BAT: Target-Instance-Free Data Preparation Synthesis via LLM-Driven Tree Search

CONGCONG GE, Zhejiang University, China
YACHUAN LIU, Zhejiang University, China
YIXUAN TANG, National University of Singapore, Singapore
YIFAN ZHU, Zhejiang University, China
YAOFENG TU, ZTE Corporation, China
YUNJUN GAO, Zhejiang University, China

In real-world scenarios, a pervasive requirement for automatic data preparation (ADP) is to transform diverse relational tables from different sources into a specified target table. Previous methods rely on labor-intensive supervision signals or access permissions to target table data, limiting their usage in commercial systems.

To tackle these challenges, we propose BAT, the first end-to-end ADP framework. It enables the synthesis of training-free data preparation pipelines without requiring any instances from target tables. BAT is formulated as an open-source large language model (LLM) driven tree-structured search problem. It consists of three pivot components, i.e., a data preparation action sandbox (DPAS), a fundamental pipeline generator (FPG), and an execution-aware pipeline optimizer (EPO). We first introduce DPAS, a lightweight action sandbox, to navigate the search-based data preparation pipeline generation process. The design of DPAS circumvents exploration of infeasible pipelines. Then, we present FPG, an LLM-driven Monte Carlo tree search process, to incrementally generate executable DP pipelines within the constraints of the predefined action sandbox. Furthermore, we propose EPO, which invokes pipeline execution results from sources to targets to evaluate the reliability of the generated pipelines in FPG. In this way, unreasonable pipelines are eliminated, thus facilitating the search process from both efficiency and effectiveness perspectives. Extensive experiments on real-world datasets show that BAT significantly outperforms 5 SOTA competitors, achieving at least 8.74% improvements in the end-to-end data preparation pipeline execution accuracy.

CCS Concepts: • **Information systems → Extraction, transformation and loading**.

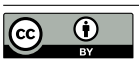
Additional Key Words and Phrases: Automatic Data Preparation, Monte Carlo Tree Search, Large Language Model

ACM Reference Format:
Congcong Ge, Yachuan Liu, Yixuan Tang, Yifan Zhu, Yaofeng Tu, and Yunjun Gao. 2026. BAT: Target-Instance-Free Data Preparation Synthesis via LLM-Driven Tree Search. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 143 (June 2026), 25 pages. https://doi.org/10.1145/3802020

1 Introduction

Self-service data preparation (DP) [8, 41], a prerequisite for data democratization, is becoming increasingly important. It aims to allow data scientists to prepare data without extensive domain

Authors' Contact Information: Congcong Ge, gcc@zju.edu.cn, Zhejiang University, School of Software Technology, Ningbo, China; Yachuan Liu, liuyachuan@zju.edu.cn, Zhejiang University, Polytechnic Institute, Hangzhou, China; Yixuan Tang, yixuan@comp.nus.edu.sg, National University of Singapore, Department of Computer Science, Singapore, Singapore; Yifan Zhu, xtf_z@zju.edu.cn, Zhejiang University, School of Software Technology, Ningbo, China; Yaofeng Tu, tu.yaofeng@zte.com.cn, ZTE Corporation, Shenzhen, China; Yunjun Gao, gaoyj@zju.edu.cn, Zhejiang University, College of Computer Science, Hangzhou, China.



ACM 2836-6573/2026/6-ART143
https://doi.org/10.1145/3802020

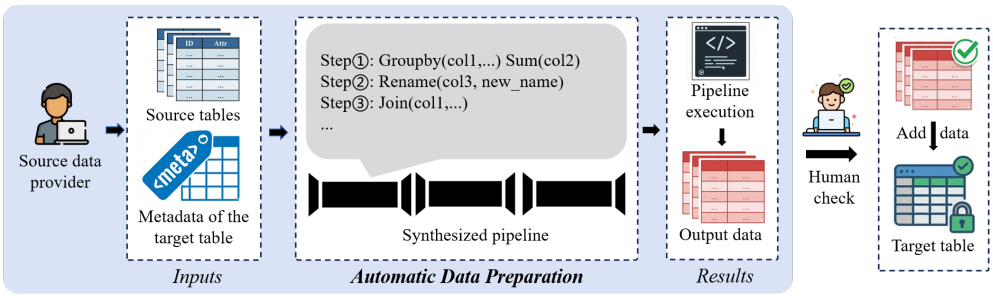


Fig. 1. An overview of the new ADP task.

