# CRAFTQA: A Code-Driven Adaptive Framework for Complex Structured Data Reasoning

**Chengtao Gan**[♠], **Zhiqiang Liu**[♠], **Long Jin**[♠], **Yushan Zhu**[♡], **Lei Liang**[♣♢], **Wen Zhang**[♠♢†]
[♠]Zhejiang University [♣]Ant Group
[♡]JIUTIAN Research, Beijing, China
[♢]ZJU-Ant Group Joint Lab of Knowledge Graph
{chengtaogan,zhang.wen}@zju.edu.cn

## Abstract

Real-world scenarios involve massive heterogeneous structured data (e.g., tables, knowledge graphs), making effective reasoning over such diverse data increasingly important. Unified structured data question answering has emerged as a prominent research trend, aiming to answer natural language questions across different structured data types within a single framework. However, existing unified methods share a common limitation: they rely on a set of predefined functions, which restricts their ability to perform complex reasoning beyond these predefined operations. To overcome this fundamental limitation, we propose **CRAFTQA**, a novel adaptive code-driven framework comprising two core modules, CodeSTEP and CRAFT. The **CodeSTEP** module is a paradigm that generates a complete executable Python code sequence, which contains step-by-step code-based reasoning operations based on the question. The **CRAFT** module dynamically generates custom code functions for operations beyond the predefined function set, and seamlessly integrates with CodeSTEP to significantly enhance flexibility in handling complex reasoning. Comprehensive experiments on multiple structured datasets demonstrate that CRAFTQA achieves remarkable improvements in complex reasoning scenarios compared to existing unified methods.

## 1 Introduction

Structured data (e.g., tables, knowledge graphs) organize information in well-defined formats, enabling efficient storage, retrieval, and computation (Tan et al., 2024). Real-world scenarios involve massive amounts of structured data. In the era of Large Language Models (LLMs), effectively **leveraging structured data sources** is crucial for enhancing modern AI systems, particularly in improving factual accuracy, reducing hallucinations, and supporting complex reasoning capabilities (Yang et al., 2024a).

Natural language reasoning over structured data has become increasingly important (Yu et al., 2024). Data-specific methods have been developed for particular data structures like tables or knowledge graphs (e.g., TAPEX (Liu et al., 2022) and DecAF (Yu et al., 2023)). However, real-world scenarios often require reasoning across different types of data sources (Yin et al., 2020). Given the massive heterogeneous structured data in real-world applications, effectively reasoning over such diverse data has become increasingly critical. This has driven significant research interest in **unified structured data question answering methods**, which aim to handle multiple structured data types within a single framework (Zhang et al., 2025). For example, retrieval-based unified methods like StructGPT (Jiang et al., 2023a) and Readi (Cheng et al., 2024) access raw data through predefined functions. To further enhance trustworthiness, TrustUQA (Zhang et al., 2025) obtains answers through a unified query language without inputting extensive raw data into the LLM.

However, existing **unified methods share a common limitation: they rely on a set of predefined functions** (Jiang et al., 2023a; Cheng et al., 2024; Zhang et al., 2025). Complex tasks involving sophisticated computation and advanced logical reasoning often require operations beyond these predefined functions (Gao et al., 2023), which fundamentally **limits their ability to handle complex reasoning beyond predefined operations**.

Methods such as PoT (Chen et al., 2023) and PAL (Gao et al., 2023) have demonstrated that LLMs can better solve complex numerical reasoning tasks by generating Python programs. We believe that decoupling reasoning from computation through code-based programs can effectively enhance LLMs' reasoning capabilities while ensuring transparency and interpretability.

† Corresponding author



*Findings of the Association for Computational Linguistics: ACL 2026*, pages 27860–27876

Inspired by code-based program reasoning (Chen et al., 2023), and to **address the fundamental limitation of existing unified methods**, we propose **CRAFTQA**, a code-driven adaptive framework for unified structured data question answering, comprising two core modules: **C**ode-Based **S**tepwise **T**ransparent **E**xecution **P**aradigm (**CodeSTEP**) and **C**ode-Based **R**easoning for **A**daptive **F**unction **T**ailoring (**CRAFT**).

**CodeSTEP** is a custom code paradigm that generates **complete executable Python code** sequences for structured data reasoning, effectively separating reasoning from computation. This code-based approach aligns better with LLMs' inherent reasoning patterns (Gao et al., 2023; Chen et al., 2023), offering enhanced reasoning capabilities, flexible representations, and transparent processing workflows. Moreover, CodeSTEP's explicit reasoning steps serve as crucial context for the CRAFT module to understand complex scenarios.

**CRAFT** is designed to dynamically handle scenarios beyond predefined functions. We proposed CRAFT to **overcome the fundamental limitation of existing unified structured data QA methods** that can only operate within predefined functions. **CRAFT can generate dedicated code for specific reasoning steps and seamlessly integrate with the main CodeSTEP execution.** This design enhances framework flexibility while maintaining verifiability, enabling the handling of "out-of-predefined" operations that existing methods cannot address, thereby making CRAFTQA particularly suitable for complex reasoning tasks.

We conducted comprehensive experiments on 6 datasets across 3 structured data types: Table (WikiSQL (Zhong et al., 2017) and TableBench (Wu et al., 2025)), Knowledge Graph (WebQSP (Yih et al., 2016)), and Temporal Knowledge Graph (CronQuestions (Saxena et al., 2021)). To validate CRAFT's effectiveness in "out-of-predefined" scenarios, we constructed WikiSQL-E by extracting QA pairs from WikiSQL that potentially require "out-of-predefined" operations. Results show that CRAFTQA achieves **state-of-the-art** performance on **complex reasoning tasks** while maintaining strong standard reasoning capabilities. Furthermore, CRAFTQA demonstrates generalizability across diverse backbone LLM families, where CRAFTQA with smaller open-source backbone LLMs even outperforms advanced baselines with larger closed-source models.

In summary, contributions of this paper are:

- We present **CRAFTQA**, a flexible code-driven framework for unified structured data question answering, comprising **CodeSTEP** for step-by-step code-based reasoning and **CRAFT** for dynamic function customization to handle "out-of-predefined" operations.

- To the best of our knowledge, we are the **first to implement unified adaptive structured data reasoning in a custom code-based form**, enabling complex reasoning tasks that existing methods struggle to address.

- Experiments on 6 datasets across 3 structured data types demonstrate that CRAFTQA significantly **outperforms existing unified methods** in complex reasoning scenarios while maintaining strong standard reasoning performance, and exhibits robust generalizability across diverse backbone LLM families.

## 2 Related Work

### 2.1 Structured Data Question Answering

Structured Data QA plays an increasingly important role in human-computer interaction across healthcare (Yang et al., 2024b; Huang et al., 2021), finance (Liu et al., 2023), and information retrieval (Zhang et al., 2022). 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), KGs (Song et al., 2023), or TKGs (Xue et al., 2024). Recent advancements include ReasoningLM (Jiang et al., 2023b), which utilizes a code-generation-based paradigm to improve multi-hop reasoning. In contrast, **unified** methods aim to support reasoning across multiple structured data types simultaneously (Khashabi et al., 2020), such as StructGPT (Jiang et al., 2023a), which handles KGs, tables, and databases.

### 2.2 LLM-Based Unified Frameworks

With the rapid advancement of large language models, several unified frameworks have been proposed for structured data question answering. StructGPT (Jiang et al., 2023a) is an iterative reading-then-reasoning framework that generates answers based on collected evidence. Readi (Cheng et al., 2024) is a reasoning-path-editing framework that collects KG evidence through edited reasoning paths. TrustUQA (Zhang et al., 2025) employs Condition

Graph and a two-layer query approach to uniformly support tables, KGs, and TKGs.

### 2.3 Code-Based Reasoning

Recent research has demonstrated that code-based methods can enhance reasoning capabilities in LLMs (Yang et al., 2025). Program of Thought (PoT) (Chen et al., 2023) and PAL (Gao et al., 2023) show that executable code can decompose complex problems into manageable computational steps, facilitating more interpretable and controllable reasoning processes.

