ProbeTQA: Verifiable Table Question Answering via Multi-Round Executable Probing

Anonymous ACL submission

Abstract

Empowering Large Language Models (LLMs) for tool-free Table Question Answering (TableQA) faces two critical bottlenecks: current "question-first" data synthesis often yields logic hallucinations, and standard Reinforcement Learning (RL) suffers from sparse rewards over long reasoning chains. In this paper, we propose **ProbeTQA**, a unified framework coupling execution-verified data construction with process-supervised RL. To eliminate data hallucinations, we shift to an **answer-first executable probing paradigm**. By allowing an agent to conduct multi-round exploratory programming on real tables, we ensure every synthesized instance is backed by a verifiable witness program. Furthermore, to resolve the sparse reward challenge, we distill these code trajectories into natural language and extract *execution-grounded process rubrics* directly from actual runtime states, providing highly reliable dense supervision. By applying a **Two-Stage Reward Curriculum** via Group Relative Policy Optimization (GRPO) using these rubrics, we successfully internalize complex tool-use reasoning into model parameters. Experimental results show our 8B model achieves state-of-the-art tool-free performance on complex benchmarks like **TableBench** (**65.8%** accuracy), surpassing GPT-5.4 and DeepSeek-v3.2 by **13.4%** and **6.8%**, respectively.

1 Introduction

Table Question Answering (TableQA) is a cornerstone task for automated data analysis, requiring multi-step logical and numerical reasoning over structured data (Pasupat and Liang, 2015; Cheng et al., 2021). Current LLM-based solutions predominantly rely on external tools (e.g., Code Interpreters) to perform such operations (Chen et al., 2022; Zhang et al., 2023). However, this tool-dependent paradigm introduces significant deployment constraints, such as inference latency, data privacy risks, and brittle system dependencies, which limit applicability in resource-constrained or offline settings (Qu et al., 2025). These limitations motivate us to ask: *Can we internalize complex tool-use reasoning directly into model parameters, enabling robust, **tool-free** TableQA without any external execution at inference time?*

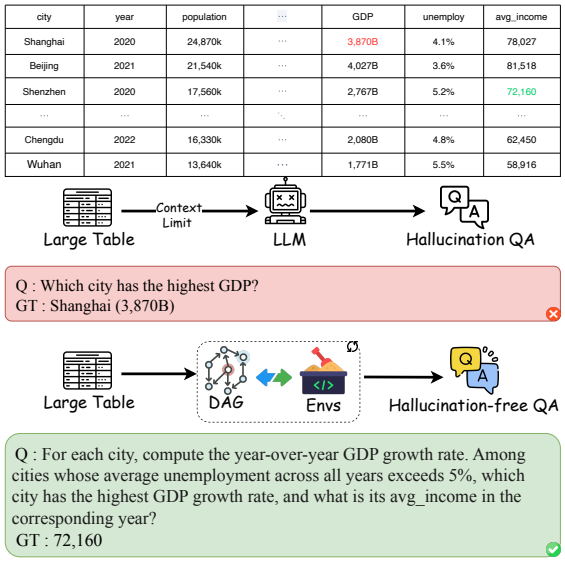

Figure 1: The question-first paradigm (top) directly prompts LLMs on large tables, producing hallucinated QA pairs with incorrect ground truths. Our answer-first paradigm (bottom) uses DAG-guided executable probing to generate complex, multi-step questions with verified answers.

Realizing this vision demands two fundamental ingredients: high-quality complex reasoning data and an effective RL alignment strategy. Unfortunately, both face critical bottlenecks. On the **data synthesis** side, existing methods adopt a "question-first" paradigm (Harsha et al., 2025), where LLMs first fabricate questions and then attempt to derive answers. Under long and complex tabular contexts, this approach suffers from severe logic hallucinations and frequently produces unexecutable reasoning paths with unreliable ground truths (Li et al.,

2024). On the **RL training** side, complex TableQA demands long-chain, multi-step reasoning, yet standard RL methods rely solely on outcome-based binary rewards (correct/incorrect). This creates a severe *sparse reward* problem: the model receives no learning signal for partially correct reasoning, leading to inefficient exploration and policy collapse (Lyu et al., 2025).

To overcome these challenges, we propose **ProbeTQA**, a unified framework coupling execution-verified data construction with process-supervised RL. Shifting from the hallucination-prone "question-first" paradigm, our core insight is an *answer-first* approach grounded in actual code execution. Our framework operates in three tightly integrated stages: **(1) Skeleton Construction.** We extract transition probabilities from real-world traces to sample DAG-structured operation skeletons, which serve as topological blueprints for complex queries. **(2) Multi-Round Executable Probing.** Guided by these skeletons, an agent interactively executes and refines probing code on real tables. This ensures every accepted instance is backed by a verified witness program, guaranteeing solvability by construction. **(3) QA & Rubric Polishing.** The validated code is verbalized into natural-language QA pairs, while its intermediate runtime states are extracted as *execution-grounded process rubrics*, providing dense, highly reliable supervision signals for RL.

Leveraging these execution-grounded rubrics, we optimize the model using Group Relative Policy Optimization (GRPO) (Liu et al., 2024) via a novel **Two-Stage Reward Curriculum**. To avoid the performance plateau often caused by prolonged process supervision, we apply dense rubrics strictly as an initial scaffold to overcome the sparse-reward cold start. The training then shifts to strict binary outcome rewards to enforce end-to-end precision, successfully breaking the local minimum and internalizing complex tool-use logic into the model's parameters. Empirical results demonstrate the effectiveness of ProbeTQA: our 8B model achieves state-of-the-art tool-free performance on complex benchmarks, reaching **65.8%** on TableBench (Wu et al., 2025). Notably, even without external execution tools, it outperforms much larger proprietary models, surpassing GPT-5.4 (OpenAI, 2026) by **13.4%** and DeepSeek-v3.2 (DeepSeek, 2025) by **6.8%**.

In summary, the main contributions of this work are three-fold:

- We introduce an **answer-first executable probing** paradigm for TableQA data synthesis, effectively resolving the logic hallucination and unexecutable path issues inherent in traditional "question-first" generation.
- We propose a novel method to extract **execution-grounded process rubrics** directly from actual code runtime states, providing highly reliable dense supervision for complex reasoning.
- By coupling our execution-verified data with a **Two-Stage Reward Curriculum** via GRPO, we successfully internalize tool-use capabilities into an 8B model, achieving new SOTA tool-free performance and outperforming proprietary giant models.

2 Related Work

Table QA and Data Synthesis. Table question answering spans semantic parsing, fact verification, and numerical reasoning. Traditional benchmarks drive progress but rely on costly, unscalable human annotations that often yield shallow queries (Pasupat and Liang, 2015; Zhong et al., 2017; Iyyer et al., 2017; Chen et al., 2019, 2020; Zhu et al., 2021; Nan et al., 2022). To mitigate annotation costs, recent efforts explore synthetic generation and table-centric pretraining (Liu et al., 2021; Jiang et al., 2022; Wang et al., 2023). However, unlike open-ended text, TableQA demands strict execution grounding. Conventional LLM-based synthesis over long tables frequently suffers from logical hallucinations—producing non-existent schema elements, unsupported predicates, or unverifiable answers.

Tool Use and Process Supervision. External tools significantly enhance complex reasoning reliability. Program-aided methods (Gao et al., 2023) and tool-augmented frameworks (Yao et al., 2022; Schick et al., 2023) interleave reasoning with programmatic execution. Concurrently, Process Reward Models (PRMs) provide crucial step-level feedback for long-chain reasoning (Lightman et al., 2024; Wang et al., 2024; Chen et al., 2024). This fine-grained supervision is further scaled by rubric-based LLM-as-a-judge evaluators (Liu et al., 2023; Li et al., 2025; Kim et al., 2024; Zheng et al., 2023). Nevertheless, current PRMs and evaluators heavily rely on human-authored or LLM-hallucinated rubrics. For structured tabular tasks, these prompt-

driven rubrics may appear linguistically plausible but inherently lack physical grounding in verifiable, intermediate execution states—a critical gap addressed by our framework.

3 Method

In this section, we introduce **ProbeTQA**, a framework that formulates TableQA data construction as an *answer-first executable probing* process to circumvent the hallucinations and unexecutable paths inherent in conventional "question-first" generation. As illustrated in Figure 2, our framework operates in three stages: (1) preparing tables alongside structural reasoning priors, (2) deploying a coding agent to interactively explore the table and discover a verified witness program, and (3) translating this program into a high-quality QA pair coupled with an execution-grounded process rubric.

3.1 Table Preparation and Structural Priors

A successful answer-first probing process relies on two foundational prerequisites: a reliable execution environment and a mechanism to guarantee reasoning complexity. Unleashing an LLM agent freely onto raw tabular data typically leads to frequent execution crashes due to ill-formatted schemas, or results in a degenerate exploration of trivial, single-hop queries. To overcome these pitfalls, the first stage of ProbeTQA prepares the essential raw materials and topological blueprints—strictly without generating any natural-language questions yet—through two sequential modules: Table Collection and Skeleton Construction.