knowledge or time-consuming efforts. *Automatic data preparation* (ADP) [49, 50] is an effective solution for self-service data preparation. It transfers raw data from disparate sources into formats that are ready for business intelligence (BI) analysis or machine learning purposes by generating sequential pipelines, which are composed of transformation operators (e.g., Join, Groupby, and Rename). Recently, various ADP techniques have emerged, such as transform-by-example [13], transform-by-pattern [25], transform-by-target [50], transform-for-joins [54], and LLM-based transformation [42]. Nonetheless, there is still a big room for implementing ADP tools in practice due to *source table variability* and *restricted data access privileges*.

New ADP task - an end-to-end solution for metadata-only data preparation. A fundamental ADP requirement is to transform diverse relational tables from different sources into a specified target table. In commercial systems, these source tables are dynamic, exhibiting schema variations across different sources and newly produced data over time [50]. Moreover, to ensure efficient collaboration across departments/regions, decentralized architectures like Data Mesh [18] have emerged as a dominant paradigm in DP. It shifts the preparation responsibility to source-side providers, leveraging their domain expertise to design valid preparation pipelines that adhere to business logic. However, this cross-departmental collaboration introduces a strict privacy constraint: due to compliance controls or data residency laws, source data providers are typically precluded from accessing target table instances [19, 45]. Thus, they must construct transformation pipelines solely based on the target's metadata (e.g, table schema or column descriptions) without visibility into the actual instances, referring to **metadata-only** data preparation. Figure 1 presents an overview of this task. However, automating such a task is non-trivial in practice. We now turn to illustrate the challenges via the Example 1.1 in detail.

Example 1.1. We consider a data preparation task in a supply chain scenario, where an ERP system integrates daily sales data from different regional stores (source) into a central data warehouse (target) in a Data Mesh architecture. Figure 2 depicts an example. Here, two distinct stores periodically submit sales data via source tables T_{I_1} and T_{I_2} to a target table T_r. Due to inconsistent local practices, these dynamic sources exhibit heterogeneous schemas Furthermore, strictly enforced privacy policies prevent providers from accessing target table instances. Consequently, they derive two different DP pipelines solely from the metadata to append their respective data to the target table, respectively. As depicted, the pipelines differ significantly: (i)T_{I_1} involves `Date Formatting`, `GroupBy`, `Rename`, and `Join` operators; whereas (ii) T_{I_2} necessitates an additional `Unpivot` operator to align the schema between the source and the target schemas.

Challenge I: *How to approximate reliable end-to-end pipelines under source variability and target instance scarcity.* Recall that the existing techniques are ineffective in metadata-only data preparation. They typically rely on ground truth guidance, such as "by-example" paradigm [2, 12, 13, 17, 46]

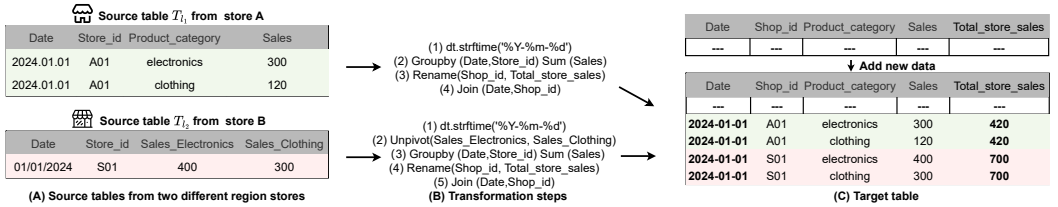


Fig. 2. A motivating example for metadata-only data preparation.

