DeepPrep: An LLM-Powered Agentic System for Autonomous Data Preparation

Meihao Fan
Renmin University of China
fmh1art@ruc.edu.cn

Ju Fan
Renmin University of China
fanj@ruc.edu.cn

Yuxin Zhang
Renmin University of China
yuxin.zhang@ruc.edu.cn

Shaolei Zhang
Renmin University of China
zhangshaolei98@ruc.edu.cn

Xiaoyong Du
Renmin University of China
duyong@ruc.edu.cn

Jie Song
ByteDance
jie.song@bytedance.com

Peng Li
ByteDance
peng.li01@bytedance.com

Fuxin Jiang
ByteDance
jiangfuxin@bytedance.com

Tieying Zhang
ByteDance
tieying.zhang@bytedance.com

Jianjun Chen
ByteDance
jianjun.chen@bytedance.com

ABSTRACT

Data preparation, which aims to transform heterogeneous and noisy raw tables into analysis-ready data, remains a major bottleneck in data science. Recent approaches leverage large language models (LLMs) to automate data preparation from natural language specifications. However, existing LLM-powered methods either make decisions without grounding in intermediate execution results, or rely on linear interaction processes that offer limited support for revising earlier decisions. To address these limitations, we propose DeepPrep, an LLM-powered agentic system for autonomous data preparation. DeepPrep constructs data preparation pipelines through iterative, execution-grounded interaction with an environment that materializes intermediate table states and returns runtime feedback. To overcome the limitations of linear interaction, DeepPrep organizes pipeline construction with tree-based agentic reasoning, enabling structured exploration and non-local revision based on execution feedback. To enable effective learning of such behaviors, we propose a progressive agentic training framework, together with data synthesis that supplies diverse and complex ADP tasks. Extensive experiments show that DeepPrep achieves data preparation accuracy comparable to strong closed-source models (*e.g.,* GPT-5) while incurring 15× lower inference cost, while establishing state-of-the-art performance among open-source baselines and generalizing effectively across diverse datasets.

PVLDB Reference Format:
Meihao Fan, Ju Fan, Yuxin Zhang, Shaolei Zhang, Xiaoyong Du, Jie Song, Peng Li, Fuxin Jiang, Tieying Zhang, and Jianjun Chen. DeepPrep: An LLM-Powered Agentic System for Autonomous Data Preparation. PVLDB, 19(11): 3371 - 3384, 2026.
doi:10.14778/3836663.3836695

†Work done during Meihao Fan's internship at ByteDance.
‡Ju Fan and Tieying Zhang are the corresponding authors.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836695

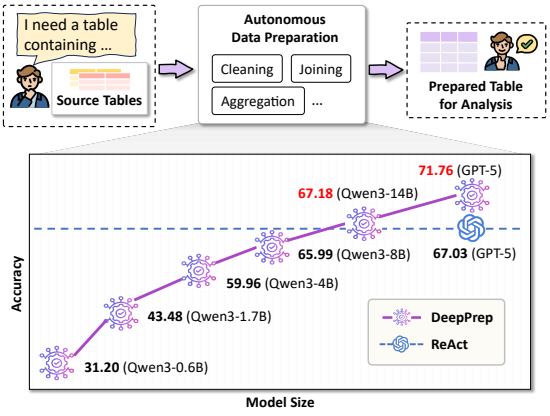

Figure 1: Overview of Autonomous Data Preparation (Top) and Performance of DeepPrep (Bottom).

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ruc-datalab/DeepPrep.

1 INTRODUCTION

Data preparation remains a major bottleneck in data science, often consuming 60%–80% of end-to-end analysis time [6]. This is due to the heterogeneity, noise, and misalignment across sources, as well as the iterative and interdependent nature of data preparation operations such as cleaning, joining, and aggregation. To address these challenges, autonomous data preparation (ADP) aims to automatically transform multiple source tables into analysis-ready data based on a high-level target specification, by generating an executable transformation pipeline. For example, as shown in Figure 1, given several source tables and a target description such as

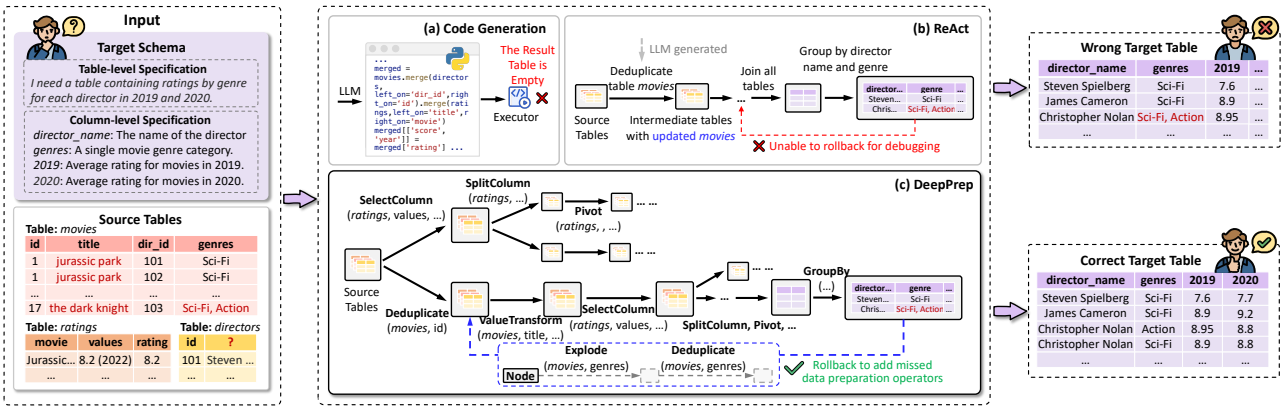

Figure 2: An illustrative example highlighting the core idea of DeepPrep. (a) Code generation makes decisions without grounding in intermediate execution results. (b) ReAct-style approaches follow linear reasoning with limited ability to revise earlier decisions. (c) DeepPrep applies a tree-based structured exploration, which enables backtracking to resolve non-local errors.

"*I need a table containing ...*", the system automatically generates and executes the required transformation steps.

State-of-the-Art Approaches. Existing approaches to autonomous data preparation broadly fall into two categories.

- Traditional methods rely on rigid specifications such as input–output examples, transformation templates, or rule-based program synthesis [3, 17, 18, 20, 36]. While effective in constrained settings, these approaches are not designed to flexibly accommodate data preparation intent expressed in natural language.
- Recent approaches leverage large language models (LLMs) to automate data preparation from natural language specifications [7, 8, 29, 31, 45]. Prompting-based methods generate data preparation code directly from task descriptions and examples [24, 28, 32], but make decisions without systematic grounding in intermediate execution results, often leading to error-prone pipelines. ReAct-style approaches incorporate execution feedback by interleaving reasoning with task-specific actions [2, 42]. However, this interaction follows a linear trajectory without explicit support for revising earlier decisions, which may be inadequate for autonomous data preparation, where early mistakes often become evident only after several dependent operations.

Our Proposal. To address these limitations, we propose **DeepPrep**, an LLM-powered agentic system for autonomous data preparation. DeepPrep constructs data preparation pipelines through iterative, execution-grounded interaction with an environment that materializes intermediate table states and returns runtime feedback. Unlike one-shot code generation and linear ReAct-style interaction, DeepPrep organizes pipeline construction with *tree-based agentic reasoning*. This design enables structured exploration and *non-local* revision when execution feedback indicates that earlier decisions are suboptimal. Specifically, given source tables and a target schema, DeepPrep operates in an iterative loop as below:

- **Planning:** observing the current execution state and synthesizing a high-level plan for transforming existing tables.
- **Orchestration:** instantiating the plan as an executable operator pipeline with specified parameters.
- **Execution:** executing the pipeline to materialize new table states and returning runtime feedback for subsequent decisions.

Through iterative interaction with execution feedback, DeepPrep incrementally refines its decisions and converges to a data preparation pipeline that produces a table conforming to the target schema.

Example 1. *Consider the example in Figure 2. The goal is to construct a table for analyzing director performance across genres using audience ratings. A linear agent following a ReAct-style interaction may prematurely join and aggregate the data, resulting in multi-valued entries such as "Sci-Fi, Action" in the genre column, which violates the target schema constraint requiring single-valued categories. Correcting this error requires revisiting an earlier cleaning decision on the movies table,* i.e., *inserting an* Explode *operation before reapplying downstream joins and aggregations.*

DeepPrep supports this naturally. It can backtrack to the execution state where the incorrect decision was made, expand an alternative branch with the corrected operation, and reuse valid operator prefixes from existing partial pipelines to produce a correct result.

Challenges and Solutions. We study the research challenges posed by the above agentic framework and outline our solutions.

The first challenge is how to support structured exploration and non-local backtracking under execution feedback. To address this, DeepPrep maintains an explicit tree of materialized execution states during pipeline construction, where nodes represent intermediate tables produced by executed operators and edges record the applied operator pipelines. This representation enables execution feedback to be precisely attributed to earlier decision points, allowing the agent to backtrack and revise upstream choices when downstream failures occur. Building on this structure, DeepPrep introduces action-structured LLM interactions that operate directly over the tree, enabling the agent to incrementally expand, revise, or backtrack branches based on execution feedback.

The second challenge is how to effectively train LLMs for the tree-based agentic reasoning. A straightforward solution is to apply reinforcement learning with outcome-level rewards [15, 22]. However, this setting introduces two difficulties. First, *reward sparsity*: long and interdependent pipelines make outcome insufficient for learning structured exploration and non-local backtracking. Second, *data scarcity*: existing ADP datasets focus on simple, short

pipelines [21, 27], limiting supervision for multi-step, execution-aware reasoning. To address these challenges, we propose a Progressive Agentic Training framework that (i) initializes the model with basic operator usage and tree-based interaction patterns, (ii) refines the policy through multi-turn reinforcement learning with a hybrid reward that considers both outcome-level and process-level signals, and (iii) leverages data synthesis to construct diverse and complex ADP tasks for training.