Table Collection and Cleaning. To prevent execution failures caused by dirty data and to ensure scale diversity, we aggregate tabular data from multiple real-world sources. We slice excessively large tables into sub-tables of varied row and column dimensions. Furthermore, we apply lightweight sanitization to filter out ill-formatted numerical columns, ensuring that the resulting table pool supports a clean, deterministic Python execution environment.Crucially, to strictly prevent data contamination, we rigorously deduplicate the collected tables against all downstream evaluation test sets using a combination of exact schema hashing and content-based N-gram overlap filtering.

Structural Skeleton Construction. To explicitly prevent the agent from collapsing into the aforementioned trivial lookup queries, we inject *reasoning skeletons* as structural priors to enforce compositional diversity. We define a set of ten primitive operations $\mathcal{O}$ (e.g., `filter`, `aggregate`, `group`). A skeleton s is modeled as a Directed Acyclic Graph (DAG) over these primitives. The detailed definitions of these primitive operations are provided in Appendix A.

To ensure the generated DAGs resemble logical human queries rather than random combinations, we estimate data-driven prior distributions—specifically, the start prior P_{start}, end prior P_{end}, and transition prior P_{trans}—from 34,955 real-world reasoning traces.

For a linear skeleton $s_{\text{lin}} = (o_1, o_2, \ldots, o_L)$ composed of a sequence of operations, its structural prior score is calculated as:

$$\text{Score}(s_{\text{lin}}) = P_{\text{start}}(o_1) P_{\text{end}}(o_L) \prod_{t=1}^{L-1} P_{\text{trans}}(o_{t+1} \mid o_t) \tag{1}$$

We enumerate valid linear paths using a Depth-First Search over the transition matrix, pruning those with negligible probabilities.

To construct parallel DAGs for complex comparative reasoning, we merge two valid linear paths (s_a and s_b) into a sink operation m:

$$s_{\text{par}} = (s_a \parallel s_b) \rightarrow m, \tag{2}$$

where $m \in \{\texttt{compare}, \texttt{derive}\}$.

Skeleton Positioning. Crucially, these DAG skeletons are *not* text templates and do not bind any concrete columns, predicates, or entities. They strictly serve as abstract topological blueprints, handing over a structural intention to Stage 2 without generating any premature natural-language questions.

3.2 Answer-First Executable Probing

At the heart of ProbeTQA lies the answer-first executable probing process. Given a real-world table T and a sampled structural skeleton s, the objective is to instantiate the abstract topological prior into a concrete, verifiable reasoning path. Unlike conventional question-first pipelines that force an LLM to hallucinate an intention before inspecting the data, ProbeTQA discovers the valid reasoning path and the corresponding answer simultaneously through physical code execution.

To systematically achieve this, we formulate the probing process into three sequential modules: a multi-round probing loop to resolve data uncertainties, program submission to format the reasoning

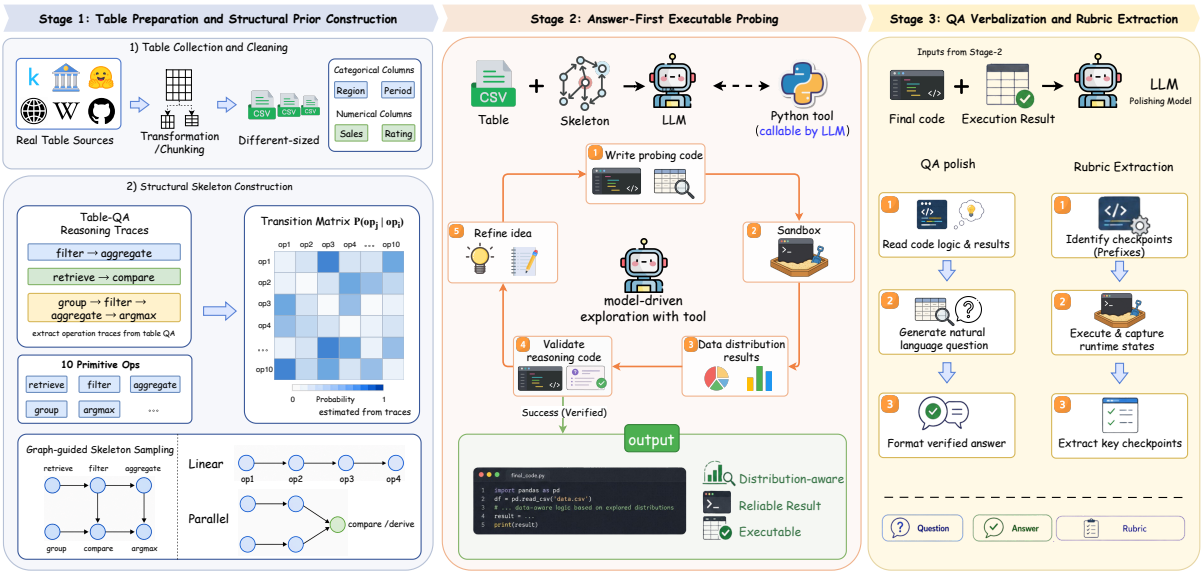

Figure 2: Overview of the ProbeTQA framework.

trace, and independent verification to guarantee correctness.

Module 1: Multi-Round Probing Loop. Because LLMs lack innate visibility into the exact cell values and implicit distributions of a table, zero-shot code generation frequently fails. To overcome this, the agent is situated in a secure Python sandbox environment, denoted as $\mathcal{E}_T(\cdot)$. At each round t, the agent observes the current exploration history $\mathcal{H}_{t-1}$ and dynamically writes a probing script z_t to explore the table schema, test logical predicates, or observe data distributions. The sandbox executes z_t and returns an observation o_t:

$$o_t = \mathcal{E}_T(z_t) \tag{3}$$

The exploration history is then updated to inform the subsequent reasoning step: $\mathcal{H}_t = \mathcal{H}_{t-1} \cup \{(z_t, o_t)\}$. This model-driven exploration ensures that the reasoning path is fundamentally grounded in actual data states rather than context-window guesses.

Module 2: Program Submission. The interactive loop continues until the agent successfully identifies a reliable reasoning path, capped at a maximum of $R = 20$ rounds. Upon success, the agent terminates the exploration by submitting a standalone final program c alongside a claimed final answer $\hat{a}$.

To guarantee process-level transparency for Stage 3, we enforce a strict *step-wise formatting* constraint on the submitted program c via explicit system prompt instructions. The program must sequentially assign critical intermediate results to explicit variables before printing the final answer. Long, chained expressions (e.g., deeply nested Pandas operations) are explicitly prohibited in the prompt, ensuring that no intermediate computational states are hidden.

Module 3: Independent Verification. Crucially, the submitted program and the claimed answer are never blindly trusted. The system independently re-executes the standalone program c in an isolated sandbox to capture the actual executed answer a:

$$a = \mathcal{E}_T(c) \tag{4}$$

An instance is accepted into the final candidate pool if and only if the independently executed answer exactly matches the claimed answer and does not evaluate to a degenerate value:

$$(a = \hat{a}) \wedge (a \notin \mathcal{D}) \tag{5}$$

where $\mathcal{D}$ denotes the set of invalid or meaningless outputs (e.g., empty sets, `NaN`, `Inf`, or non-serializable objects). Instances that trigger execution errors, exceed the round limit R, or fail the equivalence match are immediately discarded (the overall synthesis success rate is discussed in Section 4).

The Executable Witness Guarantee. Consequently, the solvability of our generated dataset is not achieved through heuristic post-hoc filtering. Instead, it is guaranteed *by construction*. Every accepted instance is backed by an **executable wit-**

ness program that constructively proves exactly how the verifiable answer is derived from the table.

3.3 Execution-Grounded Rubric Extraction and QA Verbalization

While the executable probing in Stage 2 guarantees the discovery of a valid reasoning path (program c) and its true answer (a), raw code trajectories alone are insufficient for training user-facing, tool-free TableQA models. The critical challenge at this final stage is twofold: first, how to translate the rigid code logic into natural-language queries without re-introducing logical hallucinations; second, how to extract dense supervision signals for Reinforcement Learning (RL) without relying on unreliable, LLM-guessed rubrics. To bridge this gap, given the table T, program c, and answer a, our framework deploys a polishing model to execute two parallel pipelines.

Pipeline 1: Code-to-Language QA Verbalization. To strictly preserve the answer-first guarantee, the model is prohibited from inventing a new question-answer pair from scratch. Instead, taking the executable code c and the verified answer a as inputs, the model identifies the core computational logic (e.g., filtering conditions, aggregation types) and transcribes them into a coherent natural-language query. Consequently, the generated question and the polished answer are intrinsically bound to the verified computational logic.