or auxiliary constraints (e.g., data distributions [50], data patterns [25], integrity constraints [50], or transformation hints [42]) to verify pipeline correctness. In our task, "verifiable correctness" is theoretically impractical due to the absence of ground truth. First, regarding "by-example" paradigm: they require golden samples (i.e., matching pairs between source and target) to verify the correctness of the synthesized pipeline. Since newly produced data from the source table has not yet been imported into the target, matching pairs are unavailable. Furthermore, the schema variants across sources make maintaining a sufficient set of matching pairs for every variant infeasible. These shortcomings render the "by-example" paradigm ineffective. Second, auxiliary constraints are either risky or unavailable: sharing distributions compromises privacy (e.g., re-identification [44]); tables in data lakes usually lack explicit rules (e.g., data patterns or integrity constraints); non-expert users cannot provide valid transformation hints. Recently, several commercial systems have adopted a "synthesis-then-verification" paradigm [20, 21]: metadata-based pipelines are first suggested for human review. Only verified pipelines are executed; otherwise, they are regenerated. However, these existing solutions predominantly focus on single-operator recommendations or rely on explicit transformation instructions, failing to synthesize complex, end-to-end pipelines. Consequently, the objective shifts from strict *validation* to *approximation*. **The key lies in synthesizing the most semantically plausible pipeline in an end-to-end manner to minimize subsequent human intervention**. However, achieving this approximation is non-trivial: source tables exhibit diverse schemas and ever-increasing data instances. Without instance-level feedback to prune incorrect candidates, this task faces a combinatorial explosion driven by the source variability. As detailed in Example 1.1, for the target column `Total_store_sales`, there exist multiple combinations of operators (e.g., specific `Groupby` keys, distinct `Join` conditions, or varying `Unpivot` configurations) that are syntactically valid but semantically divergent. Moreover, an end-to-end solution necessitates searching from countless possible combinations of transformation operators, which may lead to a huge exploration space. This characteristic significantly increases the difficulty of this task.

Challenge II: *How to enable user-friendly pipeline synthesis in a metadata-only setting?* To improve the efficiency of data preparation, modern data architectures are shifting pipeline creation from IT specialists to business users [50]. While this approach bypasses the delays of IT ticket processing, it creates a new hurdle: although business users possess domain knowledge, they typically lack the technical ability to implement data transformations. For instance, business users understand the semantics of the "Sales" column but typically lack the technical proficiency to implement complex transformation logic for it (e.g., specifying the *Unpivot* or *Groupby-Join* operators required in Example 1.1). This expertise limitation is significantly exacerbated when synthesizing end-to-end pipelines, as it requires users to logically orchestrate a valid sequence of multiple interdependent transformation operators. Recently, LLMs have demonstrated strong semantic comprehension capabilities, showing promise in lowering the barrier for less-technical users [28, 53]. An intuitive idea is to leverage LLMs in helping these non-technical users to synthesize end-to-end data preparation pipelines automatically. In the field of ADP, a recent trend has also seen both academia and

industries providing LLM-based solutions, such as single-step DP transformation inference [21] and knowledge-driven ADP [42]. However, applying LLMs to this new task is much more challenging than in other areas of ADP, due to the **hallucination problem**. Synthesizing DP pipelines for target tables that comply with a specific schema is typically a domain-specific task with low error tolerance. The absence of target instances makes the process highly susceptible to hallucinations. Even a minor hallucination can lead to a faulty pipeline that produces an output table whose schema does not fully match the target's, despite apparent semantic similarities. Thus, ensuring that the synthesized pipeline is executable and reliable without expert intervention or data visibility remains a formidable challenge.