Contributions. We summarize our contributions as follows.
(1) We study the problem of autonomous data preparation with support for a wide range of data preparation operators (Section 2), and introduce an LLM-powered agentic framework (Section 3).
(2) We propose two techniques to tackle the challenges of this problem: tree-based agentic reasoning for inference (Section 4) and progressive agentic training for agent optimization (Section 5).
(3) We conduct extensive experiments on real-world ADP benchmarks. Results show that DeepPrep achieves accuracy comparable to closed-source models (*e.g.,* GPT-5) at 15× lower inference cost, while establishing state-of-the-art performance among open-source baselines and generalizing well across diverse datasets.
(4) We release the code, data, and a comprehensive set of trained model weights (from 0.5B to 14B parameters), enabling users to deploy ADP agents under different computational budgets.

2 PRELIMINARIES

2.1 Problem Formulation

Source and Target Data. We consider a set of *source tables* $\mathcal{S} = \{S_1, \ldots, S_n\}$. Each source table is denoted as $S_i = (\Sigma_i, D_i)$, where Σ_i denotes the schema of S_i and D_i denotes the set of tuples conforming to Σ_i. The schema Σ_i captures both table-level and column-level specifications and is defined as $\Sigma_i = (\tau_i, C_i)$. Here, τ_i represents a textual description that summarizes the semantic meaning of the table, and $C_i = \{c_i^1, \ldots, c_i^{m_i}\}$ denotes the set of column specifications. Each column specification c_i^j includes column-level metadata such as column name, data type, and semantic description. In practice, the schema Σ_i of a source table may be partially specified or entirely unavailable. In such cases, table- and column-level specifications can be inferred from the raw data in D_i. For example, each source table S_i in Figure 2 consists of a schema Σ_i which may be specified by table- and column-level information, and a set D_i of data tuples, *e.g.,* (1,"jurassic park", 101, "Sci-Fi").

Similarly, the *target table* is denoted as $T = (\Sigma^*, D^*)$, where $\Sigma^* = (\tau^*, C^*)$ specifies the target schema and D^* denotes the target tuples to be produced. Note that, in autonomous data preparation, only the target schema Σ^* is given as input, while the target tuples D^* are unknown and must be generated by transforming and cleaning the source data. For example, in Figure 2, the target schema Σ^* is specified by a table-level description (*e.g.,* "*I need a table containing ...*") and a set of column specifications, each including a column name (*e.g.,* "*director_name*") and a corresponding semantic description (*e.g.,* "*The name of the director.*").

Data Preparation Pipeline. We model the process of data preparation as a composition of *operators* applied to source tables. Specifically, an operator represents a primitive data transformation. Formally, an operator o is characterized by its type κ and parameters θ, and defines a function $o_\theta : \mathcal{T} \to \mathcal{T}$, where $\mathcal{T}$ denotes a set of tables, and each table is of the form (Σ, D) as defined previously. Given an input table set $\mathcal{T}_{\text{in}}$, the operator produces an output table set, *i.e.,* $\mathcal{T}_{\text{out}} = o_\theta(\mathcal{T}_{\text{in}})$. Note that different operator types capture different classes of data transformations, such as filtering, joining, aggregation, and cleaning. The current operator types supported by DeepPrep will be introduced in Section 2.2.

We then define a data preparation pipeline (or pipeline for short) as an ordered sequence of operators applied iteratively to transform a set of source tables $\mathcal{S}$ into a target table $\hat{T}$. We use the symbol "$\to$" to denote the left-to-right application order between adjacent operators, *i.e.,* the output produced by an operator is fed as the input to the operator to its right. Formally, a pipeline $\mathcal{P}$ is defined as an ordered composition of k operators over $\mathcal{S}$, *i.e.,*

$$\mathcal{P} = (\mathcal{S}, o_{\theta_1}^{(1)} \to o_{\theta_2}^{(2)} \to \cdots \to o_{\theta_k}^{(k)}). \tag{1}$$

Given a set of source tables $\mathcal{S}$, the execution of pipeline $\mathcal{P}$ produces a sequence of intermediate table sets, *i.e.,* $\mathcal{T}_0 = \mathcal{S}$, $\mathcal{T}_i = o_{\theta_i}^{(i)}(\mathcal{T}_{i-1}), i = 1, \ldots, k$, where $\mathcal{T}_k$ denotes the final output of the pipeline, which contains only the target table $\hat{T}$, *i.e.,* $\mathcal{T}_k = \{\hat{T}\}$. Consider operator $\mathrm{Deduplicate}\ (\mathrm{movies},\ [\mathrm{id}],\ \mathrm{first})$ as an example. Its operator type κ is Deduplicate and its parameters θ refer to $(\mathrm{movies},\ [\mathrm{id}],\ \mathrm{first})$. Executing this operator on source tables produces intermediate tables consisting of a cleaned *movie*, while tables *ratings* and *directors* remain unchanged.

Problem Statement. Given a set of source tables $\mathcal{S} = \{S_1, \ldots, S_n\}$, a target schema Σ^*, and a set of available operator types O, the autonomous data preparation (ADP) problem is to find an optimal data preparation pipeline $\mathcal{P}$ that transforms $\mathcal{S}$ into a target table $\hat{T} = (\Sigma^*, \hat{D})$, where $\hat{D}$ denotes the tuples produced by $\mathcal{P}$ that satisfy the target schema Σ^*. We assume a ground-truth target table $T^* = (\Sigma^*, D^*)$, which is used *only* for evaluation. Thus, the optimality of a pipeline is defined by the quality of $\hat{D}$ with respect to D^* under task-specific evaluation metrics, which will be introduced in the experimental setup (Section 6.1).

Example 2. *Figure 2 provides an example of an ADP task with three source tables: movies, ratings, and directors. The goal is to generate a target table that conforms to a given target schema. The target schema describes both the overall semantics of the table and its desired structure. Specifically, it provides a table-level summary of the intended content and specifies column-level requirements, including column names* (e.g., *genre), semantic meanings* (e.g., *movie genre category), and data constraints* (e.g., *single-valued values). To achieve this goal, we construct a data preparation pipeline composed of multiple operators, such as* Deduplicate *and* Join. *Each operator is applied to the current set of tables and produces updated intermediate tables. For example, applying* Join *merges the movies and directors tables on movie titles, yielding a table named movies_directors_join that integrates movie data with director names, while keeping the ratings table unchanged. After executing all operators in the pipeline, the process produces the target table, shown on the right of Figure 2.*

2.2 Supported Data Preparation Operators

DeepPrep supports a broad set of data preparation operators and is designed to be extensible, allowing new operator types to be incorporated as needed. In total, we consider 31 operators, organized into eight categories that cover the major transformation patterns in data preparation pipelines. Due to space constraints, we provide descriptions of **typical operators** in each category, with complete specifications available in [1].

2.2.1 Data Cleaning. Data cleaning operators improve data quality by detecting, fixing, or removing erroneous, missing, or duplicate records, and are commonly applied in data preparation pipelines.

- MissingValueImputation(table, column, mode) imputes missing values in a specified column using a statistical strategy, with mode selecting the method (*e.g.,* mean, median, or mode).
- Deduplicate(table, subset, keep) removes duplicate rows based on specified columns, retaining either the first or last occurrence as determined by keep.

2.2.2 Value Normalization. Value normalization operators standardize field values and data types to ensure semantic consistency.

- ValueTransform(table, column, func) applies a user-defined transformation function to values in a specified column to standardize their format.
- StandardizeDatetime(table, column, format) converts values in a specified column to a user-defined datetime format.

2.2.3 Schema Editing. Schema editing operators modify table structure and column semantics, enabling column selection, renaming, derivation, and restructuring to align raw data with the target schema.

- RenameColumn(table, rename_map) renames columns according to a specified mapping from original to target names.
- AddNewColumn(table, name, func) adds a new column computed from existing data using a row-wise function.

2.2.4 Row Selection. Row selection operators manipulate tables by filtering, ordering, or selecting subsets of rows.

- Filter(table, func) selects rows satisfying a given predicate.
- Sort(table, by, ascending) orders rows in a table based on specified columns, with ascending controlling the direction.
- TopK(table, k) retains the top-k rows of the table.

2.2.5 Aggregation. Aggregation operators summarize data by grouping records and computing statistics, transforming row-level data into compact, analysis-ready representations.

- GroupBy(table, by, agg) groups rows by specified columns and applies aggregation functions defined by agg.
- Count(table) returns the total number of rows in the table. We treat Count as a primitive aggregation operator for convenience, though it can be viewed as a special case of GroupBy without grouping keys.

2.2.6 Table Combination. Table combination operators integrate data from multiple tables into a unified representation, enabling multi-source data preparation pipelines.

- Join(left, right, on, how) combines two tables based on specified key columns using a join type defined by how.

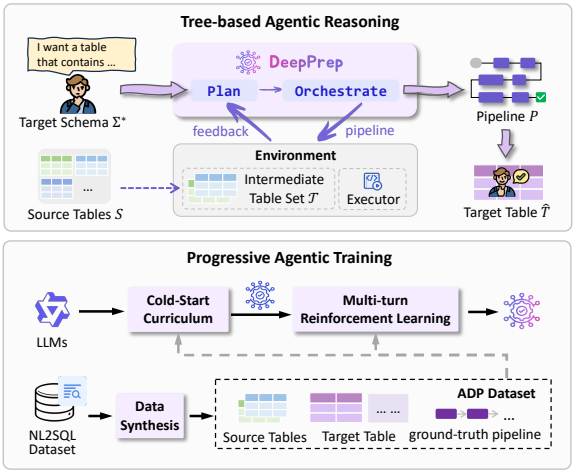

Figure 3: An overview of DeepPrep.

- Union(tables, how) vertically combines multiple tables with compatible schemas, with how determining whether duplicate rows are retained or removed.

2.2.7 Table Reshaping. Table reshaping operators reorganize the structural layout of tables, such as converting between wide and long formats or expanding nested fields.