## 3 Methodology

### 3.1 Overview

CRAFTQA is a unified code-driven framework for structured data question answering. As illustrated in Figure 1, it comprises two synergistic modules: **CodeSTEP** (**Code**-based **S**tepwise **T**ransparent **E**xecution **P**aradigm) for stepwise code-based reasoning, and **CRAFT** (**C**ode-based **R**easoning for **A**daptive **F**unction **T**ailoring) for dynamic custom function generation.

Given a data source $\mathcal{D}$ and question $\mathcal{Q}$, we transform $\mathcal{D}$ into a data schema $\mathcal{D}_{schema}$ containing structural metadata, and a Condition Graph $\mathcal{D}_{cg}$ (Zhang et al., 2025) for unified querying. An LLM $\mathcal{M}_\theta$ with few-shot prompt $\mathcal{P}$ generates executable code $\mathcal{C} = \{c_i\}_{i=1}^n$, executed on $\mathcal{D}_{cg}$ to yield the answer $\mathcal{A}$:

$$\mathcal{M}_\theta(\mathcal{D}_{schema}, \mathcal{Q}, \mathcal{P}) \rightarrow \mathcal{C}, \tag{1}$$

$$\textsc{Exec}(\mathcal{C}, \mathcal{D}_{cg}) \rightarrow \mathcal{A}. \tag{2}$$

### 3.2 CodeSTEP Module

**Stepwise Code Generation.** CodeSTEP decomposes code generation into two phases: **(1) Query Analysis.** Given question $\mathcal{Q}$ and data schema $\mathcal{D}_{schema}$, $\mathcal{M}_\theta$ constructs a reasoning path $\mathcal{S} = \{s_i\}_{i=1}^n$, where each $s_i$ is a natural language reasoning step. **(2) Stepwise Code Construction.** For each $s_i$, $\mathcal{M}_\theta$ generates a corresponding code $c_i$, constructing the complete executable sequence $\mathcal{C} = \{c_i\}_{i=1}^n$, as formulated in Equation 1.

**Condition Graph Query Operation.** The primary data retrieval operation is the query function $get$, which retrieves target entities from $\mathcal{D}_{cg}$:

$$get : (\mathcal{D}_{cg}, R, E_h, E_t, \delta_t, K, V, \delta_v) \rightarrow O, \tag{3}$$

where $R$ denotes the relation (column header for tables, edge type for KGs), $E_h$ and $E_t$ are head and tail entity sets, $\delta_t \in \{=, >, <, \geq, \leq\}$ is the comparison operator for $E_t$, and $O$ is the output entity set. $K$ specifies a conditional attribute (e.g., temporal property), $V$ provides the threshold value of $K$, and $\delta_v$ is the comparison operator of $V$.

**Example:**
$get(E_h$=`None`, $R$=`'Won'`, $E_t$=`'Nobel Prize'`, $K$=`'Year'`, $V$=2000, $\delta_v$=`'>'`),
retrieves all 'Nobel Prize' winners where the `'Year'` attribute exceeds 2000.

**Semantic Entity Alignment.** Entities in generated code may not exactly match those in $\mathcal{D}_{cg}$ due to lexical variations. We address this via Sentence-BERT (Reimers and Gurevych, 2019) alignment. For an entity $e$ in the code and candidates $\mathcal{E} = \{e_j\}_{j=1}^m$ in $\mathcal{D}_{cg}$:

$$e^* = \arg\max_{e_j \in \mathcal{E}} \cos\big(\phi(e), \phi(e_j)\big), \tag{4}$$

where $\phi(\cdot)$ denotes Sentence-BERT embedding and $\cos(\cdot,\cdot)$ computes cosine similarity. This alignment is performed automatically within $get$, ensuring accurate entity matching without explicit handling in generated code.

**Predefined Function Set.** Beyond $get$, CodeSTEP provides set and algebraic operations, as detailed in Table 1. These collectively form the predefined function set:

$$\mathcal{F}_{pred} = \{get\} \cup \mathcal{F}_{set} \cup \mathcal{F}_{cal}, \tag{5}$$

where $\mathcal{F}_{set} = \{f_\cup, f_\cap, f_-\}$ and $\mathcal{F}_{cal} = \{f_{\min}, f_{\max}, f_{avg}, f_{cnt}, f_{sum}\}$.

During the reasoning process, each code step $c_i$ may utilize functions from $\mathcal{F}_{pred}$ for its implementation. Let $W_i = \{w_k\}_{k=1}^{i-1}$ denote the set of intermediate results from all preceding steps. When a predefined function is invoked:

$$c_i = f_i(\mathcal{D}_{cg}, \tilde{W}_i), \quad f_i \in \mathcal{F}_{pred}, \tag{6}$$

where $\tilde{W}_i \subseteq W_i$ contains selected results serving as input to the current step.

However, $\mathcal{F}_{pred}$ is inherently limited in scope, unable to handle "out-of-predefined" operations required by complex questions. This limitation motivates a flexible approach that can dynamically generate custom functions tailored to specific operational requirements.

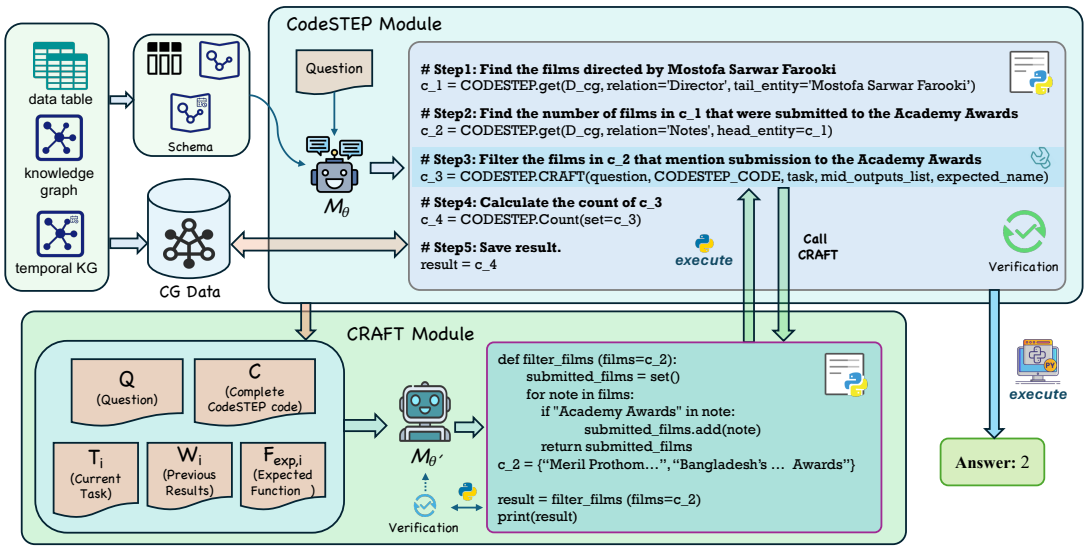


Figure 1: Overview of CRAFTQA framework. The framework comprises two synergistic modules: CodeSTEP for stepwise code-based reasoning and CRAFT for dynamic custom function generation.

| Operation | Definition |
|---|---|
| *Set Operations* | |
| Union | $f_\cup(S_1,\ldots,S_n) \mapsto \bigcup_{i=1}^n S_i$ |
| Intersection | $f_\cap(S_1,\ldots,S_n) \mapsto \bigcap_{i=1}^n S_i$ |
| Difference | $f_-(S_1,S_2) \mapsto S_1 \setminus S_2$ |
| *Algebraic Operations* | |
| Min / Max | $f_{\min}(S) \mapsto \min(S);\ f_{\max}(S) \mapsto \max(S)$ |
| Mean | $f_{avg}(S) \mapsto \frac{1}{\lvert S\rvert}\sum_{x\in S} x$ |
| Count / Sum | $f_{cnt}(S) \mapsto \lvert S\rvert;\ f_{sum}(S) \mapsto \sum_{x\in S} x$ |

Table 1: Predefined Calculation Operations

### 3.3 CRAFT Module

CRAFT extends CodeSTEP by dynamically generating custom functions for operations beyond $\mathcal{F}_{pred}$.

