.
[Expected Function Name]
- The function name and parameter representation example expected by this task step, for reference.
[Previous Steps and Results]
- The output results of each step before the current step.
- The specific functions and code implementation of each step are shown in [Complete Code].
- [Previous Steps and Results Notes]:
    1. You can get the specific data required for the current step processing from [Previous Steps and Results].
    2. In [Previous Steps and Results], if a step result does not contain data of type '[line_id]', then the original result is directly represented, that is, 'stepx':{{'original result of this step'}}.
    3. In [Previous Steps and Results], if a step result contains data of type '[line_id]', which means the data of a row, then the data information corresponding to each column of the row will be represented accordingly (column name: value), that is, the result of this step will be represented as 'stepx':{{'original result of this step':'specific information corresponding to the original result of this step'}}.

Output Requirements
Code Generation
  1. Must use print() for final result output;
  2. Code must be self-contained (executable independently);
  3. Result format: Single-line text/number.
Result Handling
  1. Return ONLY current step's processed result;
  2. Prohibit intermediate processes/explanations;
  3. Output must match the data type required by the current step (e.g. a set, a string, etc.), but must never be a dictionary!

Processing Rules
  1. Prioritize input data from [Previous Steps and Results]
  2. Ensure output directly contributes to solving [Final Question]
  3. Solve the current task by writing code based on the information and constraints provided
'''

Figure 4: Prompt template for CRAFT.