- Pivot(table, index, columns, values, aggfunc) reshapes a table from long to wide format by aggregating values at the intersection of specified index and column identifiers.
- Stack(table, id_vars, value_vars) reshapes a table to long format by collapsing multiple columns into key–value pairs.

2.2.8 Program Synthesis. The program synthesis operator provides a general mechanism for handling transformations that cannot be expressed using the above predefined operators.

- ExeCode(tables, target, func) generates a target table from input tables by executing Python code synthesized by an LLM, enabling transformations that cannot be expressed using the above predefined operators.

3 AN OVERVIEW OF DEEPPREP

DeepPrep employs an agentic framework to autonomously orchestrate data preparation pipelines, as illustrated in Figure 3. Given a set of source tables $\mathcal{S}$ and a target schema Σ^*, DeepPrep iteratively interacts with an *environment* that executes data transformations and provides feedback. The environment maintains an execution state represented by a set of intermediate tables, initialized with the source tables $\mathcal{S}$. Operator instances are executed within the environment through concrete execution backends (*e.g.,* a Python interpreter), which apply transformations to the current state and produce updated tables along with execution feedback. Given the target schema Σ^*, DeepPrep interacts with the environment through an iterative process that alternates among the *planning*, *orchestration*, and *execution* steps, as described below.

- In the *planning* step, the agent observes the current execution state and synthesizes a high-level textual plan that outlines how to transform the existing tables toward the target schema.

- In the *orchestration* step, DeepPrep translates this plan into an executable pipeline by selecting appropriate operator types and instantiating them with specific parameters.
- In the *execution* step, the orchestrated pipeline is submitted to the environment, which executes the operators, updates its internal state with newly generated intermediate tables, and returns execution feedback, such as sample outputs or error traces.

Through iterative interaction with execution feedback, DeepPrep progressively refines planning and orchestration decisions and converges to a pipeline that produces a desired target table.

Example 3. *Consider the ADP task shown in Figure 2. The environment is initialized with the source tables and target schema.*

- *To align the data with the target schema, the agent first generates a high-level textual plan, such as "Clean table movies...".*
- *Based on this plan, the orchestration step instantiates a pipeline,* i.e., *a sequence of operators,* e.g., Deduplicate(...) *followed by* ValueTransform(...). *The orchestrated operators are then submitted to the environment for execution.*
- *Upon execution, the environment applies the operators to the current state and returns execution feedback. Specifically, if execution is successful, the environment updates its state to include the cleaned movies table while retaining the original ratings and directors tables. Otherwise, execution errors are captured and propagated to the agent to revise subsequent decisions.*

The iterative process continues until the generated tables satisfy the target schema and quality requirements, at which point DeepPrep terminates and outputs the final data preparation pipeline.

To support the framework, we design two key components: *Tree-Based Agentic Reasoning* and *Progressive Agentic Training*.

Tree-based Agentic Reasoning. The key challenge in agentic pipeline orchestration is supporting *structured* exploration and backtracking under execution feedback. Specifically, pipeline construction involves interdependent operator decisions, where early choices may later prove suboptimal or infeasible. Linear agentic reasoning mechanisms such as ReAct [2] and CoT [38] follow a single decision trajectory and do not preserve alternative paths. Thus, once a candidate pipeline is executed, early errors can propagate to subsequent steps, and without a structured representation that retains historical states, recovery becomes difficult.

To address this challenge, we introduce a *tree-based agentic reasoning* mechanism. Instead of committing to a single linear execution trajectory, DeepPrep represents pipeline construction as a search tree, where each node corresponds to a planning state associated with a set of intermediate tables, and each edge represents applying a candidate operator pipeline. This tree structure preserves historical states and alternative construction paths, enabling systematic exploration and backtracking based on execution feedback. Building on this tree representation, we further define LLM interaction patterns and decision logic over the tree, allowing the agent to reason about which branch to expand, revise, or abandon during pipeline construction. We detail this tree-based reasoning and interaction mechanism in Section 4.

Progressive Agentic Training. The key challenge is how to effectively train LLMs for tree-based agentic reasoning. A straightforward approach is to directly apply existing reinforcement learning

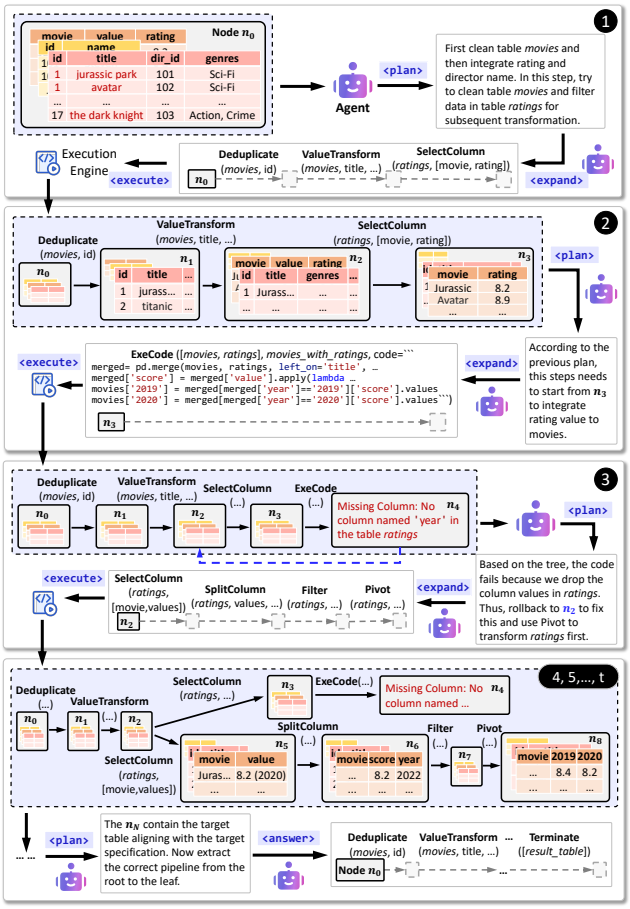

Figure 4: An example of tree-based agentic reasoning.

methods to optimize the model using outcome-level rewards. However, this setting poses two major difficulties. First, *reward sparsity*: data preparation pipelines often consist of long and interdependent operator sequences, and relying solely on final correctness makes it difficult for the model to learn structured exploration and non-local backtracking behaviors. Second, *data scarcity*: existing ADP datasets mainly involve simple transformations with short operator sequences [21, 27], providing insufficient supervision for learning multi-step reasoning that depends on execution feedback.

To address reward sparsity, we first introduce a Cold-Start Curriculum that initializes the model with basic knowledge of data preparation operators and the tree-based agentic reasoning procedure. Next, we apply multi-turn reinforcement learning with a Hybrid Reward mechanism that combines outcome-level signals with process-level feedback. To alleviate data scarcity, we develop a data synthesis approach that generates complex ADP tasks grounded in meaningful analytical transformations, while injecting diverse data quality issues without breaking the intended transformation logic. We detail our agentic training strategy in Section 5.

4 TREE-BASED AGENTIC REASONING

To construct data preparation pipelines under complex dependencies and execution feedback, DeepPrep employs a *tree-based agentic reasoning* mechanism that supports structured exploration and

non-local backtracking. Specifically, we introduce an *agentic reasoning tree* (Section 4.1) that explicitly preserves historical states and alternative data preparation pipelines. Building on this tree structure, we define a tree-based agentic inference procedure (Section 4.2) through a set of explicit agentic interaction actions. These actions specify how the LLM observes the current tree state, proposes data preparation plans, and generates promising pipelines.

4.1 Agentic Reasoning Tree

We define the *agentic reasoning tree* as a directed tree $G = (N, E)$, which represents the structured search space of pipeline construction starting from the initial source tables $\mathcal{S}$.

- **Node (N).** Each node $n \in N$ corresponds to an *environment state*, represented by the current set of intermediate tables maintained by the environment. In particular, the root node n_0 represents the initial state initialized with the source tables $\mathcal{S}$.
- **Edge (E).** A directed edge $e = (n_i \xrightarrow{o} n_j) \in E$ represents a state transition induced by executing an *operator instance* o (*i.e.,* an operator type with specific parameters) orchestrated by the agent at state n_i. Specifically, executing o updates the environment state from n_i to n_j, yielding updated intermediate tables.

In the agentic reasoning tree, each edge corresponds to a single operator instance, while a path from a node to another node represents an operator sequence, *i.e.,* a data preparation pipeline.

4.2 Tree-Based Agentic Inference

The basic idea of our tree-based agentic inference is to develop an iterative decision-making process through a structured LLM interaction pattern. Specifically, the LLM autonomously generates agentic *interaction actions* to operate over a materialized agentic reasoning tree, incrementally expanding, revising, or backtracking tree branches based on execution feedback.

Agentic Inference Framework. Given source tables $\mathcal{S}$ and a target schema Σ^*, the framework produces a reasoning trajectory R and finally generates an answer A based on R. More formally, we model this process by a probabilistic model $\Pr(\cdot)$ over the reasoning process and the answer generation conditioned on the inputs:

$$\Pr(A, R \mid S, \Sigma^*) = \underbrace{\prod_{t=1}^{T} \Pr(R_t \mid R_{<t}, S, \Sigma^*)}_{\text{Reasoning Process}} \cdot \underbrace{\Pr(A \mid R, S, \Sigma^*)}_{\text{Answer Generation}}. \quad (2)$$

where T denotes the length of the reasoning process and R_t represents the reasoning results of the agent at step t.