To address these two challenges, we propose BAT to effectively synthesize the data preparation pipeline without any target table instances. BAT is formulated as an open-source LLM-powered tree search problem to effectively explore the space of DP transformation operators. Utilizing open-source LLMs (rather than closed-source ones) enables BAT to be deployed locally, thereby eliminating data leakage risks. BAT consists of three pivotal components: (i) an *DP action sandbox* termed as DPAS; (ii) a *fundamental pipeline generator* called FPG; and (iii) an *execution-aware pipeline optimizer* named EPO; First, DPAS is presented to restrict the search scope of DP pipeline generation within a bounded decision space. Second, FPG is to generate DP pipelines via the proposed LLM-driven Monte Carlo tree search. Here, each node corresponds to a DP transformation state, and each edge represents a DP transformation action from the proposed ADP action sandbox. Once a complete DP pipeline is generated, the LLM evaluates the reliability of the generated pipeline using a *self-reward mechanism*. Finally, BAT selects the most reliable pipeline as the final result. Furthermore, EPO is introduced to optimize the entire search process. It contains two perspectives of optimization. One optimizes the self-reward mechanism by invoking the pipeline execution results from sources to targets to evaluate the reliability of the generated pipelines. In this way, unreasonable pipelines will be eliminated. The other one is used to accelerate the search progress by a *simulation cache* mechanism and an *early termination* method.

Our contributions are summarized as follows:

- *User-friendly Framework*. BAT is the first end-to-end framework that generates DP pipelines with *zero* target instances and *zero* training efforts. It enables data providers to solve the DP problem in cross-domain scenarios without access to the target data, thus ensuring data confidentiality. This training-free automation eliminates expertise barriers, enabling non-technical users to participate in data preparation.
- *Lightweight Action Sandbox*. We define a lightweight DP action sandbox DPAS, which restricts the search scope of the DP pipeline generation process. DPAS circumvents exploration of infeasible transformation operators.
- *Execution-aware Optimizer*. We present EPO, an execution-aware optimizer, to facilitate the search process from both efficiency and effectiveness perspectives. EPO invokes the pipeline execution results from sources to targets to evaluate the reliability of the generated pipelines, In this way, unreasonable candidate pipelines will be eliminated in the search process.
- *Extensive Experiments*. We conduct comprehensive experimental evaluations on ADP. Extensive experiments on real-world datasets show that BAT significantly outperforms 5 SOTA competitors, achieving at least 8.74% improvements in accuracy.

Organization. The rest of the paper is organized as follows. Section 2 presents the preliminaries. Section 3 introduces the technical details of our proposed BAT framework. Section 4 reports the experimental results and our findings. Section 5 reviews the related work. Finally, Section 6 concludes the paper.

Table 1. Overview of supported transformation operators.

Category	Transformation Operators
Table-level	Join, GroupBy, Pivot, Unpivot, Union, Add/Drop Columns, Rename
Column-level	Column Arithmetic, Date Formatting

2 Preliminaries

In this section, we present some preliminaries, including problem statement, the supported DP transformation operators in BAT, and background techniques.

2.1 Problem Statement

As described in Section 1, we focus on the ADP task that transforms data from disparate sources into targets with specific schemas while requiring *zero* target-instance access. In commercial systems, despite the access limitation, business analysts and data engineers typically possess a clear understanding of the target schema that the enterprise data should conform to. To this end, it is realistic that we have access to the schema of target tables even when the data instances are unavailable.

Let $T = (S, D)$ be a relational table, where $S = \{s_1, s_2, \ldots, s_m\}$ is the schema of T and D denotes the corresponding data rows, a.k.a., data instances. A source table can be denoted as $T_l = (S_l, D_l)$ with schema $S_l = \{s_{l_1}, s_{l_2}, \ldots, s_{l_m}\}$ and a sampled data set $D_l^s \subseteq D_l$. $T_r = (S_r, D_r)$ represents a target table with schema $S_r = \{s_{r_1}, s_{r_2}, \ldots, s_{r_m}\}$. D_r is excluded from consideration in our settings due to the lack of access to target data instances. Unless otherwise stated, the target tables T_r mentioned in the following do not contain any data instances. The objective is to synthesize a data preparation pipeline $\mathcal{P} = (o_1(p_1), o_2(p_2), \ldots, o_{|o|}(p_{|o|}))$ from a set of source tables $\mathbb{T}_l$ to a target table T_r. Formally,

$$\mathcal{P}(\mathbb{T}_l) = \{o_1(p_1) \circ o_2(p_2) \cdots \circ o_{|o|}(p_{|o|})\} \rightarrow T_r. \tag{1}$$