**CRAFT Interface Function.** To enable seamless invocation of CRAFT within CodeSTEP, we define an interface function $f_{craft}$ that serves as the bridge between the two modules:

$$f_{craft} : (\mathcal{T}_i, W_i, F_{exp,i}) \to w_i, \tag{7}$$

where $\mathcal{T}_i$ is the task description for the current step, $W_i$ contains intermediate results from preceding steps, $F_{exp,i}$ is the expected function signature, and $w_i$ is the computed result. This interface function encapsulates the entire CRAFT execution process, allowing CodeSTEP to invoke the CRAFT module to perform complex custom operations through a unified calling convention.

With this interface, we define the extended function set available to CodeSTEP:

$$\mathcal{F} = \mathcal{F}_{pred} \cup \{f_{craft}\}. \tag{8}$$

Each code step $c_i$ in the generated sequence $\mathcal{C}$ is now expressed as:

$$c_i = \begin{cases} f_i(\mathcal{D}_{cg}, \tilde{W}_i), & f_i \in \mathcal{F}_{pred}, \\ f_{craft}(\mathcal{T}_i, W_i, F_{exp,i}), & \text{otherwise.} \end{cases} \tag{9}$$

This formulation enables CodeSTEP to maintain a consistent format for stepwise code reasoning, where each $c_i$ uniformly represents a function call within the extended set $\mathcal{F}$.

**Custom Function Generation.** When $f_{craft}$ is invoked during execution, it triggers the CRAFT module to generate and execute a custom function. Internally, CRAFT utilizes an LLM $\mathcal{M}_{\theta'}$ with specialized prompt $\mathcal{P}_c$ to synthesize a self-contained custom function $\hat{f}_i$:

$$\mathcal{M}_{\theta'}(\mathcal{Q}, \mathcal{C}, \mathcal{T}_i, W_i, F_{exp,i}, \mathcal{P}_c) \to \hat{f}_i. \tag{10}$$

The output of $f_{craft}$ is obtained by executing the generated function $\hat{f}_i$ from Equation 10:

$$f_{craft}(\mathcal{T}_i, W_i, F_{exp,i}) := \textsc{Exec}(\hat{f}_i). \tag{11}$$

Note that $\mathcal{Q}$, $\mathcal{C}$, and $\mathcal{P}_c$ are accessible as global context during execution, while $\mathcal{T}_i$, $W_i$, and $F_{exp,i}$ are step-specific parameters passed through the interface.

The input to CRAFT comprises five carefully designed components that convey $\mathcal{M}_\theta$'s understanding of the current step to $\mathcal{M}_{\theta'}$, enabling context-aware code generation: **(1) the original question** $\mathcal{Q}$, which provides the ultimate answering objective, enabling $\mathcal{M}_{\theta'}$ to consider whether the current step contributes to the final answer; **(2) the**

**complete code sequence** $\mathcal{C}$ generated by $\mathcal{M}_\theta$, representing the overall reasoning chain of thought for the task, which helps $\mathcal{M}_{\theta'}$ understand the role of the current step within the holistic reasoning process; **(3) the current task description** $\mathcal{T}_i$, articulated by $\mathcal{M}_\theta$ based on its understanding of the current step's requirements, thereby conveying the specific functional needs to $\mathcal{M}_{\theta'}$; **(4) the previous results** $W_i$, providing data formats and intermediate outputs from preceding steps that interact with $\mathcal{D}_{cg}$, enabling $\mathcal{M}_{\theta'}$ to generate functions with accurate input/output handling; **(5) the expected function signature** $F_{exp,i}$, a reference function name proposed by $\mathcal{M}_\theta$ that encapsulates its understanding of the required operation, helping bridge the comprehension gap between the two models regarding the current scenario.

**Seamless Integration with CodeSTEP.** The integration between CRAFT and CodeSTEP follows a well-defined workflow:

**(1) Code Generation with Delegation.** During code generation (Equation 1), when $\mathcal{M}_\theta$ encounters operations outside $\mathcal{F}_{pred}$, it generates $f_{craft}$ calls with appropriate parameters $(\mathcal{T}_i, W_i, F_{exp,i})$, as formulated in Equation 9.

**(2) Deferred Execution.** The $f_{craft}$ calls remain as placeholders in $\mathcal{C}$ until execution time. This deferred execution strategy ensures that CRAFT has access to actual intermediate results $W_i$ computed from preceding steps.

**(3) Dynamic Function Synthesis.** Upon reaching an $f_{craft}$ call during execution, CRAFT synthesizes the custom function $\hat{f}_i$ via Equation 10. By leveraging $\mathcal{Q}$, $\mathcal{C}$, $\mathcal{T}_i$, $W_i$, and $F_{exp,i}$, CRAFT fully comprehends the context of the current step, enabling it to generate code that optimally addresses the step-specific problem.

**(4) Execution and Continuation.** The generated function $\hat{f}_i$ is executed, and its result $w_i$ is returned through $f_{craft}$ to the main CodeSTEP execution flow. Subsequent steps can then utilize $w_i$ as part of their input $W_{i+1}$, maintaining the sequential reasoning chain.

### 3.4 Code Verification and Execution

**Executability Verification.** To ensure the generated code is syntactically correct, we verify each code block $\mathcal{B}$ (either $\mathcal{C}$ from CodeSTEP or $\hat{f}$ from CRAFT) through a Python interpreter:

$$\textsc{Verify}(\mathcal{B}) = \begin{cases} 1, & \text{if } \mathcal{B} \text{ is valid,} \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

When verification fails, we regenerate $\mathcal{B}$ with identical input up to $T$ attempts. The verified code $\mathcal{B}^*$ is obtained at the first successful attempt, or defaults to $\mathcal{B}^{(T)}$ if all attempts fail.

**Complete Execution Process.** The verified code $\mathcal{C}^* = \{c_i\}_{i=1}^n$ is executed as a complete unit in a single run, where each code step $c_i$ is processed sequentially within the execution. Whether invoking a predefined function or $f_{craft}$, each step produces an intermediate result:

$$w_i = \textsc{Exec}(c_i, \mathcal{D}_{cg}, W_i). \tag{13}$$

For steps invoking $f_{craft}$, the execution internally triggers CRAFT to generate $\hat{f}_i$ (Equation 10), verifies its executability (Equation 12), and returns the result through the interface (Equation 11).

These intermediate results form a sequential reasoning chain, where each $w_i$ is accumulated into $W_{i+1} = W_i \cup \{w_i\}$ for subsequent steps. The final result $w_n$ directly corresponds to the answer $\mathcal{A}$:

$$\textsc{Exec}(\mathcal{C}^*, \mathcal{D}_{cg}) \rightarrow w_n = \mathcal{A}. \tag{14}$$

## 4 Experiments

We conduct comprehensive experiments to answer the following four key research questions: **RQ1**: How effective is CRAFTQA in complex structured data reasoning tasks, particularly in "out-of-predefined" scenarios? **RQ2**: While achieving improvements in complex reasoning, does CRAFTQA maintain competitive performance on standard reasoning tasks across different types of structured data? **RQ3**: How generalizable is CRAFTQA when applied to diverse backbone LLMs with varying capabilities and scales? **RQ4**: How does each component in the CRAFTQA framework contribute to the overall performance?

### 4.1 Experimental Setup

**Datasets and Evaluation Metrics.** For **RQ1**, we use two challenging sub-datasets from TableBench (Wu et al., 2025): Fact Checking (FC) and Numerical Reasoning (NR), along with WikiSQL-E, a dataset we constructed by extracting question-answer pairs from WikiSQL (Zhong et al., 2017) that may involve "out-of-predefined" operations

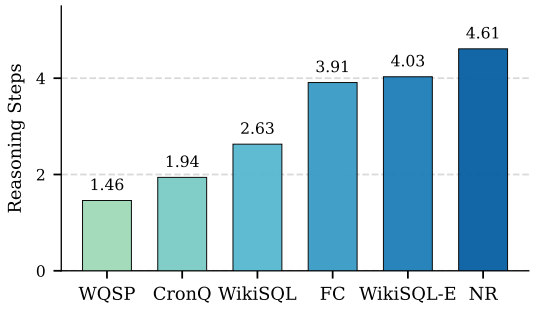


Figure 2: **Average Reasoning Steps** across Datasets.