The reasoning process is implemented by adopting an LLM-based agent to search over the agentic reasoning tree G in a generative way. Specifically, the process alternates in an iteration of *plan*, *expand* and *execute* steps where it generates logical plans for the current step based on the observation of the current tree, expands new branches and execute expanded operators to update the current tree. Based on this idea, we break the step-wise reasoning process in Eq. (2) down into the following formula.

$$\Pr(R_t|R_{<t}, S, \Sigma^*) = \underbrace{\Pr_{\text{LLM}}(\psi_t|\psi_{<t}, G_{t-1}, \Sigma^*)}_{\langle\text{plan}\rangle} \cdot \underbrace{\Pr_{\text{LLM}}(\mathcal{P}_t|G_{t-1}, \psi_t)}_{\langle\text{expand}\rangle} \cdot \underbrace{\text{Env}(G_t|G_{t-1}, \mathcal{P}_t)}_{\langle\text{execute}\rangle}. \quad (3)$$

where $\Pr_{\text{LLM}}$ denotes the generative distribution induced by the LLM agent, ψ_t and $\mathcal{P}_t$ denotes the logical plan and expanded operator pipeline generated by the agent in step t, and Env represents the environment transition distribution that materializes $\mathcal{P}_t$ to get an updated agentic reasoning tree G_t.

Actions for Agentic Inference. We regulate this interaction using control tags aligned with the components in Eq. (2) and Eq. (3).

- $\langle\text{plan}\rangle$. This action produces a high-level logical plan for the next reasoning step. Given the current agentic reasoning tree, the target schema Σ^*, and prior plans, the agent generates a plan ψ that specifies the intended reasoning decision. The plan analyzes the current environment state to determine whether to continue expanding the tree or terminate the reasoning process. If further exploration is required, it identifies the parent node to extend and the candidate operations to apply.
- $\langle\text{expand}\rangle$. This action generates a candidate operator pipeline expanded from a selected parent node n_{parent}. Conditioned on the plan ψ, the agent produces an operator sequence $\mathcal{P}$ to extend node n_{parent}. To ensure consistent node selection, the parent node is specified using a prefix-matching constraint: the agent must reference the full operator sequence from the root to that node. This constraint avoids ambiguous or index-based node references and enforces alignment between the proposed expansion and the recorded transformation history.
- $\langle\text{execute}\rangle$. This action applies the candidate operator pipeline through the environment and performs state transitions. Specifically, given a candidate pipeline $\mathcal{P}_t$, the environment executes its operators sequentially over the materialized tables at the selected parent node. After each operator is successfully applied, a new intermediate state node is created to store the resulting tables, and a directed edge is added from the previous state node to the new one. Repeating this process for all operators in $\mathcal{P}_t$ yields a chain of new nodes, which updates G_{t-1} to G_t. If execution fails due to a runtime exception (*e.g.,* column mismatch or type error), no new state node is created; instead, the parent node is annotated with a failure state that records the error trace.
- $\langle\text{answer}\rangle$. This action terminates the reasoning process and returns the final pipeline. It is triggered when a leaf node state satisfies the target schema Σ^*. The root-to-leaf operator path associated with that node is extracted as the final pipeline $\mathcal{P}^*$.

Agentic Inference Procedure. We extend the LLM vocabulary with the action tags defined above, so that our tree-based reasoning can be expressed as a structured language modeling process.

During inference, the model iteratively generates tagged outputs corresponding to the defined interaction actions. Specifically, in each iteration, the model first generates a plan in the form of $\langle\text{plan}\rangle$...$\langle/\text{plan}\rangle$, which specifies the next reasoning decision. Based on this plan, the model predicts an operator expansion wrapped by $\langle\text{expand}\rangle$...$\langle/\text{expand}\rangle$. The produced pipeline is then executed by

the environment, and the resulting state and execution feedback are recorded through ⟨execute⟩...⟨/execute⟩. The above process iterates until a state satisfies the target schema. When this condition holds, the final pipeline is returned using ⟨answer⟩...⟨/answer⟩, which terminates the reasoning process.

Complexity Analysis. The search process is bounded by two configurable parameters: the maximum exploration turn T and the maximum operator sequence length L per expansion. Therefore, the worst-case search complexity is $O(T \times L)$.

Example 4. *Figure 4 illustrates an example reasoning trajectory for the task in Figure 2.* ① ***Initialization:*** *Reasoning starts from node n_0, which corresponds to an initial state associated with source tables $\mathcal{S}$. During inference, the model generates a* ⟨plan⟩ *action, specifying a logical plan that cleans the movies table and select relevant columns in the ratings table. An* ⟨expand⟩ *action then produces operators* Deduplicate, ValueTransform *and* SelectColumn, *which are applied through* ⟨execute⟩ *to produce new states,* i.e., *n_1, n_2 and n_3.* ② ***Execution Failure:*** *Subsequent* ⟨plan⟩ *and* ⟨expand⟩ *actions proposes integrating rating data using an* ExeCode *operator. However, during* ⟨execute⟩*, a missing-column runtime error is raised, and the transition is recorded as a failure at n_4.* ③ ***Backtracking and Alternative Expansion:*** *The model preserves the failure at n_4 and append the error trace to subsequent prompts. Based on the recorded failure feedback, a new* ⟨plan⟩ *analyzes the error and tries to debug it by keeping the column values of ratings table when applying the operator* SelectColumn. *Specifically, it selects n_2 as the parent state for further exploration. In addition to applying the updated* SelectColumn, *the* ⟨expand⟩ *action also proposes an alternative operator sequence (*SplitColumn, Filter, Pivot*). By referencing the failure at n_4, the model avoids repeating the same mistake when generating this new branch. These operators are applied to generate new states (n_5, n_6, n_7, n_8).* ⓣ ***Termination:*** *When a leaf state satisfies the target schema Σ^*, an* ⟨answer⟩ *action is issued to extract the corresponding root-to-leaf operator path as the final pipeline.*

5 AGENTIC TRAINING FRAMEWORK

We propose a *Progressive Agentic Training* framework for optimization. To mitigate reward sparsity, the framework includes a *Cold-Start Curriculum* for initialization (Section 5.1), followed by a *Multi-turn RL* stage with a *Hybrid Reward* mechanism (Section 5.2). To alleviate data scarcity, we develop a *Data Synthesis* method (Section 5.3) that generates diverse ADP tasks for effective training.

5.1 Cold-Start Curriculum

To mitigate the issue where an untrained LLM fails to produce executable pipelines, we design a Cold-Start Curriculum that initializes the LLM's reasoning capability for our structured and execution-grounded planning. Specifically, pre-trained LLMs generally lack an explicit understanding of operator syntax and the tree-based agentic reasoning procedure, which are previously described. Thus, without specialized supervision, the model cannot generate meaningful action sequences for ADP tasks. To address this, we employ a two-stage supervised fine-tuning (SFT) curriculum.

Stage 1: Operator Syntax Learning. This stage focuses on the syntax and semantics of individual operators. Given source tables and a ground-truth operator pipeline $\mathcal{P}_{\text{total}}$ as defined in Eq. (1), consisting of a sequence of operators $(o^{(1)}, \ldots, o^{(N)})$. We first execute these operators sequentially to collect a sequence of generated intermediate table sets $\{\mathcal{T}_0, \ldots, \mathcal{T}_N\}$. Then, we sample a contiguous sub-sequence of operators $\mathcal{P}_{\text{sub}} = (o^{(i)}, \ldots, o^{(j)})$ with the corresponding input intermediate table set $\mathcal{T}_{\text{in}} = \mathcal{T}_{i-1}$ and output intermediate table set $\mathcal{T}_{\text{out}} = \mathcal{T}_j$. In this way, we construct the operator syntax learning dataset $\mathcal{D}_{\text{op}}$ consisting of training triples $(\mathcal{P}_{\text{sub}}, \mathcal{T}_{\text{in}}, \mathcal{T}_{\text{out}})$. Let $\phi(\cdot)$ denote the serialization function. It serializes $\mathcal{P}_{\text{sub}}$ by concatenating all serialized operators and intermediate table set $\mathcal{T}$ by serializing it into markdown format. We conduct SFT to optimize the following loss $\mathcal{L}_{\text{op}}$:

$$\mathcal{L}_{\text{op}} = - \sum_{(\mathcal{P}_{\text{sub}}, \mathcal{T}_{\text{in}}, \mathcal{T}_{\text{out}}) \in \mathcal{D}_{\text{op}}} \log \Pr_{\text{LLM}}(\phi(\mathcal{P}_{\text{sub}}) \mid \phi(\mathcal{T}_{\text{in}}), \phi(\mathcal{T}_{\text{out}})). \tag{4}$$

Through Eq. (4), the agent learns to recognize operator function signatures and argument formats, while isolating operator execution logic from higher-level planning decisions.

Stage 2: Reasoning Procedure Learning. This stage aligns the model with the *Tree-based Agentic Reasoning* mechanism (Section 4.2). To achieve this, we synthesize a dataset of high-quality reasoning trajectories by distilling knowledge from strong teacher models (*e.g.,* DeepSeek-R1). Specifically, for each ADP task defined by source tables $\mathcal{S}$ and a target schema Σ^*, we aim to generate a ground-truth trajectory τ. To enhance robustness, we apply input perturbations Φ, such as randomly shuffling rows and columns in the source tables. Next, we employ In-Context Learning [9] to guide the teacher model. By utilizing elaborate prompts that exemplify the tree-based reasoning process defined in Eq. (2), we generate a set of candidate trajectories $\mathcal{U}$. To unify the training objective, we represent each trajectory as a sequence $\tau = (r_1, \ldots, r_m)$. Here, the initial subsequence $(r_1, \ldots, r_{m-1})$ corresponds to the iterative reasoning steps $(R_1, \ldots, R_T)$ defined in Eq. (3), while the final element r_m corresponds to the answer generation A in Eq. (2). These candidates are filtered using a process reward function $R_{\text{llm}}(\tau)$ (detailed in Section 5.2), and only the top-ranked trajectories are retained to form the final dataset $\mathcal{U}^*$. Finally, the model is fine-tuned on $\mathcal{U}^*$ to minimize the masked negative log-likelihood:

$$\mathcal{L}_{\text{reason}} = - \sum_{\tau \in \mathcal{U}^*} \sum_{t=1}^{m} M_t \cdot \log \Pr_{\text{LLM}}(\phi(r_t) \mid \phi(r_{<t}), \phi(\mathcal{S}), \Sigma^*). \tag{5}$$

Here, r_t encapsulates both the current tree state and the agent's generated response. The serialization function $\phi(\cdot)$ transforms these structured components into a linear token sequence. Crucially, we introduce a loss mask M_t to ensure that gradients are only calculated for tokens corresponding to the agent's outputs.