Here, $\mathbb{T}_l = \{T_{l_1}, T_{l_2}, ..., T_{l_n}\}$, and $n \geq 1$. Each o_i is selected from a set of supported transformation operators O (see Section 2.2). Each p_i denotes the operator-specific parameters, such as join keys and aggregation columns. Note that, $\mathcal{P}$ is an executable program composed of a series of transformation operators.

It is important to illustrate that (i) our goal is to generate an executable and correct DP pipeline rather than a candidate one. The former can actually inject data from sources into the specific target table T_r with schema S_r, while the latter may generate a new table $\hat{T}_r$ that is semantically similar to T_r. This strict setting ensures the practicality of ADP in real-world scenarios. To be more concrete, commercial ADP tasks [22] require strict alignment of column names and column types between the sources and the targets; otherwise, the pipeline cannot be executed; (ii) no data instance from the target table T_r is available at synthesis time in our settings. The entire data preparation pipeline is required to be inferred solely from the source tables and the target schema without any supervision, constituting a significant challenge for the ADP task.

2.2 Supported Transformation Operators

We consider two broad categories of transformation operators in BAT, i.e., *table-level DP operators* and *column-level DP operators*, as summarized in Table 1.

Table-level DP operators. In this category, table-level DP operators can reshape the schema of the given table.

(1) *Join* [36] integrates information across multiple tables based on a shared key column. It corresponds to `pandas.merge()`.
(2) *GroupBy* [34] partitions data into groups based on one or more key columns and computes aggregate functions—such as `sum`, `mean`—for each group, aligning with `pandas.groupby()`.
(3) *Pivot* [37] transforms long-format data into wide-format by turning unique values in a categorical column into new columns, as implemented in `pandas.pivot()`.
(4) *Unpivot* [35] reverses a pivot operation by collapsing multiple columns into key-value pairs. It aligns with `pandas.melt()`.
(5) *Union* [32] combines multiple tables by appending rows with compatible schemas, typically along the vertical axis. This operation corresponds to `pandas.concat(..., axis=0)`.
(6) *Add/Drop Columns* adjusts the schema by introducing constant-valued columns or removing irrelevant ones. This is typically implemented via assignment or column selection in `pandas`.
(7) *Rename* [38], similar to `pandas.rename()`, standardizes column names for schema alignment or semantic clarity.

Column-level DP operators. In this category, DP operators focus on manipulating the contents of individual columns.

(1) *Column Arithmetic* applies arithmetic expressions across one or more columns to derive new values (e.g., `price * quantity`), using element-wise operations in `pandas`.
(2) *Date Formatting* [33] reformats or parses temporal values to enable downstream temporal analysis, such as converting strings to datetime objects or extracting date parts (e.g., year or month), via `pandas.to_datetime()` and `dt` accessors.

2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [6] is a heuristic algorithm used for solving sequential decision-making problems, such as game AI like AlphaGo and AlphaZero [10], that involve large and complex search spaces. MCTS contains the following four phases:

- **Selection**: Starting from the root, the algorithm explores the tree by selecting child nodes based on a specific strategy that can balance both exploration and exploitation. Here, researchers often utilize *Upper Confidence Bounds applied to Trees* (UCT) formula [26] as the balanced strategy.
- **Expansion**: Once a leaf node is reached, one or more feasible new child nodes, each of which corresponds to a potential future move, are added to the tree, unless the terminal state is reached.
- **Simulation**: A simulation method (often termed *rollout*) is executed from the newly expanded node, following a default policy (e.g., random) until a terminal state is reached, thereby evaluating the newly expanded node's potential.
- **Backpropagation**: The simulation result, i.e., the reward, is propagated back to the root through the visited path, updating the statistical information of each visited node, including node visit counts and an estimated node value.