Pipeline 2: Execution-Grounded Rubric Extraction. Beyond the final QA pair, ProbeTQA extracts an *execution-grounded process rubric*. Traditional Process Reward Models (PRMs) or LLM-as-a-judge approaches typically rely on prompt-driven, hallucinated rubrics that lack factual grounding. In contrast, our rubrics are anchored directly in authentic runtime states.

Specifically, the system leverages the enforced step-wise formatting of program c, automatically slicing it by its intermediate variable assignments into a sequence of K key reasoning checkpoints (typically, $K \in [3, 7]$). Let $c_{1:i}$ denote the executable code prefix up to the i-th checkpoint. We independently execute these prefixes in the sandbox to capture their intermediate runtime values v_i. The final rubric $\mathcal{R}$ is defined as:

$$\mathcal{R} = \{(p_i, v_i)\}_{i=1}^{K}, \quad \text{where } v_i = \mathcal{E}_T(c_{1:i}) \quad (6)$$

Here, p_i represents the natural-language description of the i-th step, and v_i is the concrete state obtained from physical execution (e.g., a filtered `DataFrame` of 12 rows, or a grouped statistic of `2.5B`).

Application: Two-Stage Reward Scheduling for RL. The ultimate goal of extracting these execution-grounded rubrics is to internalize the complex tool-use capabilities into the model's parameters via RL (e.g., GRPO). In complex TableQA, relying solely on binary outcome rewards (0 or 1) leads to a severe sparse-reward cold start, as the model rarely guesses the exact long-chain logic from scratch. Conversely, relying indefinitely on process rubrics often causes the training to plateau in a suboptimal local minimum. Because rubrics award partial credits for intermediate steps, the gradient signal required to fix minor final-step errors becomes too weak to drive absolute end-to-end correctness.

Inspired by recent findings in RL grokking (Sun et al., 2025), we apply the extracted data through a *two-stage reward curriculum*. In the initial warm-up phase (e.g., the first 100 steps), we strictly apply the dense rubric $\mathcal{R}$ to provide partial credits. This acts as a critical scaffolding, helping the weak model quickly grasp intermediate tabular operations and escape the zero-reward trap. Once the model surpasses this capability threshold, we completely drop the process rubrics and switch to the strict binary outcome reward. This sudden removal of scaffolding forces the model to shift its focus entirely to absolute correctness, effectively breaking the performance plateau and pushing the policy to grok the final tool-free reasoning limits. Throughout both stages, we deploy DeepSeek-v3.2 as the underlying reward evaluator, dynamically switching its prompt from a rubric-augmented context to a strict outcome-matching context (detailed configurations are provided in Appendix F).

4 Experiment

4.1 Experimental Setup

Training Data Construction. We execute the ProbeTQA pipeline using Qwen3-8B on diverse tables (4 attempts per table). By strictly filtering for instances solvable on the first attempt (`pass@1`), we purposely discard trivial queries and mine complex reasoning paths. This yields a curated RL training set of about 6,000 high-complexity instances paired with execution-grounded rubrics. All datasets and models used in this work are publicly available for research use. We follow the original licenses and

Model	WikiTQ	TabFact	HiTab	TableBench	TableEval	ProbeTQA	Avg.
GPT-5.4	**88.86**	**93.58**	83.59	52.37	**84.27**	54.96	76.27
DeepSeek-v3.2	86.76	92.27	**84.27**	59.05	82.99	52.37	76.29
Qwen3-32B	85.80	91.98	83.34	64.79	80.86	52.80	76.60
Qwen3-14B	83.76	90.78	82.07	59.93	79.96	50.98	74.58
Table-R1-Zero-8B	76.61	87.43	82.58	40.52	53.09	24.07	60.72
TableGPT2-7B	76.68	87.71	82.83	39.16	53.12	29.60	61.52
TableGPT-R1-8B	84.58	92.11	81.25	62.30	79.99	52.66	75.48
Qwen3-4B	79.87	89.97	78.28	54.18	76.89	41.20	70.06
Qwen3-4B + RL	83.22	92.27	82.55	61.53	80.56	50.51	75.11
Qwen3-8B	81.13	90.19	79.20	57.00	78.88	45.79	72.03
Qwen3-8B + RL	84.45	93.43	83.77	**65.80**	82.87	**56.78**	**77.85**

Table 1: Main evaluation results on six benchmarks. All metrics represent the performance under the unified LLM-as-a-judge evaluation to ensure semantic equivalence, with the sole exception of TabFact, which naturally uses Exact Match (Accuracy). The highest and second-best scores are shown in bold and underlined, respectively.

terms of use of the corresponding artifacts.

Benchmarks and Metrics. We comprehensively evaluate our models on five diverse public benchmarks: WikiTableQuestions (WikiTQ) (Pasupat and Liang, 2015) and HiTab (Cheng et al., 2021) for complex compositional reasoning; TableEval (Zhu et al., 2025) and TableBench (Wu et al., 2025) for execution and instruction following; and TabFact (Chen et al., 2019) for binary fact verification. Additionally, we introduce **ProbeTQA-Test**, a curated set targeting long-tail complex reasoning. Because standard Exact Match (EM) suffers from severe false negatives in complex numerical tasks (e.g., penalizing "150.00" vs. "150"), we specifically employ GPT-5.4 as an LLM-as-a-judge (Zheng et al., 2023) for WikiTQ, HiTab, and TableBench to assess true semantic equivalence. Meanwhile, TableEval and ProbeTQA-Test natively utilize GPT-5.4 for evaluation by design. TabFact retains standard accuracy due to its binary nature (judge prompts are detailed in Appendix C).

Models and Baselines. We select Qwen3-4B and Qwen3-8B (Yang et al., 2025a) as our base policy models. We compare against state-of-the-art generalists (GPT-5.4 (OpenAI, 2026), DeepSeek-v3.2 (DeepSeek, 2025), Qwen3-14B/32B) and representative table-specific models (Table-R1-Zero-8B (Yang et al., 2025c), TableGPT-2 (Su et al., 2024), TableGPT-R1 (Yang et al., 2025b)).

Training and Reward Curriculum. We optimize our policy models using GRPO (Liu et al., 2024). For the main evaluation (Table 1), we adopt a **Two-Stage Reward Curriculum**: models are bootstrapped with our dense *Rubric Reward* for the first 100 steps to overcome cold-start bottlenecks, then abruptly transitioned to a sparse *Outcome Reward Model (ORM)* for the remaining steps to enforce end-to-end precision and prevent reward hacking. Comparative analyses against pure ORM/Rubric baselines are detailed in Section 4.4. And full implementation details are in Appendix B.

4.2 Main Results

Massive Uplift and Parameter Efficiency. Table 1 reveals massive performance gains driven by our RL framework. Specifically, `Qwen3-8B + RL` achieves striking absolute improvements of **+8.80%** on TableBench and **+10.99%** on ProbeTQA over its base model. It also exhibits exceptional parameter efficiency, with the compact `Qwen3-4B + RL` (61.53 on TableBench) outperforming the much larger `Qwen3-14B` (59.93). Crucially, while the pure outcome-based `Table-R1-Zero-8B` collapses on complex tasks (24.07 on ProbeTQA), our models thrive. This validates that our execution-grounded rubrics provide essential dense supervision, effectively overcoming the sparse-reward bottleneck of standard RL.

Surpassing SOTA on Complex Reasoning. When benchmarked against external baselines, our compact 8B model achieves tool-free superiority over both table-specific models and proprietary giants on execution-heavy tasks. Notably, `Qwen3-8B + RL` outclasses the SOTA table-specific `TableGPT-R1` (65.80 vs. 62.30 on TableBench). More impressively, on benchmarks demanding deep multi-step numerical aggregation and dynamic schema alignment, our 8B model defeats hundred-billion-parameter generalists. It surpasses

DeepSeek-v3.2 (65.80 vs. 59.05 on TableBench) and surprisingly defeats **GPT-5.4** (56.78 vs. 54.96 on ProbeTQA). This exposes the inherent weakness of current massive LLMs in tool-free multi-step calculation, and validates that internalizing physical execution paths is far more effective than merely scaling up static parameters.

Strong Generalization Across Task Types. Horizontally across all six diverse benchmarks, our RL-enhanced models exhibit highly robust generalization without suffering from catastrophic forgetting, culminating in a substantial surge in the overall average score. Specifically, `Qwen3-8B + RL` achieves an impressive average performance of **77.85%**, marking a **+5.82%** absolute improvement over its base model and ultimately surpassing both `Qwen3-32B` (76.60%) and **DeepSeek-v3.2** (76.28%) in overall tabular capability. This unified uplift spans a wide spectrum of tasks—from hierarchical compositional reasoning (`HiTab`) to complex instruction following (`TableBench`). Remarkably, even on the traditional NLP binary classification task `TabFact`, it achieves an outstanding **93.43**, securing the highest score among all open-weight models and closely rivaling GPT-5.4. This proves that our framework does not merely overfit to complex operational formats, but genuinely internalizes a generalizable capacity for tabular logic.