5.2 Multi-turn RL with Hybrid Reward

Motivation. Data preparation pipeline generation is a sequential decision process with execution feedback, where each action affects subsequent states and future operator choices. Optimizing such behavior cannot be reduced to single-shot generation. Instead, the policy should be trained over multi-turn interaction trajectories. In each turn, the agent produces structured actions (*e.g.,* ⟨plan⟩ and ⟨expand⟩), the environment executes the proposed operators,

and feedback is returned. Then, subsequent decisions are conditioned on these execution outcomes.

Basic Idea. The above setting motivates us to develop a multi-turn reinforcement learning method, where the policy is optimized over action sequences rather than single outputs. Specifically, under our interaction framework, each trajectory contains both (i) agent-generated tokens (actions and their contents) and (ii) environment-generated tokens, such as serialized intermediate tables and execution traces. In particular, since environment-generated tokens are deterministic execution outputs rather than policy decisions, they are excluded from policy gradient optimization. That is, we apply a binary token mask that removes gradients on tokens inside ⟨execute⟩ blocks, and optimize the policy only over agent decision tokens within ⟨plan⟩, ⟨expand⟩, and ⟨answer⟩.

We optimize the policy using Group Relative Policy Optimization (GRPO) [34], which estimates baselines from group-level reward statistics, supporting more stable optimization over multi-turn trajectories. Using only final pipeline correctness as the reward, however, results in sparse and delayed learning signals. We therefore introduce a hybrid reward design, as described below.

Hybrid Reward. We design a hybrid reward mechanism that provides denser learning signals by evaluating both outcome quality and reasoning behavior. The overall reward $R(\tau)$ for a trajectory τ is defined as a weighted sum of three components:

$$R(\tau) = \alpha \cdot R_{\text{out}}(\tau) + \beta \cdot R_{\text{part}}(\tau) + \gamma \cdot R_{\text{llm}}(\tau). \tag{6}$$

where R_{out}, R_{part}, and R_{llm} correspond to strict correctness, partial progress, and reasoning process signals, respectively, and α, β, γ are hyperparameters that balance their contributions. Now, we detail each of the three components R_{out}, R_{part}, and R_{llm} as follows.

Outcome Reward R_{out}. This reward provides a direct success signal based on final output correctness. When the ⟨answer⟩ action is issued, the generated resulting target table $\hat{T}$ is evaluated by:

$$R_{\text{out}}(\tau) = \mathbb{I}(\hat{T} \equiv T^*). \tag{7}$$

where $\mathbb{I}(\cdot)$ denotes the indicator function.

Partial Reward R_{part}. This reward provides a partial-progress signal when strict correctness is not achieved ($R_{\text{out}}(\tau) = 0$). We measure the similarity between the generated table $\hat{T}$ and ground-truth T^*, and define $R_{\text{part}}(\tau) = \text{avg}(S_{sch}, S_{shp}, S_{cnt})$, where $S_{sch}, S_{shp}, S_{cnt}$ evaluate alignment in schema, shape and content, respectively:

$$\begin{aligned} S_{sch} &= \frac{|C_{\hat{T}} \cap C_{T^*}|}{|C_{\hat{T}} \cup C_{T^*}|}, \quad S_{shp} = \exp\left(-\left|\frac{|\hat{D}| - |D^*|}{|D^*|}\right|\right), \\ S_{cnt} &= \frac{1}{|C_m|} \sum_{c \in C_m} \frac{\sum_i \mathbb{I}(\hat{D}[c]_i = D^*[c]_i)}{\max(|\hat{D}|, |D^*|)}. \end{aligned} \tag{8}$$

Here, $C_{\hat{T}}$ and C_{T^*} denote the column-name sets, $|\hat{D}|$ and $|D^*|$ denote the row counts, and $C_m = C_{\hat{T}} \cap C_{T^*}$ is the set of matched columns used for cell-wise value comparison.

Process Reward R_{llm}. Relying solely on data similarity rewards may lead to *reward hacking*, which has been studied in other reinforcement learning settings [37, 46], and can also arise in ADP scenarios (*e.g.,* renaming columns without performing the required data cleaning). To discourage such "shortcut" strategies, we introduce a process reward that evaluates the quality of the reasoning trajectory. We use an instruction-tuned LLM (*e.g.,* GPT-4o) as a judge to score each trajectory based on the actions defined in Section 4.

Specifically, the judge evaluates three criteria: (1) *Plan–action consistency*, measuring whether generated pipeline (from ⟨expand⟩) correctly implements the plan (from ⟨plan⟩). (2) *Feedback responsiveness*, checking whether subsequent ⟨plan⟩ outputs explicitly respond to the prior execution feedback (*e.g.,* reported error traces). (3) *Backtracking justification*, verifying that parent-node switches are supported by recorded failure evidence from the current branch.

5.3 Data Synthesis for Training

To train our agentic model, we require supervision in the form of executable data preparation pipelines together with their source and target tables. Since such ADP task instances are not available at scale, we construct them through synthesis by converting SQL benchmarks into ADP tasks consisting of source tables, a target table, and a corresponding transformation pipeline. Synthesizing such training tasks has two key challenges. First, the generated pipelines should correspond to meaningful analytical transformations rather than arbitrary operator chains, so that the supervision reflects realistic operator dependencies. Second, the synthesized source tables should include diverse data quality issues while preserving the intended transformation logic. Excessive noise injection may break key fields or relationships [30], making it difficult to construct any feasible pipeline that produces the target table.

ADP Task Construction from NL2SQL. To obtain realistic ADP tasks, we instantiate them from NL2SQL benchmarks [26, 43]. For each query q, we execute it to produce the target table T^* and use an LLM to generate the corresponding target schema specification. We also translate q into an operator pipeline by generating candidate pipelines with an LLM and selecting the shortest one that exactly reproduces T^* through execution. This yields a clean task instance consisting of source tables, a target schema specification, a target table, and an operator pipeline.

Reversible Noise Injection. Since NL2SQL data is typically clean, we introduce controlled noise to increase data diversity for training. We inject noise by applying inverse transformations of data-cleaning operators, so that each corruption corresponds to a valid cleaning step. Specifically, we sample an operator and use an LLM to generate its inverse transformation logic. For example, the inverse of a date-standardization operator converts a normalized date (*e.g.,* "2023-01-01") into heterogeneous formats (*e.g.,* "01/01/23"). After each corruption step, we verify executability by applying the corresponding cleaning operator and checking that the previous table state is restored. Only reversible corruptions are kept. Repeating this process produces dirty source tables together with a matching cleaning pipeline. The final ground-truth pipeline is formed by concatenating the cleaning pipeline with the task pipeline.

6 EXPERIMENTS

6.1 Experimental Setup

Datasets. As shown in Table 1, we evaluate DeepPrep on four datasets, including one public benchmark, two datasets synthesized using our synthesis method, and one real-world dataset.

Table 1: Statistics of datasets.

Dataset	Statistics		Pipeline Length	# Op Types
	# Train	# Test		
Synth-Spider	6,788	2,008	1∼28	31
Synth-Bird	-	1,150	2∼25	31
Parrot [13]	13,965	1,365	1∼8	17
Buildings [35]	-	105	-	-

(1) **Parrot** [13] is a benchmark for translating natural language into data preparation pipelines. To make it suitable for ADP evaluation, we augment each case by generating the task specification from the target tables and operator sequences using an LLM [33].

Note that the public benchmark **Parrot** [13] mainly contains short operator pipelines and limited operator-type coverage. We therefore synthesize two datasets to provide more complex pipelines and broader operator coverage, using our method in Section 5.3.

(2) **Synth-Spider** is a dataset synthesized by our method from the Spider NL2SQL benchmark [43]. Using the ADP task synthesis pipeline described in Section 5.3, we convert NL2SQL cases into ADP tasks with executable operator pipelines.

(3) **Synth-Bird** is another synthesized dataset constructed from the Bird benchmark [26], which features larger databases and more complex analytical queries. We apply the same synthesis procedure to generate ADP tasks from the Bird validation set. As we use Synth-Bird only for evaluation, this dataset only has test split.

(4) **Buildings** [35] is a real-world dataset comprising 105 tasks collected from data processing pipelines in the smart building domain across 21 energy companies. It involves messy, real-world data with complex multi-step data preparation operations. It only provides source tables and target table schema descriptions.

Evaluation Methodology. For comprehensive evaluation, all models are trained only on the **Synth-Spider** training set and evaluated under two test settings: *in-domain* and *out-of-domain*.

(1) In-domain Test. We evaluate on the **Synth-Spider** test set, which is synthesized using the same procedure as the training data.

(2) Out-of-domain Test. **Synth-Bird** is closer to the training distribution since it is generated by the same synthesis pipeline but from a different benchmark. **Parrot** is an independent dataset with different task characteristics. **Buildings** is a real-world dataset with complex multi-step data preparation.

Baselines. We compare DeepPrep with both *prompting baselines* and *training baselines*. Prompting baselines use off-the-shelf LLMs, while training baselines involve parameter updates.

(1) Prompting Baselines. These methods use frozen LLMs with different planning and search strategies to generate pipelines.

- CodeGen [28, 32] prompts an LLM to iteratively generate and execute data preparation code until the target table is obtained.
- Plan-and-Solve (PaS) [38] prompts an LLM to produce a plan and then generates an operator pipeline to obtain the target table.
- ReAct [2] is a linear reasoning agent that iteratively predicts the next operator based on previous execution results.
- MCTS-OP [23] applies Monte Carlo Tree Search (MCTS) over individual operators defined in Section 2.2
- MontePrep [12] applies MCTS over actions such as schema mapping, operator discovery, code synthesis, and refinement instead of individual operators.
- DeepAnalyze [44] is an agentic LLM that generates data processing programs using supervised and reinforcement learning.
- AutoPrep [10] is a multi-agent framework for data preparation. It generates a high-level plan and then produces data preparation programs through specialized agents.
- TRAE and Claude Code (CC) are AI-powered coding agents with agentic coding capabilities. We evaluate them as representative state-of-the-art agentic coding systems*.