During the search process, MCTS iteratively selects, expands, simulates, and backpropagates reward signals to explore the most promising paths. This approach supports constrained exploration over large but non-deterministic search spaces with feedback.

3 Our Framework

In this section, we describe our framework BAT in detail. Figure 3 illustrates the BAT architecture, which includes three components: a *DP action sandbox* (DPAS), followed by a *fundamental pipeline generator* (FPG) and an *execution-aware pipeline optimizer* (EPO).

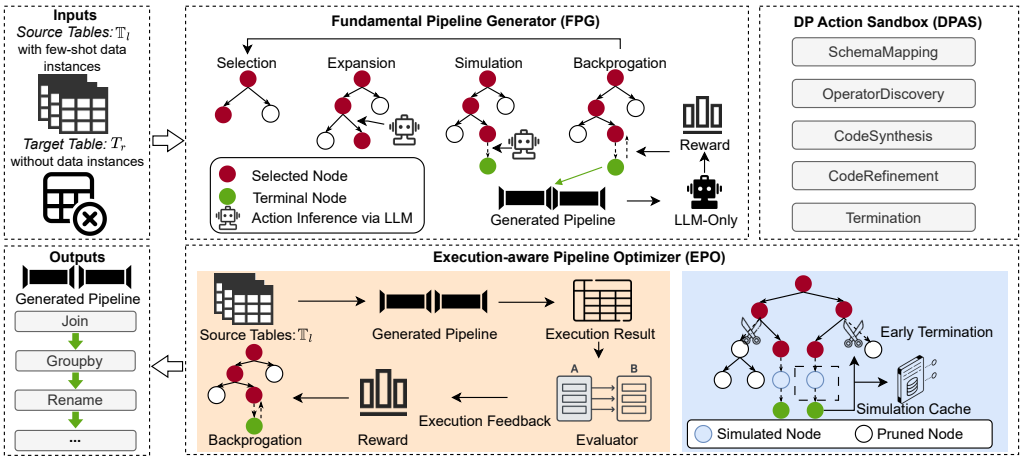


Fig. 3. Overview of the BAT framework for automatic data preparation.

3.1 DP Action Sandbox

Motivated by the empirical viability of search-oriented methods, we introduce a discrete symbolic sandbox (DPAS) to elevate reasoning abstraction. Unlike [50], which generates DP pipelines directly from a set of transformation operators, DPAS encapsulates transformation sub-goals into high-level actions, enabling explicit goal-oriented planning rather than ad-hoc code generation.

The sandbox is composed of five action types, termed as $\mathcal{A} = \{A_1, A_2, A_3, A_4, A_5\}$, each of which corresponds to a distinct transformation intent. A specific action $a_i = (A_j^i, \theta_i)$ consists of an action type $A_j^i \in \mathcal{A}$ and a parameter object θ_i instantiated based on the current state. Next, we detail the supported action types.