4.3 Analysis

Answer Groundedness and Factuality. Conventional data synthesis frequently suffers from logical hallucinations. As shown in Table 2, the *Self-instruction* baseline (traditional "question-first" LLM prompting) struggles severely, achieving only 61.5% Human and 58.4% Oracle Solvability, often hallucinating non-existent schema elements. In contrast, `ProbeTQA` reverse-engineers queries from verified execution states ("answer-first"), securing 96.0% Human Solvability. Grounding generation in physical sandbox execution ensures the RL training set remains completely free of unsolvable premises.

Synthesis Method	Human Solv. ↑	Oracle Solv. ↑
Self-instruction	61.5%	58.4%
ProbeTQA (Ours)	**96.0%**	**93.2%**

Table 2: Evaluation of answer groundedness. **Human Solv.** is verified on a subset ($N = 200$). **Oracle Solv.** is a scalable proxy measured by GPT-5.4 (`pass@4`).

Structural Complexity and Reasoning Depth. Questions must possess sufficient reasoning depth to prevent RL stagnation. In Table 3, we evaluate a *Code-only* baseline that omits our DAG skeletons (Section 3.1). Without skeletons, it produces trivial queries with a remarkably narrow complexity gap ($\Delta = 16.6$): the weak policy (Qwen3-8B) easily achieves 82.5% accuracy, closely trailing the strong model (Qwen3-235B at 99.1%).

Conversely, `ProbeTQA` depresses the weak model's accuracy to 56.8%. While increased difficulty slightly reduces the strong model's performance to 97.5%, queries remain highly solvable, creating a massive performance gap ($\Delta = 41.7$). This widened gap proves our structural priors inject non-trivial constraints—yielding questions that effectively stump weak models yet remain solvable by strong ones—thus providing a steep reasoning gradient for optimization.

Synthesis Method	Qwen3-8B ↓	Qwen3-235B ↓	Δ ↑
Code-only (No Skeleton)	82.5%	99.1%	16.6
ProbeTQA (Ours)	**56.8%**	**97.5%**	**41.7**

Table 3: Evaluation of question complexity ($N = 2,000$). The gap (Δ) between strong and weak models indicates reasoning depth, while lower accuracies (↓) reflect increased difficulty from structural priors.

Impact on Downstream RL Dynamics. The downstream manifestation of these data properties is illustrated in Figure 3. Models trained on *Self-instruction* suffer from an alignment tax (e.g., degrading from 79.2 to 77.9 on HiTab) because high hallucination noise provides misleading reward signals. The *Code-only* baseline avoids this noise but yields severely limited gains (59.3 on TableBench) as its shallow complexity fails to push the model's reasoning limits. By perfectly balancing absolute groundedness with high structural complexity, `ProbeTQA` maximizes policy learning, unlocking the highest downstream reasoning capabilities (66.7 on TableBench and 86.2 on HiTab).

4.4 Ablation Study

Training Optimization Trajectories. We first ablate the learning dynamics of three reward schemes on Qwen3-4B: pure Outcome Reward Model (ORM), pure Rubric, and the Two-Stage Reward Curriculum. As illustrated in Figure 4, training exclusively with pure ORM (green line) yields a severely stunted learning curve. Because complex tabular reasoning requires exact long-chain logic, zero-shot guessing the perfect answer is extremely

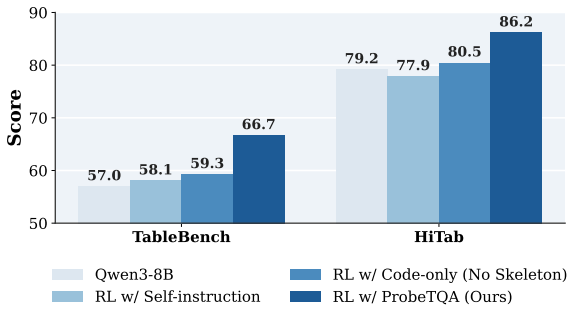

Figure 3: Downstream RL performance across synthesis paradigms. `ProbeTQA` avoids the hallucination tax of *Self-instruction* and the low complexity of *Code-only*, maximizing overall gains.

difficult, causing the policy to hit a sparse-reward wall. Conversely, the pure Rubric reward (blue line) enables rapid bootstrapping during the first 100 steps by providing dense, partial credits. However, its trajectory ultimately plateaus. Because the rubric assigns partial scores for intermediate steps, the gradient signal to fix minor final-step errors becomes weak. Consequently, the model settles into a *suboptimal local minimum*, lacking the strict enforcement needed for absolute correctness. The Two-Stage Reward Curriculum (red line) neutralizes both limitations. It leverages the dense rubric for 100 steps to cross the cold-start threshold, then abruptly transitions to strict ORM (marked by the vertical dashed line). Stripped of partial scaffolding, the policy is forced to refine its end-to-end logic, successfully breaking the plateau and achieving the highest optimization score.

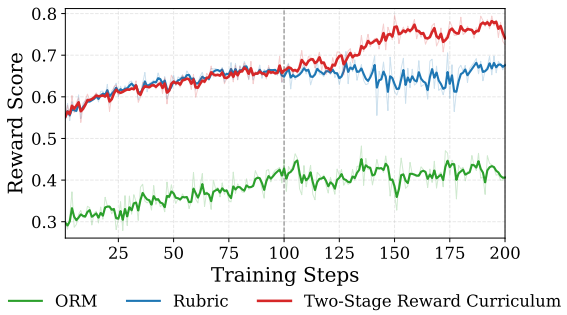

Figure 4: Training trajectories (mean reward scores) over 200 steps. The dashed line marks the transition from Rubric to ORM.

Downstream Performance Capabilities. The optimized training dynamics directly translate to superior reasoning capabilities on unseen test sets. As shown in Figure 5, the pure ORM policy fails to generalize effectively due to severe under-exploration during training, yielding only 56.7 on TableBench. The pure Rubric policy significantly improves this baseline but is ultimately capped by its inability to enforce absolute final-answer precision (60.4 on TableBench). By seamlessly combining rapid bootstrapping with strict end-to-end verification, the Two-Stage Reward Curriculum avoids early-stage stagnation and late-stage local minima, achieving the highest final accuracy across complex benchmarks (61.6 on TableBench and 82.5 on HiTab). This validates that our scheduling strategy is the optimal recipe for tool-free tabular RL.

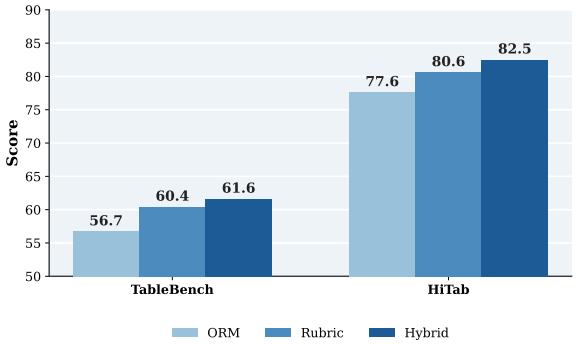

Figure 5: Downstream performance comparison at step 200 (Qwen3-4B). The Two-Stage Curriculum achieves the highest final accuracy.

5 Conclusion

We presented **ProbeTQA**, an execution-verified framework empowering compact LLMs with state-of-the-art tool-free TableQA capabilities. To eliminate hallucinations inherent in "question-first" data synthesis, we introduce an *answer-first executable probing* paradigm. Guided by DAG skeletons, an agent interactively explores real tables, guaranteeing complex and verifiable queries by construction. Furthermore, to address sparse RL rewards, we extract *execution-grounded process rubrics* directly from actual runtime states. Coupling these dense physical signals with a *Two-Stage Curriculum*, we effectively prevent reward hacking and internalize complex tool-use logic. Empirically, our RL-enhanced 8B model achieves unprecedented tool-free performance, successfully bridging the parameter scale gap to outperform massive proprietary generalists like DeepSeek-v3.2 and GPT-5.4. Ultimately, ProbeTQA paves a highly reliable path for hallucination-free data synthesis and complex reasoning alignment.

Limitations

We identify three limitations in our current work, which also point to future directions.

Emphasis on verifiable reasoning types. By design, ProbeTQA prioritizes reasoning operations that can be explicitly grounded in code, such as filtering, grouping, and aggregation. This deliberate scope ensures absolute correctness for process-level supervision. While it does not focus on subjective queries requiring external commonsense, it rigorously covers the core algorithmic operations that are central to reliable tabular analysis. Future iterations could seamlessly bridge this gap by interleaving our executable probing with semantic retrieval or external API tools to encompass open-ended factual reasoning.