(2) Training Baselines. These baselines use the same backbone models as DeepPrep but adopt different training strategies, allowing us to evaluate the effect of our agentic training framework.

- Imitation Learning (IL) [25] fine-tunes the model on the high-quality trajectories distilled from strong teacher models.
- RL-GRPO [34] applies reinforcement learning with Group Relative Policy Optimization using only outcome-level rewards.

Prompt Design and Tuning for Baseline Methods. Specifically, (1) for methods with public implementations, i.e., MontePrep [12], DeepAnalyze [44], AutoPrep [10], and MCTS-Op [23], we use their official codebases and default configurations (e.g., prompt design, parameters, etc.). When a method was designed for a different task and could not be directly applied to ADP, we adapted its prompt to the ADP setting and tuned its parameters using the same grid-search procedure adopted by DeepPrep. For example, AutoPrep was designed for TableQA rather than autonomous data preparation, so we replaced its original prompt with an ADP-oriented prompt and tuned it accordingly. (2) For methods without public implementations, including CodeGen, ReAct, and PaS, we reimplemented them following the descriptions in their papers and applied the same prompt-tuning and parameter-selection procedure used for DeepPrep. Therefore, all methods were evaluated under a comparable tuning method. Complete prompts, hyperparameters, and implementation details are provided in our technical report [1].

Evaluation Metrics. We evaluate all methods using three metrics.

(1) Exact Match Accuracy (Acc.) measures the percentage of test cases where the produced table $\hat{T}$ matches the ground-truth table T^*. We follow the table matching method in [13], which is invariant to row and column permutations but requires exact cell-value equality.

(2) Completion Rate (Comp.) measures the percentage of test cases that produce a final output table. A case is counted as incomplete if execution exceeds the maximum interaction limit, triggers runtime errors, or produces an empty result.

(3) Inference Cost (Cost) measures the average monetary cost per case. For closed-source LLMs, we use official API pricing. For open-source models, we convert end-to-end runtime to cost using the hourly GPU price: $c_i = p_{\text{gpu}} t_i / 3600$, where t_i is the runtime in seconds. We report the average over all cases, using an A800-80G GPU priced at \$0.91/hour.

Backbone LLMs and DeepPrep Variants. We implement DeepPrep on both closed-source and open-source LLM backbones.

(1) Closed-source backbones. We implement DeepPrep with our proposed tree-based agentic reasoning directly on the strongest proprietary models, including GPT-5 (*i.e.,* gpt-5-mini-2025-08-07) and Claude-4 (*i.e.,* claude-sonnet-4-20250514).

*We use CC v2.1.119 and TRAE from https://github.com/bytedance/trae-agent.

Table 2: Comparison of DeepPrep with baselines on open-source LLMs. The best and second-best results are highlighted in bold and underlined, respectively.

(a) Results on Qwen3-14B.

Method	Synth-Spider		Synth-Bird		Parrot	
	Acc.	Comp.	Acc.	Comp.	Acc.	Comp.
Prompting Baselines						
PaS	7.67	20.67%	3.09	21.86%	9.56	22.36%
MCTS-OP	21.91	72.56%	7.22	78.69%	24.95	83.02%
MontePrep	9.46	19.62%	18.63	38.15%	26.94	57.87%
AutoPrep	36.75	71.61%	10.41	39.89%	28.54	74.05%
CodeGen	45.47	68.18%	29.48	49.20%	30.83	81.08%
ReAct	40.39	69.02%	16.04	46.56%	30.80	76.44%
Training Methods (Only Trained on Synth-Spider)						
IL	61.70	85.26%	48.01	81.23%	37.95	89.39%
RL-GRPO	51.22	77.68%	32.25	67.22%	35.06	82.9%
DeepPrep (Ours)	**67.18**	**97.21%**	**54.09**	**92.26%**	**40.44**	**96.41%**

(b) Results on Qwen3-8B.

Method	Synth-Spider		Synth-Bird		Parrot	
	Acc.	Comp.	Acc.	Comp.	Acc.	Comp.
Prompting Baselines						
PaS	4.68	7.42%	0.65	4.33%	3.09	8.32%
MCTS-OP	12.90	15.19%	0.90	73.56%	17.33	23.71%
MontePrep	8.81	19.97%	13.70	21.96%	25.95	56.08%
AutoPrep	26.44	50.85%	8.52	42.28%	25.20	70.42%
CodeGen	6.27	12.10%	7.47	15.69%	7.27	25.50%
ReAct	29.63	52.59%	10.41	46.22%	22.11	57.97%
DeepAnalyze	35.66	82.02%	1.74	62.17%	29.01	66.67%
Training Methods (Only Trained on Synth-Spider)						
IL	59.65	90.54%	43.30	80.43%	36.15	91.08%
RL-GRPO	47.81	76.54%	29.95	68.69%	32.82	85.05%
DeepPrep (Ours)	**65.99**	**97.46%**	**53.39**	**92.78%**	**39.93**	**98.46%**

(2) Open-source backbones. We implement DeepPrep based on open-source LLMs, *i.e.,* Qwen2.5 [19] (with 0.5B, 1.5B, 3B, 7B and 14B parameters) and Qwen3 [39] (0.6B, 1.7B, 4B, 8B and 14B).

6.2 Overall Comparison

Exp-1: Accuracy and Completion Rate Comparison. We compare DeepPrep with prompting baselines and training baselines on accuracy and completion rate. Results on Qwen3-14B and Qwen3-8B are reported in Table 2, as they are strong open-source backbones at two model scales commonly used in code and data tasks. Additional backbone results are presented in later experiments.

(1) In-Domain Test. DeepPrep achieves the highest accuracy and completion rate across both backbones on the Synth-Spider test set. On Qwen3-14B, it reaches 67.18 accuracy (Table 2a), exceeding the strongest prompting baseline CodeGen by 21.71 points and the strongest training baseline by 5.48 points, with similar gains in completion rate. These gains stem from combining tree-structured exploration with execution-grounded actions, enabling the agent to revise earlier decisions and avoid invalid operator sequences.

(2) Out-of-Domain Test. Under the single-source training method, DeepPrep maintains a clear advantage on out-of-domain datasets. On Synth-Bird, DeepPrep (Qwen3-14B) achieves 54.09 accuracy compared with 48.01 for IL, and the 8B model achieves 53.39 versus 43.30. On Parrot, we observe a similar ordering among methods. DeepAnalyze shows much lower performance on Synth-Bird (1.74 accuracy), indicating limited robustness when pipeline length and operator dependencies increase. These results align with our training design, where progressive curriculum and hybrid rewards expose the model to multi-step, feedback-driven reasoning trajectories rather than single-pass program generation.

Table 3: Overall Ablation Study on Qwen3-8B. PAT represents Progressive Agentic Training and TAR represents Tree-based Agentic Reasoning.

Method	Synth-Spider		Synth-Bird		Parrot	
	Acc.	Comp.	Acc.	Comp.	Acc.	Comp.
DeepPrep	**65.99**	**97.46%**	**53.39**	**92.78%**	**39.93**	**98.46%**
w/o PAT	22.21	59.76%	8.87	41.90%	25.27	70.62%
w/o PAT, TAR	4.68	7.42%	0.65	4.33%	3.09	8.32%

Finding 1: DeepPrep achieves the highest accuracy and completion rate among the compared methods and shows good generalization across datasets.

Exp-2: Cost–Accuracy Trade-off Comparison. We compare methods under a cost–accuracy trade-off by jointly evaluating accuracy and cost. Figure 5 plots prompting baselines and DeepPrep variants across different LLM backbones, and shows two Pareto frontiers: *Previous Pareto*, computed from baselines only, and *Ours Pareto*, computed after adding DeepPrep.

First, DeepPrep lies on or near the Pareto frontier in most settings and extends the frontier across all three datasets. In Figure 5, the *Ours Pareto* curve is shifted toward the upper-left relative to the *Previous Pareto*, meaning that DeepPrep achieves higher accuracy at the same cost, or similar accuracy at lower cost. This appears on Synth-Spider, Synth-Bird, and Parrot, indicating that the cost–accuracy advantage is consistent across datasets and backbones.

Second, DeepPrep reaches accuracy close to strong closed-source baselines at substantially lower cost. On Synth-Spider, DeepPrep (Qwen3-14B) achieves 67.18 accuracy at a very low cost, achieving a comparable accuracy with ReAct on Claude-4 (69.92 accuracy) while reducing cost by about 57×. This is attributed to our design that uses structured tree-based reasoning to guide smaller or open models toward valid pipelines with fewer failed executions.

Third, compared with the agentic coding systems TRAE and Claude Code, DeepPrep achieves comparable accuracy at much lower cost. For example, with GPT-5, DeepPrep achieves accuracy comparable to TRAE while reducing cost by 2.7× on Synth-Spider. This further demonstrates the cost-effectiveness of DeepPrep.

Finding 2: DeepPrep extends the cost–accuracy Pareto frontier, achieving accuracy close to strong closed-source baselines at substantially lower inference cost.