(details in Appendix A.1). We use Denotation Accuracy (DA) (Jiang et al., 2023a) as the primary metric, and compute an Overall metric as the weighted average of DA:

$$\text{Overall} = \frac{\sum_{i=1}^{K} n_i \cdot DA_i}{\sum_{i=1}^{K} n_i}, \tag{15}$$

where $K$ is the number of datasets, $n_i$ and $DA_i$ denote the sample size and DA of the $i$-th dataset.

To assess the correctness of answers requiring "out-of-predefined" functions, we propose Calling Denotation Accuracy (**CDA**):

$$\text{CDA} = \frac{N_{calling\_correct}}{N_{calling}}, \tag{16}$$

where $N_{calling}$ is the number of questions calling "out-of-predefined" functions, and $N_{calling\_correct}$ denotes the number that are answered correctly. Detailed definitions are in Appendix A.2.

For **RQ2**, we use WebQSP (Yih et al., 2016) (KGQA), WikiSQL (TableQA), and CronQuestions (Saxena et al., 2021) (TKGQA), with Hit@1 for WebQSP and CronQuestions, DA for WikiSQL.

To quantify the reasoning complexity of datasets, inspired by TableBench (Wu et al., 2025), we adopt Average Reasoning Steps as a metric. As shown in Figure 2, **RQ1** datasets (WikiSQL-E, TableBench-FC, and TableBench-NR) exhibit higher average reasoning steps, indicating greater complexity. In contrast, **RQ2** datasets (WebQSP, CronQuestions, and WikiSQL) have lower average reasoning steps, representing standard reasoning tasks. Detailed statistics are provided in Appendix A.3.

For **RQ3**, we evaluate on TableBench (FC and NR) using CDA, DA, and F1. More dataset statistics are in Appendix A.

**Baselines.** For RQ1 and RQ2, we compare CRAFTQA against advanced unified structured data reasoning methods: PoT (Chen et al., 2023), which generates executable programs for reasoning; StructGPT (Jiang et al., 2023a), which employs iterative reading-then-reasoning; Readi (Cheng et al., 2024), which edits reasoning paths via environmental feedback; and TrustUQA (Zhang et al., 2025), which uses unified Condition Graph for querying. For the "Out-of-Predefined" scenarios experiments (RQ1), we adopt TrustUQA as the primary baseline, as it represents the **state-of-the-art** among published methods.

| Methods | TableBench | | WikiSQL-E | Overall |
|---|---|---|---|---|
| | FC | NR | DA(%) | Avg. |
| **PoT** (Chen et al., 2023) | | | | |
| – GPT-3.5-Turbo | 54.2 | 36.4 | 25.0 | 29.4 |
| – GPT-4o-mini | 41.7 | 36.9 | 25.0 | 28.8 |
| – GPT-4o | 51.0 | 43.7 | 27.4 | 32.6 |
| **StructGPT** (Jiang et al., 2023a) | | | | |
| – GPT-3.5-Turbo | 56.3 | 23.0 | 44.0 | 39.7 |
| – GPT-4o-mini | 58.3 | 24.2 | 38.8 | 36.5 |
| – GPT-4o | 63.5 | 42.2 | 43.5 | 44.3 |
| **Readi** (Cheng et al., 2024) | | | | |
| – GPT-3.5-Turbo | 51.0 | 33.6 | 45.1 | 42.7 |
| – GPT-4o-mini | 55.2 | 38.4 | 37.5 | 38.7 |
| – GPT-4o | 62.5 | 49.5 | 51.3 | 51.5 |
| **TrustUQA** (Zhang et al., 2025) | | | | |
| – GPT-3.5-Turbo | 50.0 | 20.2 | **70.0** | 57.1 |
| – GPT-4o-mini | 55.2 | 21.7 | 74.5 | 61.0 |
| – GPT-4o | 62.5 | 29.6 | 80.1 | 67.2 |
| **CRAFTQA(Ours)** | | | | |
| *Open-source Models* | | | | |
| – Qwen2.5-7B | 57.9 | 35.4 | 65.9 | 58.3 |
| – LLaMA-3.1-8B | 57.3 | 31.1 | 76.0 | 64.4 |
| *Closed-source Models* | | | | |
| – GPT-3.5-Turbo | **61.5** | **38.6** | 67.8 | **60.6** |
| – GPT-4o-mini | **64.6** | **40.7** | **79.0** | **69.2** |
| – GPT-4o | **68.8** | **51.3** | **85.6** | **76.6** |

Table 2: **Complex reasoning** performance comparison.

**Implementation Details.** Our experiments employ various LLMs as inference engines, including GPT (Achiam et al., 2023), LLaMA (Dubey et al., 2024), DeepSeek (Liu et al., 2024a), Gemini (Team et al., 2023), and Qwen (Bai et al., 2023) series. We adopt a self-consistency strategy with 5 samples, set the maximum retry attempts $T$=3, and use Sentence-BERT (Reimers and Gurevych, 2019) to semantically align entities. Further details are provided in Appendix B.

### 4.2 CRAFTQA for Complex Reasoning (RQ1)

To answer **RQ1**, we evaluate CRAFTQA on three complex reasoning datasets: TableBench-FC, TableBench-NR, and WikiSQL-E, which demand significantly more reasoning steps (Figure 2).

Table 2 compares CRAFTQA with advanced uni-

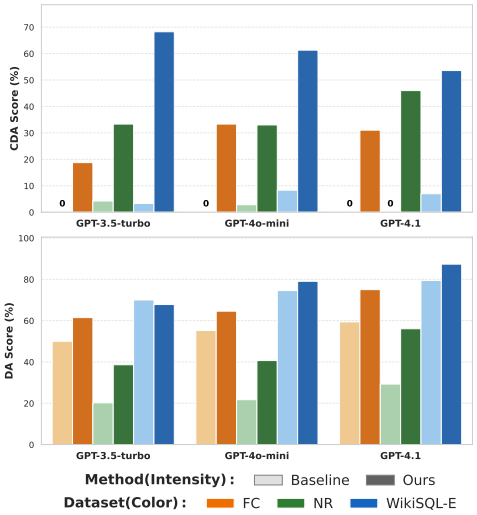


Figure 3: **Calling Denotation Accuracy (CDA) and Denotation Accuracy (DA) comparison between CRAFTQA and the state-of-the-art baseline.** Detailed results are in Table 7.

fied structured data reasoning methods, including PoT (Chen et al., 2023), StructGPT (Jiang et al., 2023a), Readi (Cheng et al., 2024), and TrustUQA (Zhang et al., 2025). Under the same backbone LLM, CRAFTQA consistently achieves the best performance across nearly all datasets. With GPT-4o, CRAFTQA attains 68.8%, 51.3%, and 85.6% DA on FC, NR, and WikiSQL-E respectively, significantly outperforming all baselines. Similar advantages are observed with GPT-4o-mini. The only exception occurs with GPT-3.5-Turbo on WikiSQL-E, where TrustUQA achieves 70.0% compared to our 67.8%. This is expected since code-based methods inherently require sufficient backbone LLM capabilities (Chen et al., 2023), and GPT-3.5-Turbo's limited reasoning ability affects the performance. Nevertheless, CRAFTQA still achieves the best Overall metric (60.6%) with GPT-3.5-Turbo.

We further explore CRAFTQA with smaller open-source models. Notably, CRAFTQA-LLaMA-3.1-8B (64.4%) and CRAFTQA-Qwen2.5-7B (58.3%) even outperform baselines using larger closed-source models, such as Readi-GPT-4o (51.5%), StructGPT-GPT-4o (44.3%), and PoT-GPT-4o (32.6%), demonstrating CRAFTQA's superiority in complex reasoning scenarios.

### 4.3 CRAFTQA for "Out-of-Predefined" Scenarios (RQ1)

To investigate CRAFTQA's effectiveness in handling "out-of-predefined" scenarios, we compare it against TrustUQA (Zhang et al., 2025), the current published state-of-the-art method on unified structured data QA. We evaluate on WikiSQL-E and TableBench (FC and NR) using GPT-3.5-Turbo, GPT-4o-mini, and GPT-4.1 as backbone LLMs, with results shown in Figure 3.

To quantify performance on "out-of-predefined" scenarios, we track questions that cannot be solved by predefined functions and calculate the Calling Denotation Accuracy (CDA) for these cases. As shown in Figure 3, **CRAFTQA consistently achieves substantially higher CDA across all datasets and backbone LLMs**. For instance, on WikiSQL-E with GPT-4.1, CRAFTQA achieves a CDA of 53.57% compared to TrustUQA's 6.98%, demonstrating CRAFT's advantage in handling tasks requiring "out-of-predefined" functions.