Focus on structured environments and exploration cost. To guarantee reliable code execution and strict intermediate-state extraction, our current framework natively operates on machine-readable formats like CSV and HTML. Extending this highly precise probing mechanism to visual or unstructured documents requires integrating standard OCR or layout-parsing pipelines as a prerequisite. Additionally, while the multi-round sandbox exploration inherently incurs a higher computational cost during data synthesis, we consider this an acceptable upfront investment. It yields the high-fidelity supervision signals necessary to ultimately empower a compact, tool-free model with zero-latency inference.

Single-table constraints and multi-table reasoning. Currently, our DAG skeletons and interactive probing environments are tailored for single-table scenarios, which aligns with the scope of mainstream TableQA benchmarks. However, real-world enterprise databases often require complex relational reasoning, such as dynamic JOIN operations across multiple highly normalized tables. Expanding our answer-first probing framework to effectively navigate multi-table schemas, foreign-key dependencies, and cross-table aggregations represents a challenging yet highly valuable direction for future structural prior designs.

References

Guoxin Chen, Minpeng Liao, Chengxi Li, and Kai Fan. 2024. Alphamath almost zero: process supervision without process. *Advances in Neural Information Processing Systems*, 37:27689–27724.

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

Wenhu Chen, Hanwen Zha, Zhiyu Chen, Wenhan Xiong, Hong Wang, and William Yang Wang. 2020. Hybridqa: A dataset of multi-hop question answering over tabular and textual data. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1026–1036.

Zhoujun Cheng, Haoyu Caps Dong, Zhiruo Wang, Ran Jia, Jiaqi Guo, Yan Gao, Shi Han, Jian-Guang Lou, and Dongmei Zhang. 2021. Hitab: A hierarchical table dataset for question answering and natural language generation. *arXiv preprint arXiv:2108.06712*.

DeepSeek. 2025. Deepseek-v3.2 release. https://api-docs.deepseek.com/news/news251201. Accessed: 2026-05-16.

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.

Chetan Harsha, Karmvir Singh Phogat, Sridhar Dasaratha, Sai Akhil Puranam, and Shashishekar Ramakrishna. 2025. Synthetic data generation using large language models for financial question answering. In *Proceedings of the Joint Workshop of the 9th Financial Technology and Natural Language Processing (FinNLP), the 6th Financial Narrative Processing (FNP), and the 1st Workshop on Large Language Models for Finance and Legal (LLMFinLegal)*, pages 76–95.

Mohit Iyyer, Wen-tau Yih, and Ming-Wei Chang. 2017. Search-based neural structured learning for sequential question answering. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1821–1831.

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

Seungone Kim, Jay Shin, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Ryan Shin, Sungdong Kim, James Thorne, Minjoon Seo, and 1 others. 2024.

Prometheus: Inducing fine-grained evaluation capability in language models. In *International Conference on Learning Representations*, volume 2024, pages 29927–29962.

Yue Li, Meng Tian, Zhenyu Lin, Jiangtong Zhu, Dechang Zhu, Haiqiang Liu, Yueyi Zhang, Zhiwei Xiong, and Xinhai Zhao. 2025. Fine-grained evaluation of large vision-language models in autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9431–9442.

Zhuowan Li, Bhavan Jasani, Peng Tang, and Shabnam Ghadar. 2024. Synthesize step-by-step: Tools templates and llms as data generators for reasoning-based chart vqa. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13613–13623.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pages 39578–39601.

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

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*.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 2511–2522.

Chengqi Lyu, Songyang Gao, Yuzhe Gu, Wenwei Zhang, Jianfei Gao, Kuikun Liu, Ziyi Wang, Shuaibin Li, Qian Zhao, Haian Huang, and 1 others. 2025. Exploring the limit of outcome reward for learning mathematical reasoning. *arXiv preprint arXiv:2502.06781*.

Linyong Nan, Chiachun Hsieh, Ziming Mao, Xi Victoria Lin, Neha Verma, Rui Zhang, Wojciech Kryściński, Hailey Schoelkopf, Riley Kong, Xiangru Tang, and 1 others. 2022. Fetaqa: Free-form table question answering. *Transactions of the Association for Computational Linguistics*, 10:35–49.

OpenAI. 2026. Introducing gpt-5.4. https://openai.com/index/introducing-gpt-5-4/. Accessed: 2026-05-16.

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

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551.

Aofeng Su, Aowen Wang, Chao Ye, Chen Zhou, Ga Zhang, Gang Chen, Guangcheng Zhu, Haobo Wang, Haokai Xu, Hao Chen, and 1 others. 2024. Tablegpt2: A large multimodal model with tabular data integration. *arXiv preprint arXiv:2411.02059*.

Yiyou Sun, Yuhan Cao, Pohao Huang, Haoyue Bai, Hannaneh Hajishirzi, Nouha Dziri, and Dawn Song. 2025. Rl grokking recipe: How does rl unlock and transfer new algorithms in llms? *arXiv preprint arXiv:2509.21016*.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pages 13484–13508.

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.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025a. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Saisai Yang, Qingyi Huang, Jing Yuan, Liangyu Zha, Kai Tang, Yuhang Yang, Ning Wang, Yucheng Wei,

Liyao Li, Wentao Ye, and 1 others. 2025b. Tablegpt-r1: Advancing tabular reasoning through reinforcement learning. *arXiv preprint arXiv:2512.20312*.

Zheyuan Yang, Lyuhao Chen, Arman Cohan, and Yilun Zhao. 2025c. Table-r1: Inference-time scaling for table reasoning. *arXiv preprint arXiv:2505.23621*.

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

Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M Patel. 2023. Reactable: enhancing react for table question answering. *arXiv preprint arXiv:2310.00815*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623.

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. 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 3277–3287.

Junnan Zhu, Jingyi Wang, Bohan Yu, Xiaoyu Wu, Junbo Li, Lei Wang, and Nan Xu. 2025. Tableeval: A real-world benchmark for complex, multilingual, and multi-structured table question answering. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 7137–7157.

A Primitive Operations for Skeleton Construction

To construct the reasoning skeletons, we define a set of 10 primitive tabular operations. These operations are abstracted from real-world table reasoning traces and serve as the foundational building blocks for our structural priors. Table 4 provides the complete list of these primitives along with their descriptions.

Operation	Description
`retrieve`	Retrieve cells, rows, or entities from a table.
`filter`	Select rows satisfying one or more conditions.
`aggregate`	Compute summary values such as sum, mean, maximum, or minimum.
`argmax`	Locate the entity associated with an extreme value.
`compare`	Compare quantities, entities, or results from different branches.
`count`	Count rows, entities, or distinct values.
`derive`	Compute a new value or metric from existing columns.
`locate`	Find a target entity satisfying specified constraints.
`group`	Partition rows according to categorical or discrete columns.
`rank`	Sort or rank entities according to a given criterion.

Table 4: Complete list of primitive operations used to generate reasoning skeletons.

B Experimental Settings

In this appendix, we detail the experimental and hyperparameter configurations used during the reinforcement learning (RL) training phase of our policy models.

Algorithm and Objective. We adopt Group Relative Policy Optimization (GRPO) (Liu et al., 2024) as the core RL algorithm. GRPO eliminates the need for an explicit critic network by sampling a group of outputs for each input prompt and normalizing the rewards within the group. This significantly reduces memory overhead during training, allowing us to train larger policy backbones under a unified framework.

Sampling Configurations. During the rollout phase, we use a constant temperature of 1.0 and a Top-P value of 1.0 to encourage diverse exploration of reasoning paths. For each input prompt, the policy generates $n = 8$ independent candidate responses (rollout samples). To accommodate complex reasoning chains and extensive code execution traces, the maximum prompt length is set to $8,192$ tokens, and the maximum response length is extended to $16,384$ tokens.

Category	Configuration
Algorithm	RL Algorithm: GRPO
Backbone	Model: Qwen3-4B / Qwen3-8B
Sampling	Train Temperature: 1.0 Train Top-P: 1.0, Top-K: -1 Rollout Samples (n): 8 Max Prompt Length: 8,192 Max Response Length: 16,384
Training	Optimizer: AdamW Learning Rate: 1×10^{-6} Warmup Steps: 10 Weight Decay: 0.1 Training Batch Size: 256 Mini Batch Size: 64
Hardware	GPUs: 8× NVIDIA H200

Table 5: Hyperparameter configurations for the reinforcement learning training phase.

Training and Optimization Parameters. The model weights are optimized using the AdamW optimizer. We apply a constant learning rate of 1×10^{-6} with a brief warmup phase of 10 steps to ensure training stability. The training batch size is configured to 256 prompts per step, which is processed using a mini-batch size of 64. All RL experiments are executed on an industrial cluster equipped with $8\times$ NVIDIA H200 GPUs. The detailed hyperparameter checklist is summarized in Table 5.