6.3 Ablation Studies

Exp-3: Overall Ablation Studies of DeepPrep. We evaluate the contribution of two modules (TAR and PAT) in DeepPrep. The results are reported in Table 3. As reported, removing PAT while retaining TAR causes substantial degradation, reducing accuracy by 43.78 on Synth-Spider, 44.52 on Synth-Bird, and 14.66 on Parrot. This indicates that the model must first learn the operator syntax

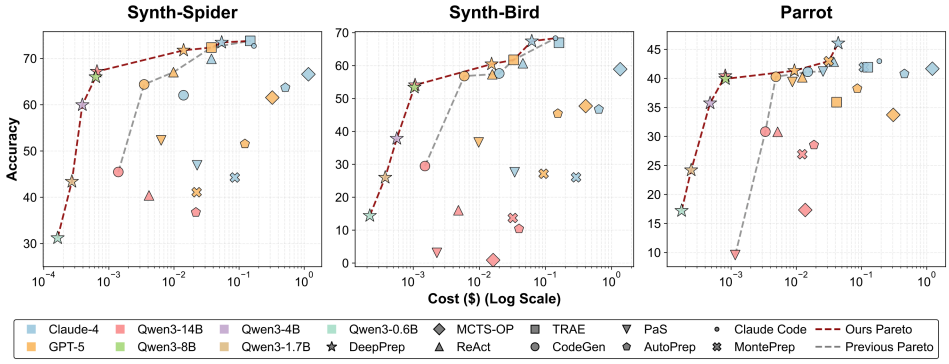

Figure 5: Cost–Accuracy trade-off comparison. Colors represent different LLM backbones, and shapes denote different methods. The red and grey dashed lines indicate the Pareto frontiers of our method and previous baselines, respectively.

Table 4: Detailed Ablation Studies on PAT (Qwen3-8B).

(a) Ablation study on Cold Start stage. OSL represent Operator syntax Learning module in Section 5.1.

Method	Synth-Spider		Synth-Bird		Parrot	
	Acc.	Comp.	Acc.	Comp.	Acc.	Comp.
Cold-Start	**60.66**	**91.63%**	**45.27**	**81.39%**	**36.22**	**92.16%**
w/o OSL	59.65	90.54%	43.30	80.43%	36.15	91.08%

(b) Ablation study on the reward of Multiturn RL stage.

Method	Synth-Spider		Synth-Bird		Parrot	
	Acc.	Comp.	Acc.	Comp.	Acc.	Comp.
DeepPrep	**65.99**	97.46%	**53.39**	92.78%	**39.93**	98.46%
w/o R_{llm}	64.54	96.96%	51.67	92.31%	39.12	94.21%
w/o R_{llm}, R_{part}	61.85	**98.71%**	47.15	**95.57%**	37.22	**99.34%**

and reasoning protocol before tree-based reasoning can be effectively utilized. Further removing TAR causes the method to degenerate into single-turn operator generation, leading to a larger performance drop (by 61.31 on Synth-Spider). Meanwhile, the completion rate falls to near-zero levels. This demonstrates that TAR is critical for leveraging execution feedback, and recovering from erroneous decisions through backtracking.

Exp-4: Detailed Ablation Studies on Progressive Agentic Training. We evaluate the contribution of specific components within our progressive agentic training. The results are detailed in Table 4.

Table 4a demonstrates that cold-start curriculum is useful. Removing this component causes a simultaneous decline in both accuracy and completion rate across all datasets. For instance, the completion rate on Synth-Spider drops from 91.63 to 90.54. Because this component familiarizes the model with the operator syntax and parameters. Without it, the model tends to generate non-executable actions and fail to complete the reasoning trajectory.

Table 4b validates the design of our reward function in RL. First, removing process reward (R_{llm}) results in an accuracy drop, which shows that R_{llm} prevents "reward hacking". Without semantic verification, the policy learns to satisfy superficial schema constraints, such as renaming columns, while neglecting the actual data transformation logic. Second, relying solely on outcome rewards (w/o R_{out}, R_{part}) yields the highest completion rate but the lowest accuracy. This suggests that sparse rewards induce conservative behavior. The agent avoids complex exploration or backtracking to minimize the risk of runtime errors. Thus, it prioritizes safe but incorrect paths to ensure the episode terminates successfully.

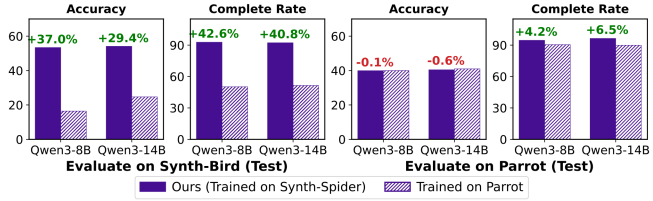

Figure 6: Evaluation results of models trained on Synth-Spider (Train) and Parrot (Train).

Table 5: Results on the Buildings dataset (real-world).

Method	Acc.	Comp.	Cost ($)
ReAct	74.29	99.05%	**0.0052**
CodeGen	71.43	100%	0.0054
MontePrep	72.38	86.67%	0.043
MCTS-OP	69.52	99.05%	0.057
DeepPrep (Ours)	**82.85**	**100%**	0.0082

Finding 3&4: Each key component of DeepPrep contributes to its overall performance.

6.4 In-Depth Analysis

Exp-5: Effectiveness of Synthesized Training Data. We evaluate whether models trained on our synthesized Synth-Spider dataset generalize better than those trained on the public Parrot dataset. We first evaluate these two models on Synth-Bird (Test) under the out-of-domain setting. As illustrated in the left of Figure 6, training on Synth-Spider yields much higher performance: accuracy increases by 37.0 and 29.4 points on Qwen3-8B and Qwen3-14B over Parrot. To further highlight the effectiveness of synthesized data, we adopt a more challenging setting. We evaluate models trained on Synth-Spider under a totally out-of-domain setting and models trained on Parrot under an in-domain setting on Parrot (Test), as illustrated in the right of Figure 6. Our synthesized data are still competitive under such a challenging experimental setting. Models trained on Synth-Spider achieve accuracy comparable to models trained on Parrot (40.44 vs. 40.99 on Qwen3-8B).

Finding 5: Our synthetic training data provides effective supervision for agentic training, significantly improving the performance on unseen tasks.

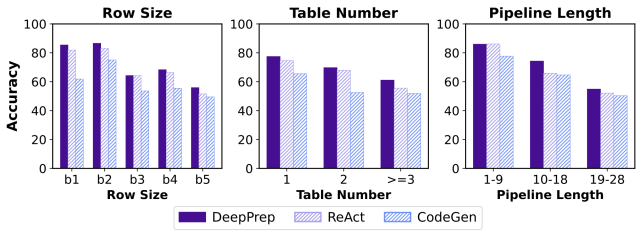

Figure 7: Scalability analysis on Synth-Spider with varying (a) row size, (b) source table number, and (c) pipeline length.

Exp-6: Generalization on Real-World Dataset. We have compared DeepPrep with prompting baselines on the real-world dataset Buildings [35]. Table 5 reports Accuracy, Completion Rate and API Cost per case. As illustrated, DeepPrep generalizes well on the real-world dataset. Specifically, DeepPrep achieves the best Accuracy of 82.85 and maintains relatively low inference cost (0.008$ per case). This enhances its usability in real-world.

Error Analysis. We further conduct an error analysis of this experiment and identify two key challenges in real-world scenarios. (1) Real cases often contain abbreviated or domain-specific column names, leading to selecting the wrong columns required to be mapped into the target columns. (2) The target specification in real cases is often under-specified. For example, whether daily or monthly granularity is expected is not specified clearly in the requirement. We will address these challenges in our future work. The complete statistics can be found in our technical report [1].

Finding 6: DeepPrep generalizes well to the real-world Buildings dataset.

Exp-7: Scalability Analysis. We evaluate the scalability of DeepPrep on Synth-Spider from three dimensions: row size, number of source tables, and pipeline length.

Scalability Analysis of Accuracy. As shown in Figure 7(a), DeepPrep remains robust on small and medium-sized tables. Performance degrades only for the largest row-size bucket (b_5: 36 to 124k+ rows), where accuracy decreases by 29.54 points compared with the smallest bucket (b_1: 1 to 9 rows). This degradation is primarily caused by the table sampling strategy used to control token usage, where relevant information may be omitted from the sampled content. As shown in Figure 7, performance decreases as more source tables are involved, but DeepPrep consistently achieves the highest accuracy across different table number groups. As illustrated, all methods degrade as the pipeline length increases. However, DeepPrep shows the smallest degradation, and its advantage over baselines becomes larger on complex cases.

Scalability Analysis of Time and Resource Usage. DeepPrep exhibits stable growth in both execution time and memory usage as workload complexity increases. For example, on Synth-Spider, the average execution time increases only from 8.91 to 11.30 seconds when moving from the smallest to the largest column-size bucket, while memory usage remains between 66 MB and 69 MB. The stable scaling behavior can be explained by the generative search framework, as defined in Section 4. The process is bounded by the maximum exploration turn T and the maximum operator sequence length L per expansion, yielding a worst-case complexity of $O(T \times L)$. Unlike MCTS-based methods, DeepPrep generates only one operator pipeline conditioned on the current tree state. The "pruning" is implicit in the LLM's decision-making. Consequently, the effective branching factor depends on the LLM's learned policy and remains bounded regardless of input size, which explains the stable growth in time and memory usage.

Complete scalability results across all datasets and detailed time and resource usage analysis are provided in our technical report [1].

Finding 7: DeepPrep maintains robust accuracy and stable resource usage as workload complexity increases.

7 RELATED WORK

Data Preparation. Prior data preparation work mainly focuses on programming-by-example (PBE). PBE systems [3, 14, 17, 20, 36] synthesize transformation programs from input-output examples, which are effective for well-specified transformations but require example-level supervision for each task. Other systems target specific sub-problems of data preparation [5, 10, 11, 18, 21, 27, 40]. For example, Auto-Tables [27] focuses on table reconstruction operations such as pivoting and stacking. Such systems emphasize structural transformations but are not designed to jointly handle value-level cleaning [4] and multi-table integration [47]. Text-to-Pipeline [13] maps natural language descriptions to pipelines, but assumes detailed step-wise specifications as input. In contrast, DeepPrep targets end-to-end autonomous data preparation from high-level target schema descriptions and supports a much broader operator space covering cleaning, reshaping, and integration.