Furthermore, improvements in "out-of-predefined" scenarios contribute to substantial overall performance gains. CRAFTQA **significantly outperforms** TrustUQA in both DA and F1 metrics across nearly all settings. For example, on the Numerical Reasoning dataset with GPT-4.1, CRAFTQA achieves a DA of 56.06% versus TrustUQA's 29.29%, an improvement of 26.77 percentage points. These results demonstrate our framework's superiority in complex reasoning tasks, where effective handling of "out-of-predefined" scenarios plays a crucial role.

### 4.4 CRAFTQA for Standard Reasoning(RQ2)

Having demonstrated CRAFTQA's superiority in complex "out-of-predefined" scenarios (RQ1), we now investigate whether CRAFTQA maintains competitive performance on standard reasoning tasks across heterogeneous structured data sources. We compare CRAFTQA against state-of-the-art unified structured data QA methods on three representative datasets: WebQSP (Knowledge Graph), WikiSQL (Table), and CronQuestions (Temporal Knowledge Graph), as shown in Table 3.

As shown in Table 3, CRAFTQA achieves the best performance on both WebQSP (85.20% Hit@1) and WikiSQL (86.10% DA), outperforming TrustUQA (Zhang et al., 2025) by 1.70 and 0.40 percentage points, respectively. On CronQuestions, CRAFTQA achieves 97.10% accuracy, comparable to TrustUQA's 97.20%. This marginal difference can be attributed to the inherent simplicity of TKG-based tasks: as illustrated in Figure 2, CronQuestions requires fewer reasoning steps and TKG data involves limited operation variety, resulting in rare "out-of-predefined" scenarios that leave limited room for CRAFT to demonstrate its advan-

| Methods | WebQSP | WikiSQL | CronQ |
|---|---|---|---|
| | Hit@1 | DA(%) | Hit@1 |
| PoT (Chen et al., 2023) | 14.70 | 44.42 | 57.63 |
| StructGPT (Jiang et al., 2023a) | 69.60 | 57.90 | – |
| Readi (Cheng et al., 2024) | 74.30 | 64.70 | – |
| TrustUQA (Zhang et al., 2025) | 83.50 | 85.70 | **97.20** |
| **CRAFTQA (Ours)** | **85.20** | **86.10** | 97.10 |

Table 3: **Standard reasoning performance across heterogeneous structured data sources.**

tage. Nevertheless, CRAFTQA still achieves near-optimal performance, demonstrating its robustness across varying task complexities.

These results demonstrate that **CRAFTQA maintains comparable or even superior standard reasoning capabilities while significantly enhancing complex reasoning performance**, and is applicable to diverse structured data sources including KG, Table, and TKG.

### 4.5 Generalization of CRAFTQA (RQ3)

To evaluate CRAFTQA's generalization, we assess its performance across different backbone LLMs. We experiment on TableBench's Fact Checking and Numerical Reasoning datasets using 4 open-source models (LLaMA-3.1-8B, Qwen2.5-7B, Qwen3-32B, DeepSeek-V3) and 8 closed-source models (Qwen-Max, Gemini-2.5-Flash/Pro, GPT-3.5, GPT-4o-mini, GPT-4o, GPT-5-mini, o4-mini). Detailed results are provided in Appendix Table 9.

Figure 4 illustrates the relationship between CDA (line) and overall metrics DA/F1 (bars) across backbone LLMs, revealing two key observations:

First, CRAFTQA's performance scales with the backbone LLM's capability. On Fact Checking, DA improves from 61.5% (GPT-3.5) to 68.8% (GPT-4o), and further to 83.3% (o4-mini), suggesting that **CRAFTQA is well-positioned to benefit from future LLM advancements**.

Second, within the same LLM family, CDA trends exhibit strong consistency with DA and F1, validating that overall performance gains directly stem from CRAFT's capability in handling "out-of-predefined" scenarios.

Notably, as shown in Figure 4, the dashed line represents TrustUQA's DA with GPT-4o. Combined with Table 2, we observe that CRAFTQA enables small-parameter open-source models to achieve competitive or superior performance compared to advanced baselines using large closed-source models (GPT-4o). These results confirm CRAFTQA's strong generalization and scalability across diverse backbone LLMs.

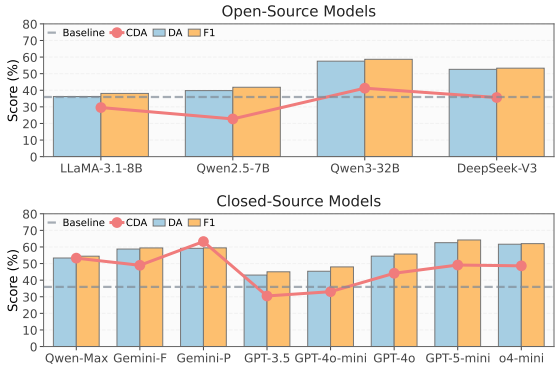


Figure 4: **Generalization of CRAFTQA** across backbone LLMs.

| Methods | FC | | NR | | WikiSQL-E | |
|---|---|---|---|---|---|---|
| | DA | F1 | DA | F1 | DA | F1 |
| **CRAFTQA** | **68.8** | **71.6** | **51.3** | **51.9** | **87.3** | **87.6** |
| w/o CRAFT | 65.3 | 67.7 | 45.9 | 46.3 | 84.5 | 84.6 |
| w/o CRAFT&CodeSTEP | 59.4 | 63.2 | 18.4 | 19.7 | 79.8 | 80.0 |

Table 4: **Ablation Study results** across Fact Checking (FC), Numerical Reasoning (NR), and WikiSQL-E.

### 4.6 Ablation Study (RQ4)

To validate the contribution of each module in CRAFTQA, we conduct ablation experiments on TableBench's Fact Checking and Numerical Reasoning datasets using GPT-4o, and on WikiSQL-E using GPT-4.1, with results presented in Table 4.

Removing either the CRAFT or CodeSTEP module significantly degrades performance. For instance, DA on Numerical Reasoning drops from 51.3% (CRAFTQA) to 45.9% (w/o CRAFT) and further to 18.4% (w/o CRAFT&CodeSTEP). These results validate that both modules are crucial: CodeSTEP provides core reasoning structure while CRAFT offers flexibility for handling "out-of-predefined" scenarios, and their synergy further enhances performance on complex reasoning tasks.

## 5 Conclusion

In this paper, we present CRAFTQA, an efficient and flexible code-driven framework for unified structured data question answering, comprising two core modules: CodeSTEP and CRAFT. CodeSTEP generates and executes code sequences to answer natural language questions directly, and works seamlessly with the CRAFT module, which can

effectively handles "out-of-predefined" scenarios. Experiments on diverse datasets demonstrate our framework's effectiveness, particularly on complex reasoning tasks, and validate its generalization capabilities. As LLMs continue to evolve, CRAFTQA is well-positioned to benefit from future LLMs advancements. Looking forward, we plan to extend our framework to more data forms, and further enhance its capabilities.

## Limitations

While CRAFTQA demonstrates significant improvements in complex reasoning scenarios, we acknowledge several limitations. First, the performance gains on simpler standard reasoning tasks are modest, these tasks typically involve fewer "out-of-predefined" operations, which limits the opportunities for the CRAFT module to fully leverage its dynamic reasoning capabilities. Second, as a code-based framework, CRAFTQA may present challenges for LLMs with limited code generation abilities, potentially restricting its applicability to models with weaker programming proficiency. These limitations present opportunities for future research to extend the framework's applicability across broader reasoning scenarios.

## Acknowledgments

This work is founded by National Natural Science Foundation of China (NSFC62306276/NSFCU23B2055), New Generation Artificial Intelligence-National Science and Technology Major Project 2030 (2025ZD0122800), Yongjiang Talent Introduction Programme (2022A-238-G), and Fundamental Research Funds for the Central Universities (226-2023-00138). This work was supported by Ant Group.

## References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, and 1 others. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Trans. Mach. Learn. Res.*, 2023.