C Exact Match Limitations and LLM-based Evaluation

Exact match (EM) is simple and reproducible, and is widely used in table question answering benchmarks. However, EM can be overly strict when a model prediction is semantically correct but differs from the reference answer in surface form. This issue is particularly common in WikiTQ, HiTab, and TableBench, where answers may involve formatting variations, aliases, separators for list answers, abbreviations, or numeric normalization. Table 6 shows representative cases where EM may reject answers that are semantically consistent with the reference. To quantitatively illustrate this discrepancy, Table 7 compares the standard EM scores against the LLM-as-a-judge evaluations across WikiTQ, HiTab, and TableBench, clearly demonstrating how EM underestimates the true semantic reasoning capabilities of the models.

To mitigate such false negatives, we exclusively

Issue Type	Reference	Model Answer
Hyphen variation	1982-1985	1982–1985
List separator	Jahaira Novgorodova | Carmen Jenockova	Jahaira Novgorodova, Carmen Jenockova
Whitespace	Engr.Sikandar Zaman	Engr. Sikandar Zaman
Partial entity name	DaMarcus Beasley	Beasley
Number normalization	6	six
More complete phrasing	less	less than 10
Additional abbreviation	West Germany	West Germany (FRG)
Entity abbreviation	Japan (JPN) | South Korea (KOR)	Japan, South Korea
More complete answer	male	more male

Table 6: Representative WikiTQ cases where exact match may reject semantically correct model answers due to surface-form mismatch.

employ GPT-5.4 as an LLM-based evaluator for selected benchmarks. Following the evaluation style used in Reasoning-Table, the judge is given the original question, the reference answer, and the candidate answer, and is asked to determine whether the two answers are semantically consistent. The prompt used for evaluation is shown in Figure 6.

Standardized Inference Setup. To ensure a fair and rigorous comparison across all evaluated models—including SOTA API models and our trained policy models—we strictly standardize the inference settings. Specifically, the evaluation prompts for all API models are kept strictly identical to those used for our deployed models. We set the decoding temperature uniformly to 0.6 across all evaluations. Furthermore, we explicitly disable the use of any external tools (e.g., code interpreters or web search) to purely assess the models' native tabular reasoning capabilities. To eliminate formatting bias, identical table serialization formats are applied to all models. We also confirm that no context truncation occurred during evaluation, as all serialized table inputs fit comfortably within the respective models' context windows. Finally, all predictions are generated via a single-pass sampling process, without relying on multiple sampling or majority voting techniques.

D System Prompts Overview

The detailed system prompt configurations deployed in our framework are provided for reference. Specifically, Figure 7 illustrates the instructions and operational constraints designed for the data-probing agent, which conducts the multi-round exploration phase. Figure 8 details the prompt template provided to the rubric extraction agent for reverse-engineering the verified code into execution-grounded checkpoints and natural-language questions.

E Agent Interaction Trace

To provide a complete illustration of the framework, we present the initial task contexts alongside their corresponding execution traces. Figure 9 defines the starting data and skeleton context for the multi-round exploration phase, whose trial-and-observation steps are detailed in Figures 10 to 15. Similarly, Figure 16 presents the verified program and target skeleton serving as the inputs for the subsequent decomposition stage, with its execution-grounded rubric generation trace shown in Figures 17 to 20.

F Evaluator Prompts for Reward Calculation

To compute the reward signals during the reinforcement learning phase, we deploy DeepSeek-v3.2 as a dynamic evaluator. Figure 22 details the rubric-augmented prompt used during the initial warm-up stage, which instructs the evaluator to verify intermediate reasoning steps against the execution-grounded rubric and assign precise partial credits based on the step completion ratio. In contrast, Figure 21 illustrates the Outcome Reward Model (ORM) prompt used in the second stage. By removing all rubric scaffolding, this prompt restricts the evaluator to issuing a strict binary score (0 or 1) based solely on the end-to-end semantic equivalence of the final answer, ensuring the policy ultimately optimizes for absolute correctness without relying on partial reward hacking.

G Manual Data Quality Inspection Details

The detailed protocol and strict criteria used by student volunteers to evaluate the *Human Solvability* of 200 sampled instances per method are presented in Figure 23.

Model	WikiTQ		HiTab		TableBench	
	EM	Judge	EM	Judge	EM	Judge
GPT-5.4	85.01	88.86	78.72	83.59	46.83	52.37
DeepSeek-v3.2	82.53	86.76	78.91	84.27	52.44	59.05
Qwen3-32B	82.00	85.80	79.67	83.34	58.00	64.79
Qwen3-14B	79.71	83.76	77.02	82.07	52.5	59.93
Table-R1-Zero-8B	40.68	76.61	81.00	82.58	34.33	40.52
TableGPT2-7B	40.33	76.68	81.19	82.83	32.99	39.16
TableGPT-R1-8B	81.01	84.58	77.15	81.25	56.08	62.30
Qwen3-4B	76.30	79.87	74.75	78.28	48.27	54.18
Qwen3-4B + RL	79.56	83.22	78.13	82.55	55.84	61.53
Qwen3-8B	77.60	81.13	75.33	79.20	50.85	57.00
Qwen3-8B + RL	80.82	84.45	79.43	83.77	55.74	65.80

Table 7: Quantitative comparison of Exact Match (EM) and LLM-as-a-judge evaluation across TableBench, WikiTQ, and HiTab. The significant gap between the two metrics illustrates how EM severely underestimates the true semantic reasoning capabilities of the models due to surface-form mismatches.

Evaluation Prompt

Here is the original question, the correct answer, and the candidate answer. Please evaluate whether the correct answer and the candidate answer are consistent.

Evaluation Rules:
1. Semantic Consistency (Focus on meaning) - Answers with the SAME core meaning are consistent, even if worded differently. - Focus on factual correctness, not exact phrasing.
2. Number and Unit Equivalence - Different formats: 1,000,000 = 1000000 = 1 million - Unit conversions: 5 km = 5000 m - Format differences: 2023-10-01 = October 1, 2023 - Minor rounding (within 0.5%) is acceptable: 12.5 $\approx$ 12.54
3. List/Multiple Value Answers - PARTIAL MATCH ALLOWED - Candidate answer MUST contain ALL required items. - Additional reasonable items are ACCEPTABLE. - Order does NOT matter.
4. Text Comparison - Case-insensitive: "USA" = "usa" - Extra spaces ignored; minor formatting differences ignored.

Examples:
Question: What is the capital of France?
Candidate Answer: The capital of France is Paris
Correct Answer: Paris is the capital of France
Consistent: Yes

Question: What is the distance from Paris to London?
Candidate Answer: 5 km / Correct Answer: 5000 m
Consistent: Yes

Question: How many people live in the city?
Candidate Answer: 1 million / Correct Answer: 1000000
Consistent: Yes

Question: What is the temperature in Paris?
Candidate Answer: 25°C / Correct Answer: 77°F
Consistent: No (Reason: Different values, not equivalent)

Question: In which three consecutive years was the record the same?
Candidate Answer: 1971, 1972, 1976 / Correct Answer: 2004|2005|2006
Consistent: No (Reason: Completely different years)

Your Task:
Respond with only "Yes" or "No" (without quotes). Do not include a rationale.
Question: {question}
Candidate Answer: {candidate_answer}
Correct Answer: {correct_answer}
Consistent:

Figure 6: The evaluation prompt template.

System Prompt — Data-Probing Agent

You are an expert data-probing agent.
You will receive a data table (which may be provided as a CSV, JSON, or HTML file) and an abstract reasoning skeleton (a topological sequence of table operations). Your job is to instantiate this skeleton into a concrete, executable reasoning path over the real table.
You must NOT generate any natural language questions. Your sole goal is to discover a reliable, multi-step computation path that yields a short, definitive answer.

What Makes a Good Reasoning Path:
Good paths require "looking up, filtering, and connecting" across the table, simulating human analytical behavior:
- "Find entity A -> look up A's attribute -> use that attribute to filter -> count/find result"
- "Among rows meeting condition X, which group has the highest total Y?"
- "The item ranked #1 in metric A — what is its rank in metric B?"
Bad paths are just trivial math formulas over columns:
- "Calculate (col_A * col_B) / (col_C + col_D) for each row, then find the max"
- "Compute the Pearson correlation between X and Y"
- "What is the standard deviation of column Z divided by the mean?"

Anti-Collapse Rules:
- Rule 1 (Strict Dependency): Each step must strictly use the previous step's output. Every operation in the provided skeleton must be necessary.
- Rule 2 (Column Diversity): The final answer must depend on multiple columns interacting (minimum 3 columns), not just a single column lookup.
- Rule 3 (Entity Hops for Deep Skeletons): If the skeleton depth >= 3, you should incorporate entity hopping: "Find entity A based on criterion X -> use A's attribute Y as a new filter condition."