LLM-Powered Data Preparation. Recent work explores using LLMs for data preparation pipeline construction. Prompting-based approaches directly generate transformation code from natural language specifications [21, 28, 32]. To incorporate execution feedback, ReAct-style methods interleave reasoning with tool actions and update subsequent decisions based on observed results [2, 42]. Such approaches follow a linear interaction trajectory, where actions are appended sequentially. More recent work explores tree-structured search over reasoning or operator spaces, including Tree-of-Thoughts (ToT) [41] and MCTS-based agents [12, 23]. For example, MCTS-style approaches typically rely on rollout-based value estimates and scalar reward signals to guide node selection and backpropagation [16]. However, data preparation pipelines expose structured execution feedback, such as schema mismatches, execution errors for operators, and intermediate table states. DeepPrep is designed to operate directly over such execution-grounded states, where each node corresponds to materialized tables and branch decisions are guided by structured execution feedback rather than scalar rollout statistics. This design supports operator-level revision and state-aware backtracking during pipeline construction.

ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China (Grant Nos. 62436010, 62441230), the Scientific Research Innovation Capability Support Project for Young Faculty (Grant No. SRICSPYF-ZY2025001), Beijing Natural Science Foundation (L244010) and the Outstanding Innovative Talents Cultivation Funded Programs 2025 of Renmin University of China.

REFERENCES

[1] 2026. *Technical Report.* https://github.com/ruc-datalab/DeepPrep/tree/main/pdf/report.pdf

[2] Renat Aksitov, Sobhan Miryoosefi, Zonglin Li, Daliang Li, Sheila Babayan, Kavya Kopparapu, Zachary Fisher, Ruiqi Guo, Sushant Prakash, Pranesh Srinivasan, et al. 2023. Rest meets react: Self-improvement for multi-step reasoning llm agent. *arXiv preprint arXiv:2312.10003* (2023).

[3] Daniel W Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. *ACM SIGPLAN Notices* 50, 6 (2015), 218–228.

[4] Chengliang Chai, Kaisen Jin, Nan Tang, Ju Fan, Dongjing Miao, Jiayi Wang, Yuyu Luo, Guoliang Li, Ye Yuan, and Guoren Wang. 2025. Cost-effective Missing Value Imputation for Data-effective Machine Learning. *ACM Transactions on Database Systems* 50, 3 (2025), 1–36.

[5] Qixu Chen, Yeye He, Raymond Chi-Wing Wong, Weiwei Cui, Song Ge, Haidong Zhang, Dongmei Zhang, and Surajit Chaudhuri. 2025. Auto-Test: Learning Semantic-Domain Constraints for Unsupervised Error Detection in Tables. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–27.

[6] Tamraparni Dasu and Theodore Johnson. 2003. *Exploratory data mining and data cleaning.* John Wiley & Sons.

[7] Chao Deng, Ju Fan, Yuyu Luo, Qinliang Xue, Meihao Fan, Yuxin Zhang, Min Zhang, Xiaofeng Jia, Jing Zhang, and Xiaoyong Du. 2026. TACO: A Benchmark for Open-Domain Text-to-SQL with Ambiguous and Cross-Database Queries. *arXiv preprint arXiv:2606.14201* (2026).

[8] Haoyu Dong and Zhiruo Wang. 2024. Large language models for tabular data: Progresses and future directions. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 2997–3000.

[9] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, et al. 2024. A survey on in-context learning. In *Proceedings of the 2024 conference on empirical methods in natural language processing.* 1107–1128.

[10] Meihao Fan, Ju Fan, Nan Tang, Lei Cao, Guoliang Li, and Xiaoyong Du. 2024. AutoPrep: Natural Language Question-Aware Data Preparation with a Multi-Agent Framework. *CoRR* abs/2412.10422 (2024). https://doi.org/10.48550/ARXIV.2412.10422 arXiv:2412.10422

[11] Meihao Fan, Xiaoyue Han, Ju Fan, Chengliang Chai, Nan Tang, Guoliang Li, and Xiaoyong Du. 2024. Cost-effective in-context learning for entity resolution: A design space exploration. In *2024 IEEE 40th International Conference on Data Engineering (ICDE).* IEEE, 3696–3709.

[12] Congcong Ge, Yachuan Liu, Yixuan Tang, Yifan Zhu, Yaofeng Tu, and Yunjun Gao. 2025. MontePrep: Monte-Carlo-Driven Automatic Data Preparation without Target Data Instances. *arXiv preprint arXiv:2509.17553* (2025).

[13] Yuhang Ge, Yachuan Liu, Yuren Mao, and Yunjun Gao. 2025. Text-to-Pipeline: Bridging Natural Language and Data Preparation Pipelines. *arXiv preprint arXiv:2505.15874* (2025).

[14] Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

[15] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[16] Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing.* 8154–8173.

[17] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-data-by-example (TDE) an extensible search engine for data transformations. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1165–1177.

[18] Zhipeng Huang and Yeye He. 2018. Auto-detect: Data-driven error detection in tables. In *Proceedings of the 2018 International Conference on Management of Data.* 1377–1392.

[19] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[20] Zhongjun Jin, Michael R Anderson, Michael Cafarella, and HV Jagadish. 2017. Foofah: Transforming data by example. In *Proceedings of the 2017 ACM International Conference on Management of Data.* 683–698.

[21] Eugenie Y Lai, Yeye He, and Surajit Chaudhuri. 2025. Auto-Prep: Holistic Prediction of Data Preparation Steps for Self-Service Business Intelligence. *arXiv preprint arXiv:2504.11627* (2025).

[22] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems* 35 (2022), 21314–21328.

[23] Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025. Alpha-sql: Zero-shot text-to-sql using monte carlo tree search. *arXiv preprint arXiv:2502.17248* (2025).

[24] Changlun Li, Chenyu Yang, Yuyu Luo, Ju Fan, and Nan Tang. 2025. Weak-to-strong prompts with lightweight-to-powerful llms for high-accuracy, low-cost, and explainable data transformation. *Proceedings of the VLDB Endowment* 18, 8 (2025), 2371–2384.

[25] Dacheng Li, Shiyi Cao, Tyler Griggs, Shu Liu, Xiangxi Mo, Eric Tang, Sumanth Hegde, Kourosh Hakhamaneshi, Shishir G Patil, Matei Zaharia, et al. 2025. LLMs Can Easily Learn to Reason from Demonstrations Structure, not content, is what matters! *arXiv preprint arXiv:2502.07374* (2025).

[26] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems* 36 (2024).

[27] Peng Li, Yeye He, Cong Yan, Yue Wang, and Surajit Chaudhuri. 2023. Auto-Tables: Synthesizing Multi-Step Transformations to Relationalize Tables without Using Examples. *Proc. VLDB Endow.* 16, 11 (2023), 3391–3403. https://doi.org/10.14778/3611479.3611534

[28] Xue Li and Till Döhmen. 2024. Towards efficient data wrangling with llms using code generation. In *Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning.* 62–66.

[29] Weizheng Lu, Jing Zhang, Ju Fan, Zihao Fu, Yueguo Chen, and Xiaoyong Du. 2025. Large language model for table processing: A survey. *Frontiers of Computer Science* 19, 2 (2025), 192350.

[30] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. 2017. Holoclean: Holistic data repairs with probabilistic inference. *arXiv preprint arXiv:1702.00820* (2017).

[31] Yucheng Ruan, Xiang Lan, Jingying Ma, Yizhi Dong, Kai He, and Mengling Feng. 2024. Language modeling on tabular data: A survey of foundations, techniques and evolution. *arXiv preprint arXiv:2408.10548* (2024).

[32] Ranjan Sapkota, Konstantinos I Roumeliotis, and Manoj Karkee. 2025. Vibe coding vs. CodeGen: Fundamentals and practical implications of agentic ai. *arXiv preprint arXiv:2505.19443* (2025).

[33] ByteDance Seed, Jiaze Chen, Tiantian Fan, Xin Liu, Lingjun Liu, Zhiqi Lin, Mingxuan Wang, Chengyi Wang, Xiangpeng Wei, Wenyuan Xu, et al. 2025. Seed1.5-thinking: Advancing superb reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.13914* (2025).

[34] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).

[35] Ankita Sharma, Xuanmao Li, Hong Guan, Guoxin Sun, Liang Zhang, Lanjun Wang, Kesheng Wu, Lei Cao, Erkang Zhu, Alexander Sim, et al. 2023. Automatic data transformation using large language model-an experimental study on building energy data. In *2023 IEEE International Conference on Big Data (BigData).* IEEE, 1824–1834.

[36] Rishabh Singh. 2016. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment* 9, 10 (2016), 816–827.

[37] Mia Taylor, James Chua, Jan Betley, Johannes Treutlein, and Owain Evans. 2025. School of reward hacks: Hacking harmless tasks generalizes to misaligned behavior in llms. *arXiv preprint arXiv:2508.17511* (2025).

[38] Xuezhi Wang and Denny Zhou. 2024. Chain-of-thought reasoning without prompting. *Advances in Neural Information Processing Systems* 37 (2024), 66383–66409.

[39] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[40] Junwen Yang, Yeye He, and Surajit Chaudhuri. 2021. Auto-pipeline: synthesizing complex data pipelines by-target using reinforcement learning and search. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2563–2575.

[41] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems* 36 (2023), 11809–11822.

[42] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR).*

[43] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing.* 3911–3921.

[44] Shaolei Zhang, Ju Fan, Meihao Fan, Guoliang Li, and Xiaoyong Du. 2025. Deepanalyze: Agentic large language models for autonomous data science. *arXiv preprint arXiv:2510.16872* (2025).

[45] Yuxin Zhang, Ju Fan, Meihao Fan, Shaolei Zhang, and Xiaoyong Du. 2026. CODA-BENCH: Can Code Agents Handle Data-Intensive Tasks? *arXiv preprint arXiv:2606.15300* (2026).

[46] Yuxin Zhang, Meihao Fan, Ju Fan, Mingyang Yi, Yuyu Luo, Jian Tan, and Guoliang Li. 2025. Reward-sql: Boosting text-to-sql via stepwise reasoning and process-supervised rewards. *arXiv preprint arXiv:2505.04671* (2025).

[47] Erkang Zhu, Yeye He, and Surajit Chaudhuri. 2017. Auto-join: Joining tables by leveraging transformations. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1034–1045.