Sitao Cheng, Ziyuan Zhuang, Yong Xu, Fangkai Yang, Chaoyun Zhang, Xiaoting Qin, Xiang Huang, Ling Chen, Qingwei Lin, Dongmei Zhang, Saravan Rajmohan, and Qi Zhang. 2024. Call me when necessary: LLMs can efficiently and faithfully reason over structured environments. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4275–4295, Bangkok, Thailand. Association for Computational Linguistics.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

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, Xin Zhao, and Ji-Rong Wen. 2023a. StructGPT: A general framework for large language model to reason over structured data. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9237–9251, Singapore. Association for Computational Linguistics.

Jinhao Jiang, Kun Zhou, 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. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3721–3735, Singapore. Association for Computational Linguistics.

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. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1896–1907, Online. Association for Computational Linguistics.

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

Chuang Liu, Junzhuo Li, and Deyi Xiong. 2023. 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. 2022. TAPEX: table pre-training via learning a neural SQL executor. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

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

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Apoorv Saxena, Soumen Chakrabarti, and Partha Talukdar. 2021. Question answering over temporal knowledge graphs. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 6663–6676, Online. Association for Computational Linguistics.

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. Struct-x: Enhancing large language models reasoning with structured data. *arXiv preprint arXiv:2407.12522*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, and 1 others. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

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.

Chao Xue, Di Liang, Pengfei Wang, and Jing Zhang. 2024. Question calibration and multi-hop modeling for temporal question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19332–19340.

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, Yuhe Ke, Wanxin Li, Lechao Cheng, Qingyu Chen, James Caverlee, Yutaka Matsuo, and Irene Li. 2024b. KG-rank: Enhancing large language models for medical QA with knowledge graphs and ranking techniques. In *Proceedings of the 23rd Workshop on Biomedical Natural Language Processing*, pages 155–166, Bangkok, Thailand. Association for Computational Linguistics.

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.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for joint understanding of textual and tabular data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8413–8426, Online. Association for Computational Linguistics.

Donghan Yu, Sheng Zhang, Patrick Ng, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Yiqun Hu, William Yang Wang, Zhiguo Wang, and Bing Xiang. 2023. Decaf: Joint decoding of answers and logical forms for question answering over knowledge bases. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Fei Yu, Hongbo Zhang, Prayag Tiwari, and Benyou Wang. 2024. Natural language reasoning, a survey. *ACM Computing Surveys*, 56(12):1–39.

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.

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

# 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 requiring reasoning over Freebase. We use Hit@1 as the evaluation metric. **CronQuestions** (Saxena et al., 2021) is a temporal knowledge graph question answering (TKGQA) dataset designed to evaluate models' ability to reason over time-dependent facts. Each question is associated with entities, relations, and timestamps, requiring temporal reasoning across evolving knowledge graphs. We use Hit@1 as the evaluation metric. **WikiSQL** (Zhong et al., 2017) is a table QA dataset requiring to answer questions over Wikipedia tables. We use Denotation Accuracy (DA)(Jiang et al., 2023a) as the evaluation metric. **TableBench** (Wu et al., 2025) is a comprehensive and complex benchmark for table reasoning. We focus on two challenging subtasks, Fact Checking (FC) and Numerical Reasoning (NR), using Denotation Accuracy (DA) as the evaluation metric. Statistics of datasets are shown in Table 5.

| Dataset | # QA Pairs | Struct. Data Vol. | Type |
|---|---|---|---|
| WebQSP | 1,639 | Retrieved Ver. | KG |
| CronQuestions | 30,000 | 5,000 triples | TKG |
| WikiSQL | 15,878 | 5,230 tables | Table |
| TableBench-FC | 96 | 96 tables | Table |
| TableBench-NR | 396 | 396 tables | Table |
| WikiSQL-E | 1,190 | 925 tables | Table |

Table 5: **Statistics of datasets used in evaluation.** We report the number of Question-Answer (QA) pairs, the volume of structured data, and the target data type (KG: Knowledge Graph, TKG: Temporal KG).

## A.1 WikiSQL-E Construction

To better evaluate the performance of CRAFTQA in complex reasoning and "out-of-predefined" scenarios, we constructed a specialized dataset by extracting questions from WikiSQL that could not be well solved using only predefined functions, and used the extracted questions to construct a dataset named **WikiSQL-E**.

The construction of WikiSQL-E follows a systematic 4-step automated process, designed to identify questions that genuinely require operations beyond predefined function sets:

**Step 1: Compile a Unified Predefined Function List.** We systematically reviewed the predefined operations of existing state-of-the-art unified methods, including StructGPT (Jiang et al., 2023a), Readi (Cheng et al., 2024), and TrustUQA (Zhang et al., 2025), and consolidated a comprehensive predefined function list that covers virtually all "predefined" executable operations shared across these methods. The compiled function list includes: (1) "`get_information`": retrieves information by querying a data source using specified relations and entities; (2) set operations: "`set_union`", "`set_intersection`", and "`set_difference`"; and (3) algebraic operations: "`Min`", "`Max`", "`Mean`", "`Count`", and "`Sum`".

**Step 2: Add a Placeholder Interface.** We additionally introduced an "`out_of_predefined()`" placeholder function, which does not perform actual data processing at this stage, to simulate a callable interface for operations beyond the predefined function list. This enables the screening model to explicitly signal when predefined operations are insufficient for a given question.

**Step 3: Automated LLM Screening.** Using GPT-3.5-Turbo as the backbone LLM, we processed the entire WikiSQL dataset with prompts instructing the model to prioritize predefined functions and only invoke "`out_of_predefined()`" when no suitable predefined operation exists. This design allows the LLM to assess whether predefined operations are sufficient based on the question semantics, effectively identifying questions that require complex reasoning operations beyond the standard predefined function set.

**Step 4: Filtering and Dataset Formation.** QA pairs whose generated code invoked "`out_of_predefined()`" at least once were

extracted to form the WikiSQL-E dataset. This filtering process yielded 1,190 QA pairs spanning 925 tables, as reported in Table 5.

This automated construction process ensures that WikiSQL-E is not manually curated but systematically identified through an objective screening mechanism, capturing questions that are genuinely challenging for predefined-function-only methods.

We primarily use Denotation Accuracy (DA) and F1 scores to evaluate the overall performance of the methods. Additionally, we introduce a new metric called Calling Denotation Accuracy (**CDA**), which is designed to evaluate the correctness of the answers produced specifically by the "out-of-predefined" functions and we will define CDA in more detail below. These metrics help validate the effectiveness of the CRAFT module.

Experiments on the constructed dataset WikiSQL-E effectively explore our framework's improvements and capabilities in complex reasoning scenarios and provide a valuable dataset and baseline for future research in this area.

### A.2 Definition of Calling Denotation Accuracy

The Calling Denotation Accuracy (**CDA**) metric is designed to evaluate the correctness of answers generated for questions that have called "out-of-predefined" functions. We utilize the self-consistency (Wang et al., 2023) strategy in experiments. Specifically, for each question, the LLM generates $n$ distinct reasoning paths, and the final answer is determined by a majority vote among them. In our experiments, we set $n = 5$.

Let $Q_{total}$ be the set of all questions in a given dataset. A question $q$ is considered to have called "out-of-predefined" function if at least one of its $n = 5$ generated reasoning paths calls an "out-of-predefined" function. Let $Q_{calling}$ be the subset of such questions from $Q_{total}$, and $N_{calling}$ be its size.

For each question $q_{calling}$ in the $Q_{calling}$ subset, we first identify all reasoning paths that have called "out-of-predefined" functions. Assume that for each question $q_{calling}$, there are $k$ such paths (where $1 \leq k \leq n$). We then perform a majority vote only among the $k$ answers generated by these reasoning paths to determine a single, definitive answer $a_{calling}$ for that question. Let $N_{calling\_correct}$ be the number of questions in $Q_{calling}$ for which $a_{calling}$ matches the ground-truth answer. The Calling Denotation Accuracy (**CDA**) is then defined as:

$$\text{CDA} = \frac{N_{calling\_correct}}{N_{calling}}. \tag{17}$$

| Dataset | Scenario | Reasoning Steps | |
|---|---|---|---|
| | | Avg. | Max. |
| WebQSP | KGQA | 1.46 | 4 |
| CronQuestions | Temporal KGQA | 1.94 | 6 |
| WikiSQL | TableQA | 2.63 | 9 |
| TableBench-FC | Complex TableQA | 3.91 | 9 |
| TableBench-NR | Complex TableQA | 4.61 | 15 |
| WikiSQL-E | Complex TableQA | 4.03 | 9 |

Table 6: **Reasoning complexity statistics.** We list the applicable task scenarios for each dataset, and statistically analyze the average and maximum reasoning steps required to answer the questions in each dataset.