Answer Format Rules (Critical):
The executed answer MUST be one of these forms:
- A short entity name (person, place, category): e.g., "China", "John Smith", "Action"
- An integer: e.g., "5", "142"
- A simple decimal (max 2 decimal places): e.g., "42.5", "3.14"
- A percentage (max 1 decimal place): e.g., "23.5%"
- A short comma-separated list: e.g., "Apple, Google, Microsoft"
Do not produce formulas, large unformatted numbers (e.g., 2415168000), or overly precise decimals. Round numbers directly in your Python code if necessary.

Multi-Round Exploration Phase:
1. First, write probing scripts to load the data (using appropriate pandas methods like pd.read_csv(), pd.read_json(), or pd.read_html() depending on the provided file format), inspect schemas, check column data types, and find interesting patterns or valid threshold values.
2. Observe the execution results to verify if your intended filters or aggregations are logically sound and yield non-empty results.
3. Iteratively refine your code until the computation path successfully instantiates the entire skeleton.

Final Submission Contract (Rubric-Friendly Code):
Once you find a valid path, submit the complete, standalone program:
- final_code: Must be fully executable from data loading (pd.read_*) to print(answer).
- No Chained Expressions: To allow the system to extract process-level rubrics, you MUST assign major intermediate reasoning results to explicit variables (e.g., filtered_df = ..., max_val = ...). Do NOT chain multiple pandas operations in a single line.
- answer: Must exactly match the printed output of your script.

Figure 7: The system prompt configuration for the data-probing agent.

System Prompt — Rubric Extraction Agent

You are an expert code-to-language verbalization and reasoning extraction agent. Your task is to reverse-engineer a verified Python reasoning program into a natural-language Question, its Final Answer, and an Execution-Grounded Rubric.

Inputs:
You will receive:
- Verified Code: A standalone Python script that successfully computes an answer from a table.
- Reasoning Skeleton: The abstract logical topology (e.g., filter -> group -> aggregate) that inspired this code. Use this as a reference for determining the granularity of your rubric steps.
- Executed Answer: The actual final output produced by the code.

Task 1: Question Verbalization:
Translate the exact filtering, comparison, ranking, and aggregation logic of the Verified Code into a fluent, natural-language Question.
- Constraint 1: Do not mention pandas syntax, programmatic operations, or specific table column names (e.g., instead of "where the 'dept_id' is 5", use "in the sales department").
- Constraint 2: The question must be perfectly answerable using *only* the reasoning path defined in the code. Do not hallucinate extra conditions.

Task 2: Execution-Grounded Rubric Extraction:
Decompose the Verified Code into a human-readable reasoning rubric.
- CRITICAL (Tool Use): Do not guess the intermediate states! You must use the execute_code tool to run the code segments and observe the actual runtime states (e.g., filtered row counts, intermediate maximums, specific entity names).
- Semantic Checkpoints: Write step descriptions in natural language reflecting the analytical intent. Do not describe Python syntax (e.g., use "Filter for films sharing a genre with the anchor", NOT "Execute df[df['genre'].apply(...)]").
- Specific Expected Values: Your Expected Value for each step must be a concrete runtime observation (e.g., "11 candidates remain", "The top scorer is John (98 points)").

Strict Output Format:
You must submit your final result exactly in the following plain text format:
Question: <Your generated natural language question>
Answer: <The Executed Answer>
Answer Type: <numeric / entity / list / boolean / verification>
Rubric:
Step 1 [<Natural language description of the logical operation>]: Expected Value = <Concrete runtime state/value observed>
Step 2 [<Natural language description of the logical operation>]: Expected Value = <Concrete runtime state/value observed>
...

Figure 8: The system prompt configuration for the rubric extraction agent.

Input / Task Context

Table: A 27-row × 11-column movie catalog
Skeleton: A depth-5 lookup skeleton with the following chain:

`retrieve` → `retrieve` → `filter` → `rank` → `locate`

Figure 9: The input and task context configuration.

Execution Log — Turn 1

Thought: I haven't seen this table yet. To plan a depth-5 chain I need to know its shape, column types, and what kind of values live in each column — especially whether there is a numeric column I can rank on and a column that can carry an attribute set from one row to a filter on others.

Code:

```
df = pd.read_csv(CSV)
print(df.shape)
print(df.dtypes)
print(df.head(3).to_string())
```

Observe:

```
(27, 11)
title object · year float · kind object · genre object · rating float · vote float ...
0 Ruby's Bucket...    2001   tv movie   ['Drama','Music']    6.0    317
1 Battalion of...     1905   movie     ['Doc','Short']      NaN    NaN
```

Figure 10: The process output of Turn 1.

Execution Log — Turn 2

Thought: rating and vote look like the two numeric columns. Before I commit to using either one as the ranking key, I want to confirm they are not too sparse and that they actually spread over a range — otherwise argmax / sort would be trivial or undefined.

Code:

```
print(df[['rating','vote']].notna().sum())
print(df['rating'].describe())
print('vote min / max:', df['vote'].min(), df['vote'].max())
```

Observe:

```
rating 22, vote 22
rating: count 22, mean 7.18, std 1.20, min 5.10, 25% 6.30, 50% 7.20, max 9.30
vote min / max: 3.0   2462087.0
```

Figure 11: The process output of Turn 2.

Execution Log — Turn 3

Thought: I noticed genre looks like a stringified list. If it is genuinely multi-label per row, it can serve as the bridge column... I need to confirm it parses cleanly as a Python list.

Code:

```
for g in df['genre'].dropna().head(5).tolist():
    print(repr(g), '->', ast.literal_eval(g))
```

Observe:

```
"['Drama', 'Music']" → ['Drama', 'Music']
"['Documentary', 'Short']" → ['Documentary', 'Short']
"['Action','Adventure','Drama','Sci-Fi']" → ['Action','Adventure','Drama','Sci-Fi']
```

Figure 12: The process output of Turn 3.

Execution Log — Turn 4

Thought: The schema permits a clean instantiation of the skeleton: use rating for the first retrieve... To anchor the chain I first need the actual top-rated row.

Code:

```
top_idx = df['rating'].idxmax()
print('top_idx =', top_idx)
print(df.loc[top_idx, ['title','rating','genre','vote']].to_string())
```

Observe:

```
top_idx = 19
title: The Shawshank Redemption, rating: 9.3, genre: ['Drama'], vote: 2462087.0
```

Figure 13: The process output of Turn 4.

Execution Log — Turn 5

Thought: If I now apply the share-a-genre filter, I need to make sure the surviving set is non-trivial... I also want to see whether one candidate dominates by vote.

Code:

```
target = ast.literal_eval(df.loc[top_idx, 'genre'])
def shares(v, t):
    if pd.isna(v): return False
    return any(g in t for g in ast.literal_eval(v))
cand = df[df['genre'].apply(lambda v: shares(v, target))
        & (df.index != top_idx)].dropna(subset=['vote'])
print('candidates:', len(cand))
print(cand.sort_values('vote', ascending=False)[['title','rating','vote']].to_string())
```

Observe:

candidates: 11

```
2 Superman 7.3 166692.0
23 Poldark 8.3 28102.0
14 Lady Jane 7.1 6438.0
```

. . . *(output truncated for brevity)*

Figure 14: The process output of Turn 5.

Execution Log — Turn 6

Thought: The chain produces 11 surviving candidates... I'll assemble the standalone program from pd.read_csv to print(answer) and submit.

Code:

```
submit_code(
     final_code = """
import pandas as pd, ast
df = pd.read_csv('/.../tbl_00130.csv')

# ... (Implementation omitted for brevity) ...

print(ranked.iloc[0]['title'])
""",
     answer = "Superman"
)
```

Observe:
Superman
✓ verified (server re-ran final_code in a fresh namespace; stdout matches claimed answer)

Figure 15: The process output of Turn 6.

Input / Task Context — Decompose & Rubric Stage

Verified Code Input (from Stage 1):

```
import pandas as pd, ast
df = pd.read_csv('/.../tbl_00130.csv')
top_idx = df['rating'].idxmax()
target = ast.literal_eval(df.loc[top_idx, 'genre'])
def shares(v, t):
     if pd.isna(v): return False
     return any(g in t for g in ast.literal_eval(v))
cand = df[df['genre'].apply(lambda v: shares(v, target))
         & (df.index != top_idx)].dropna(subset=['vote'])
ranked = cand.sort_values('vote', ascending=False)
print(ranked.iloc[0]['title'])
```

Target Skeleton:

retrieve (S1) → retrieve (S2) → filter (S3) → rank (S4) → locate (S5)

Figure 16: The input code and task description for the decomposition and rubric generation stage.

Execution Log — Turn 1 (Decompose)

Thought: The skeleton has five steps but the final_code runs them as one block. To write a rubric whose expected_value for each step is real (not guessed), I need to break the program into segments and re-execute them one segment at a time. I'll start with the two retrieve steps — the anchor row and its genre set — because those values feed every later step.

Code:

```
df = pd.read_csv(CSV)

# segment matching skeleton step S1: retrieve the anchor
top_idx = df.dropna(subset=['rating'])['rating'].idxmax()
print('S1 anchor row index :', top_idx)
print('S1 anchor title :', df.loc[top_idx, 'title'])
print('S1 anchor rating :', df.loc[top_idx, 'rating'])

# segment matching skeleton step S2: retrieve the anchor's genre set
top_genre = ast.literal_eval(df.loc[top_idx, 'genre'])
print('S2 anchor genre set :', top_genre)
```

Observe:

```
S1 anchor row index : 19
S1 anchor title     : The Shawshank Redemption
S1 anchor rating    : 9.3
S2 anchor genre set : ['Drama']
```

Figure 17: The process output of Turn 1 (Decompose).

Execution Log — Turn 2 (Decompose)

Thought: Next is the filter step. The rubric for S3 needs to record (a) how many candidates survive — this is the only number that quantifies whether the filter is doing real work — and (b) why the anchor row and rows with missing votes are excluded. I'll re-run only the filter segment and print both the count and the surviving rows.

Code:

```
def shares(v, t):
     return any(g in t for g in ast.literal_eval(v))

cand = df[df['genre'].apply(lambda v: shares(v, top_genre))
         & (df.index != top_idx)].dropna(subset=['vote'])

print('S3 candidate count :', len(cand))
print(cand.sort_values('vote', ascending=False)
                     .head(4)[['title','rating','vote','genre']].to_string())
```

Observe:

```
S3 candidate count : 11
   title      rating     vote      genre
2   Superman     7.3     166692.0   ['Action', 'Adventure', 'Drama', 'Sci-Fi']
23 Poldark      8.3      28102.0    ['Drama', 'History', 'Romance']
14 Lady Jane    7.1       6438.0    ['Biography', 'Drama', 'History', 'Romance']
22 Mrs Dalloway  6.7       3804.0    ['Drama', 'Romance']
```

Figure 18: The process output of Turn 2 (Decompose).

Execution Log — Turn 3 (Decompose)

Thought: The rank step (S4) sorts the 11 candidates by vote descending. For the rubric I need both the top entry and the runner-up — the runner-up is what makes the checkpoint discriminative, because a model that confuses "highest-rated" with "most-voted" within the candidate pool would land on Poldark (rating 8.3) instead of Superman (rating 7.3). The locate step (S5) then just reads the title of the top row.

Code:

```
ranked = cand.sort_values('vote', ascending=False)
print('S4 top :', ranked.iloc[0]['title'], '|', ranked.iloc[0]['vote'])
print('S4 runner :', ranked.iloc[1]['title'], '|', ranked.iloc[1]['vote'])
print('S5 answer :', ranked.iloc[0]['title'])
```

Observe:

```
S4 top     : Superman | 166692.0
S4 runner : Poldark   |   28102.0
S5 answer : Superman
```

Figure 19: The process output of Turn 3 (Decompose).

Execution Log — Turn 4 (Submit)

Thought: I now have a real value for every step in the chain — the anchor (Shawshank, 9.3), its genre set ({Drama}), the surviving candidate count (11), the top and runner-up by vote (Superman 166,692 vs Poldark 28,102), and the located title (Superman). I'll convert the program into a fluent question that does not mention column names or pandas operators, and write a 5-step rubric where each expected_value is taken verbatim from the executions above.

Code:

```
submit_result(
     question = "Which film in the table, excluding the highest-rated film itself,
                shares at least one genre with the highest-rated film and has
                received the greatest number of votes?",

     answer = "Superman",

     rubric = [
         { "step": 1,
              "checkpoint":  "Identify the anchor – the highest-rated film in the
table.",
              "reasoning_should_contain":  "Locate the row whose rating is the
maximum.",
             "expected_value": "The Shawshank Redemption (rating 9.3)" },

         { "step": 2,
              "checkpoint":  "Read the anchor's genre set; this set defines the
downstream filter.",
             "reasoning_should_contain": "Treat the anchor's genre field as a set of
tags, not a single string.",
             "expected_value": "{Drama}" },

         { "step": 3,
              "checkpoint": "Filter to films sharing ≥1 genre tag with the anchor;
exclude the anchor itself and rows with missing vote counts.",
              "reasoning_should_contain":  "Set-intersect each candidate's tag list
with the anchor's set; explain why the anchor and missing-vote rows are dropped.",
             "expected_value": "11 candidate films remain" },

         { "step": 4,
             "checkpoint": "Rank the surviving candidates by vote count, descending.",
              "reasoning_should_contain":  "Sort by votes, not by rating, since the
question asks for 'most-voted', not 'highest-rated'.",
             "expected_value": "top = Superman (166,692 votes); runner-up = Poldark
(28,102 votes)" },

         { "step": 5,
             "checkpoint": "Return the title of the top-ranked candidate.",
             "reasoning_should_contain": "Output the title field of the top row, not
its index, rating, or vote count.",
             "expected_value": "Superman" }
     ]
)
```

Observe:
✓ recorded

Figure 20: The process output of Turn 4 (Submit).

ORM Evaluation Prompt

You are an objective and rigorous data science objective-question evaluation expert. Your task is to determine whether the tested model's output response is substantively equivalent to the ground truth.

Core Evaluation Principles (Highest Priority):
1. Semantic Equivalence (Ignore natural language wrapping) - As long as the extracted core numerical values, entity lists, or boolean conclusions match the ground truth, the response must be judged as correct (score 1).
2. Tolerating Reasonable Formatting Differences - Numerical questions: 1.0 is equivalent to 1; 25,363 is equivalent to 25363. - List-based questions: Sets with different element orders but identical elements are considered equivalent; different delimiters (e.g., commas, semicolons, or slashes) are considered equivalent. - Boolean questions: "Yes" is equivalent to "True"; "No" is equivalent to "False" (case-insensitive).
3. Tolerating Reasonable Numerical Precision Differences - Differences caused by rounding or different decimal places are considered equivalent. - Numerical answers within a 5% relative error that clearly stem from precision/rounding differences should be judged as correct.
4. Error Handling - Deduct points only for substantive errors. Give a score of 0 only when the core conclusion is incorrect.

Output Format:
Please strictly return the following JSON format, without any additional text:
`{"reason": "brief explanation", "score": 1 or 0}`

Your Task:
Ground Truth: {ground_truth}
Tested Model Output: {response}
Evaluation Result (JSON):

Figure 21: The ORM evaluation prompt template.

Rubric-based Evaluation Prompt

You are an objective and rigorous data science evaluation expert. Your task is to evaluate the model's complete output based on the [Rubric].

Scoring Rules:
1. Mandatory Step-by-Step Evaluation - You must evaluate each step in the Rubric sequentially, regardless of whether the final answer is correct or incorrect.
2. Scoring Calculation - The final score is strictly determined by the steps completed. The score must be calculated as: `score = number of correct steps / total steps`.
3. Tolerating Reasonable Differences - Numerical: 1.0 is equivalent to 1; 25,363 is equivalent to 25363. - List: Sets with different element orders but identical elements are considered equivalent. - Boolean: "Yes" is equivalent to "True" (case-insensitive). - Numerical precision: A relative error within 5% is considered correct.

Output Format:
Please strictly return the following JSON format, without any additional text:
`{"reason": "brief explanation", "answer_correct": true/false, "steps": [true/false, ...], "score": 0.0~1.0}`

Where: - `answer_correct` indicates whether the final answer is correct. - `steps` is an array of booleans indicating whether each step in the Rubric was successfully completed. - `score` = number of correct steps / total steps

Your Task:
Ground Truth: {ground_truth}
Rubric: {rubric}
Model Output: {response}
Evaluation Result (JSON):

Figure 22: The rubric-based evaluation prompt template.

Manual Data Quality Inspection Protocol

1. Inspection Setup

- **Sample Size:** 200 instances randomly sampled *per synthesis method* (Self-instruction baseline and ProbeTQA).
- **Evaluators:** Recruited student volunteers, independent of the core algorithmic development.
- **Objective:** To quantify the **Human Solvability** of the generated data and conduct a rigorous quality assurance check.

2. Evaluation Criteria (Defining Human Solvability)
For each sampled instance, volunteers verified three dimensions. An instance is marked as having **Human Solvability** only if it satisfies all of the following:

- **Answerability:** The query is fully grounded and solvable based strictly on the provided table schema and data.
- **Logical Consistency:** The reasoning process or execution path is logically sound, without hallucinated operations.
- **Correctness:** The final synthesized answer is demonstrably correct.

3. Ethics and Compliance

- **Data Privacy:** The inspection focused entirely on public open-source datasets. No personal data, opinions, or behaviors were collected from the volunteers.
- **IRB Exemption:** This task constitutes factual verification of algorithmic outputs on public data, not human subjects research.

Figure 23: The formal protocol utilized by student volunteers to measure Human Solvability across different data synthesis methods.