### A.3 Quantification of Dataset Reasoning Complexity

To quantify the reasoning complexity of datasets, inspired by TableBench (Wu et al., 2025), we adopt **Reasoning Steps** as the metric for measuring the reasoning difficulty of each dataset. The reasoning steps represent the number of intermediate operations required to derive the final answer from a given question.

For a dataset $\mathcal{D}$ containing $M$ question-answer pairs, let $step_i$ denote the reasoning steps required for the $i$-th question. We define the **Average Reasoning Steps** and **Maximum Reasoning Steps** as follows:

$$\text{Avg. Reasoning Steps} = \frac{1}{M}\sum_{i=1}^{M} step_i, \tag{18}$$

$$\text{Max. Reasoning Steps} = \max_{i \in 1,\ldots,M} step_i. \tag{19}$$

Table 6 presents the reasoning complexity statistics for all datasets used in our experiments. WebQSP, CronQuestions, and WikiSQL are relatively simple datasets with lower average reasoning steps (ranging from 1.46 to 2.63), which are employed to investigate the performance of our method on standard reasoning scenarios across different types of structured data. In contrast, TableBench-FC, TableBench-NR, and WikiSQL-E are more complex datasets with higher average reasoning steps (ranging from 3.91 to 4.61), which are utilized to evaluate the effectiveness of our method in complex reasoning scenarios.

| Model | Dataset | CDA (%) | DA (%) | F1 (%) |
|---|---|---|---|---|
| | | (TrustUQA / Ours) | (TrustUQA / Ours) | (TrustUQA / Ours) |
| GPT-3.5-Turbo | WikiSQL-E | 3.32 / 68.24 ↑ | 70.00 / 67.79 | 70.40 / 68.28 |
| | FactChecking | 0.00 / 18.75 ↑ | 50.00 / 61.46 ↑ | 56.61 / 65.44 ↑ |
| | Numerical Reasoning | 4.23 / 33.33 ↑ | 20.20 / 38.63 ↑ | 20.87 / 40.10 ↑ |
| GPT-4o-mini | WikiSQL-E | 8.33 / 61.29 ↑ | 74.54 / 79.04 ↑ | 75.06 / 79.66 ↑ |
| | FactChecking | 0.00 / 33.33 ↑ | 55.21 / 64.58 ↑ | 58.39 / 70.19 ↑ |
| | Numerical Reasoning | 2.82 / 33.00 ↑ | 21.72 / 40.66 ↑ | 22.73 / 42.57 ↑ |
| GPT-4.1 | FactChecking | 0.00 / 31.03 ↑ | 59.38 / 75.00 ↑ | 62.64 / 78.23 ↑ |
| | Numerical Reasoning | 0.00 / 45.99 ↑ | 29.29 / 56.06 ↑ | 28.89 / 56.37 ↑ |
| | WikiSQL-E | 6.98 / 53.57 ↑ | 79.44 / 87.34 ↑ | 79.52 / 87.58 ↑ |

Table 7: Performance comparison on different datasets across GPT-3.5-Turbo, GPT-4o-mini, and GPT-4.1. The results are presented in the format of *TrustUQA / Ours*. The ↑ symbol indicates that our method outperforms the baseline.

## B Other Implementation Details

All experiments are conducted on a server equipped with an Intel(R) Xeon(R) Gold 6148 CPU and three NVIDIA A100-SXM4-40GB GPUs, running on the Ubuntu 20.04.6 LTS operating system.

We use GPT-3.5 (GPT-3.5-Turbo) as the backbone LLM for the unified method on the WebQSP dataset, use GPT-4o-mini as the backbone LLM for the unified method on the CronQuestions and WikiSQL dataset. We use three different backbone LLMs, GPT-3.5, GPT-4o-mini, and GPT-4o, for experiments on different unified methods on the TableBench and WikiSQL-E dataset. And we use 4 open-source models (LLaMA-3.1-8B-Instruct, Qwen2.5-7B-Instruct, Qwen3-32B, DeepSeek-V3) and 8 closed-source models (Qwen-Max, Gemini-2.5-Flash, Gemini-2.5-Pro, GPT-3.5-Turbo, GPT-4o-mini, GPT-4o, GPT-5-mini, o4-mini) in the experiment that evaluate the generalization capabilities of CRAFTQA. For all LLM-based experiments, we use the self-consistency strategy of 5 times and use SentenceBERT (Reimers and Gurevych, 2019) as the dense text encoder.

**PoT Baseline Implementation.** Our PoT baseline is implemented following the original prompt design from PoT (Chen et al., 2023), which provides a general-purpose paradigm applicable to various structured data types. Specifically, the structured data and natural language question are directly provided to the LLM along with a zero-shot prompt that instructs the model to implement a "`solver()`" function with step-by-step Python program reasoning. The complete prompt template used in our PoT baseline experiments is presented in Table 8.

**PoT Prompt Template**

```
[data]: {data}
[Question]: {question}
[zero-shot-PoT Prompt]:
# Answer this question by implementing
  a solver() function.
def solver():
   # Let's write a Python program step
   by step, and then return the answer
```

Table 8: Prompt template for the PoT baseline (Chen et al., 2023). `{data}` and `{question}` denote placeholders for the structured data and the natural language question, respectively.

## C CDA Overall Assessment

Table 7 presents the complete numerical results corresponding to Figure 3 in the main text. This table provides a detailed comparison between CRAFTQA and the current state-of-the-art published method TrustUQA (Zhang et al., 2025) across three datasets (FactChecking, Numerical Reasoning, and WikiSQL-E) under different backbone LLMs (GPT-3.5-Turbo, GPT-4o-mini, and GPT-4.1). The evaluation specifically targets "out-of-predefined" scenarios where reasoning operations extend beyond predefined operator sets. Results are presented in the format of *TrustUQA / Ours*, and the ↑ symbol indicates that CRAFTQA achieves superior performance compared to the baseline. As shown in the table, CRAFTQA consistently outperforms TrustUQA across nearly all metrics and configurations, demonstrating its effectiveness in handling "out-of-predefined" reasoning scenarios.

| Method | Fact Checking | | | Num. Reasoning | | | Overall Metric | | |
|---|---|---|---|---|---|---|---|---|---|
| | CDA | DA | F1 | CDA | DA | F1 | CDA | DA | F1 |
| *Open-Source Models* | | | | | | | | | |
| CRAFTQA-LLaMA-3.1-8B | 25.0 | 57.3 | 61.2 | 30.7 | 31.1 | 32.5 | 29.6 | 36.2 | 38.1 |
| CRAFTQA-Qwen2.5-7B | 28.6 | 57.9 | 62.9 | 21.3 | 35.4 | 36.7 | 22.7 | 39.8 | 41.8 |
| CRAFTQA-Qwen3-32B | 32.4 | **72.9** | **76.2** | 43.4 | **53.8** | **54.4** | 41.3 | **57.5** | **58.7** |
| CRAFTQA-DeepSeek-V3 | 30.4 | 70.8 | 71.5 | 36.9 | 48.2 | 48.9 | 35.6 | 52.6 | 53.3 |
| *Closed-Source Models* | | | | | | | | | |
| CRAFTQA-Qwen-Max | 61.8 | 75.0 | 76.9 | 51.2 | 48.1 | 49.0 | 53.3 | 53.3 | 54.4 |
| CRAFTQA-Gemini-2.5-Flash | 48.8 | 75.0 | 76.0 | 49.0 | 54.8 | 55.4 | 49.0 | 58.7 | 59.4 |
| CRAFTQA-Gemini-2.5-Pro | 69.7 | 76.0 | 76.6 | 61.7 | 55.1 | 55.3 | 63.3 | 59.2 | 59.5 |
| CRAFTQA-GPT-3.5 | 18.8 | 61.5 | 65.4 | 33.3 | 38.6 | 40.1 | 30.5 | 43.1 | 45.0 |
| CRAFTQA-GPT-4o-mini | 33.3 | 64.6 | 70.2 | 33.0 | 40.7 | 42.6 | 33.1 | 45.4 | 48.0 |
| CRAFTQA-GPT-4o | 20.0 | 68.8 | 71.6 | 50.0 | 51.3 | 51.9 | 44.1 | 54.7 | 55.7 |
| CRAFTQA-o4-mini | 50.0 | **83.3** | 83.9 | 48.3 | 56.4 | 56.7 | 48.6 | 61.6 | 62.0 |
| CRAFTQA-GPT-5-mini | 45.5 | 82.3 | **84.9** | 50.0 | **57.8** | **59.2** | 49.1 | **62.6** | **64.2** |

Table 9: Generalization evaluation of CRAFTQA across different backbone LLMs on TableBench. CDA: Calling Denotation Accuracy, DA: Denotation Accuracy, F1: F1 Score.

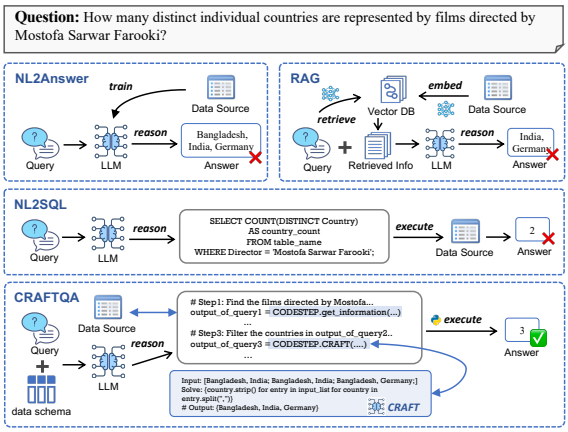


Figure 5: Comparison of NL2Answer, RAG, NL2SQL paradigm, and our proposed CRAFTQA.

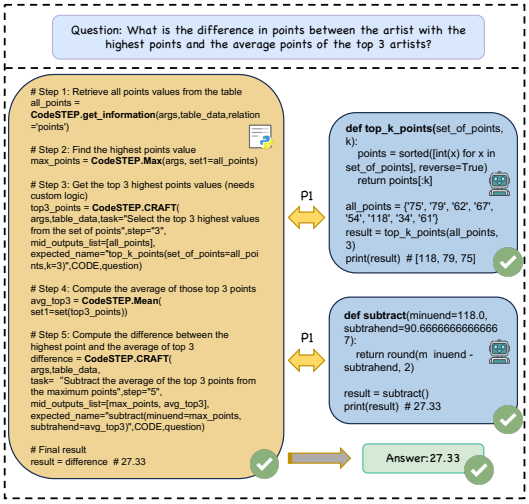


Figure 6: A more complex positive case study of the CRAFTQA framework

## D Generalization Across Backbone LLMs

Table 9 presents the complete numerical results corresponding to Figure 4 in the main text. This table evaluates the generalization capability of CRAFTQA across a diverse range of backbone LLMs, including both open-source models (LLaMA-3.1-8B, Qwen2.5-7B, Qwen3-32B, DeepSeek-V3) and closed-source models (Qwen-Max, Gemini-2.5-Flash/Pro, GPT-3.5, GPT-4o-mini, GPT-4o, o4-mini, GPT-5-mini). The evaluation is conducted on TableBench across Fact Checking and Numerical Reasoning tasks, reporting CDA, DA, and F1 metrics. The results demonstrate that CRAFTQA maintains robust performance across various backbone LLMs with different scales and architectures, and consistently benefits from the advancement of LLM capabilities, confirming its strong generalization potential.

## E Different Paradigm

Our method represents a fundamental evolution of the NL2Query paradigm beyond traditional approaches (e.g., NL2SQL (Liu et al., 2024b) and NL2SPARQL (Jung and Kim, 2020)), as illustrated in Figure 5.

## F Case Study

To further demonstrate CRAFTQA's capability in handling complex multistep reasoning tasks, we present an additional positive case for the question: "What is the difference in points between the artist

with the highest points and the average points of the top 3 artists?" Figure 6 illustrates the execution process for this sophisticated query that requires multiple custom operations.

This case exemplifies effective collaboration between CodeSTEP and CRAFT through multiple interactions. CodeSTEP first retrieves all points values from the table and identifies the maximum value using predefined functions. When encountering the need to select the top 3 values, which is a task beyond predefined operations, CodeSTEP appropriately delegates this to CRAFT with clear specifications. CRAFT responds by generating a top "k" points function that sorts the points in descending order and returns the first "k" elements, successfully extracting "[118, 79, 75]" from the data.

After computing the average of these top 3 values using the predefined Mean function, CodeSTEP faces another custom operation requirement: calculating the precise difference between two floating point numbers. Again, recognizing that this exceeds predefined capabilities, CodeSTEP invokes CRAFT with the specific values ("118.0" and "90.67"). CRAFT generates a subtract function with appropriate rounding to maintain numerical precision, yielding the final result of "27.33".

This case demonstrates several key strengths of the CRAFTQA framework: (1) CodeSTEP's ability to recognize when custom code generation is necessary and provide clear task specifications to CRAFT; (2) CRAFT's capability to generate appropriate functions based on task descriptions and context; (3) The seamless integration of multiple CRAFT invocations within a single reasoning chain; and (4) The framework's effectiveness in handling complex queries requiring both predefined operations and custom implementations. The successful execution showcases how modular design enables flexible problem solving while maintaining computational accuracy throughout the multistep process.

# G Prompt Template

## G.1 Prompt for CodeSTEP Generation

Table 10 shows the prompt template for CodeSTEP module.

## G.2 Prompt for CRAFT Module

Table 11 shows the prompt template for CRAFT module.

**System Prompt for CodeSTEP Module**

**Role & Task Description**

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.

**Core Function Definition**

The conditional graph query function is defined as: `CODESTEP.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.

**Constraints & Usage Guidelines**

[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 `CODESTEP.CRAFT()` to process the query.
3. Use Set and Calculator functions as necessary to complete the task.
4. Record whether `CODESTEP.CRAFT()` was used by setting the "`use_CRAFT`" variable to `True` or `False`.

**Preset Function List**

*Conditional Graph Query functions:*

- `CODESTEP.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:*

- `CODESTEP.set_union(set1, set2, set3=None, set4=None, set5=None)`: Get the union of multiple sets. Returns a set.
- `CODESTEP.set_intersection(set1, set2, set3=None, set4=None, set5=None)`: Get the intersection of multiple sets. Returns a set.
- `CODESTEP.set_difference(set1, set2)`: Get the difference between two sets. Returns a set.
- `CODESTEP.set_negation(table_data, set1)`: Get all rows except those in `set1`. Returns a set.

*Calculator functions:*

- `CODESTEP.Min(args=args, set1)`: Get the smallest element in `set1`. Returns a set of a numeric value.
- `CODESTEP.Max(args=args, set1)`: Get the largest element in `set1`. Returns a set of a numeric value.
- `CODESTEP.Mean(set1)`: Get the average value of all elements in `set1`. Returns a set of a numeric value.
- `CODESTEP.Count(set1)`: Get the number of elements in `set1`. Returns a set of a numeric value.
- `CODESTEP.Sum(set1)`: Get the sum of elements in `set1`. Returns a set of a numeric value.

*If further assistance is needed, use the* `CODESTEP.CRAFT()` *function:*

- `CODESTEP.CRAFT(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.

**Code Generation 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 `CODESTEP.CRAFT()` is used, record it in the '`use_CRAFT`' variable.

Table 10: Prompt template for the CodeSTEP module. The system prompt instructs the backbone LLM $\mathcal{M}_\theta$ to generate step-by-step executable Python code for structured data reasoning using predefined functions and the CRAFT interface.

**System Prompt for CRAFT Module**

**# 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 `CODESTEP.CRAFT()` 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.

Table 11: Prompt template for the CRAFT module. The system prompt guides the LLM $\mathcal{M}_{\theta'}$ to generate self-contained custom code functions for reasoning steps beyond predefined operations.