- A_1: `SchemaMapping`. In data preparation tasks, source and target table schemas often differ in naming, granularity, or structure. Automatically inferring how columns in the source table align with fields in the target schema is a prerequisite for meaningful transformation. This action establishes such correspondences by identifying semantically related columns.
- A_2: `OperatorDiscovery`. BAT formulates transformation operator discovery as a semantic planning step. This action leverages LLM to discover concrete transformation operators (as described in Section 2.2) based on partial context, including the schema of the target table, column descriptions, and schema mapping results. Here, the schema mapping results are optional. The reason is that the `OperatorDiscovery` action may be invoked either before or after the `SchemaMapping` action. This flexibility reflects common strategies in human data wrangling, where one may either hypothesize the transformation logic upfront or derive it after examining the data.
- A_3: `CodeSynthesis`. After identifying an appropriate transformation logic, BAT generates executable Python code to implement it. This action instantiates the chosen operator using a specific syntax, i.e., `pandas` in our current implementation, based on the mapped schema and discovered logic.
- A_4: `CodeRefinement`. In the `CodeSynthesis` action, the generated executable code may contain errors because of ambiguous schemas, inconsistent naming conventions, or incomplete specifications. To address these issues, we propose the `CodeRefinement` action. It aims to identify and correct such errors in the aforementioned executable code snippet. This action is crucial for robustness, particularly in zero-shot or noisy settings, where initial code generations are often imperfect.

Table 2. Available next actions in the DP action sandbox.

Action Type	Valid Next Actions
—	SchemaMapping, OperatorDiscovery, CodeSynthesis
SchemaMapping	OperatorDiscovery, CodeSynthesis
OperatorDiscovery	SchemaMapping, CodeSynthesis
CodeSynthesis	Termination, CodeRefinement
CodeRefinement	Termination
Termination	—

- A_5: `Termination`. This action signals the end of the search process. It triggers a validation step to check whether the composed pipeline satisfies the schema of the target table. Specifically, if the output is incorrect or incomplete, MCTS will backtrack and explore alternative paths.

This abstraction enables modular construction of data preparation pipelines by structuring the synthesis process as a sequence of symbolic actions in DPAS. Rather than treating pipeline generation as an atomic step (e.g., Join), we decompose the process into semantically meaningful sub-decisions, allowing each part to be controlled and revised during the pipeline synthesis process.

Furthermore, to ensure that action sequences in BAT produce reasonable pipelines, we define a set of transformation constraints within the sandbox. These constraints are derived from structural patterns commonly observed in real-world data preparation workflows and specify reasonable transformations between action types. For example, the `CodeRefinement` action should be executed after the `CodeSynthesis` action, and the `Termination` action must be invoked after the pipeline has been synthesized or refined. The complete transformation constraints are summarized in Table 2. Concretely, initial actions at the root node include `SchemaMapping`, `OperatorDiscovery`, and `CodeSynthesis`. The selection of subsequent nodes is subject to the constraints based on the current reasoning state and previously selected actions. These constraints are dynamically enforced during node expansion to prune infeasible branches early in the search process of BAT. By constraining which actions can logically follow others, we prevent inconsistent sequences and guide the BAT toward constructing reasonable and executable DP pipelines.

Discussion. We emphasize that designing a bounded action space via DPAS, rather than directly using the action space composed of atomic transformation operators (atomic action space for short), significantly enhances the pipeline generation process in BAT, as to be detailed in Section 3.2. Recall that BAT is driven by large language models (LLMs) with a tree-structured search mechanism. The search complexity of the tree-structured search method can be denoted as $O(|\mathbb{N}|^{|d|})$, where $|\mathbb{N}|$ is the number of nodes per level in the search tree and $|d|$ represents the depth of the entire tree structure. Considering that data development engineers usually implement custom DP transformation operators through User-Defined Functions (UDFs), the atomic action space exhibits continuous expansion. However, DPAS contains only five abstract operators. Such a constrained design drastically reduces the search complexity. Moreover, our proposed transformation constraints, as illustrated in Table 2, can further reduce the search complexity. In addition, the design of DPAS significantly reduces reasoning complexity for LLMs. The reason is that selecting the correct operators from a small operator set is simpler for LLMs than choosing from a large collection. Furthermore, errors may propagate through operator sequences during the tree-structured search process, and thus fewer abstract operators contained in DPAS enable minimizing the risks of error propagation.

3.2 Fundamental Pipeline Generator

Motivated by the effectiveness of MCTS in solving complex search problems (as illustrated in Section 2.3), we propose the *fundamental pipeline generator* (FPG), a Monte-Carlo-driven search

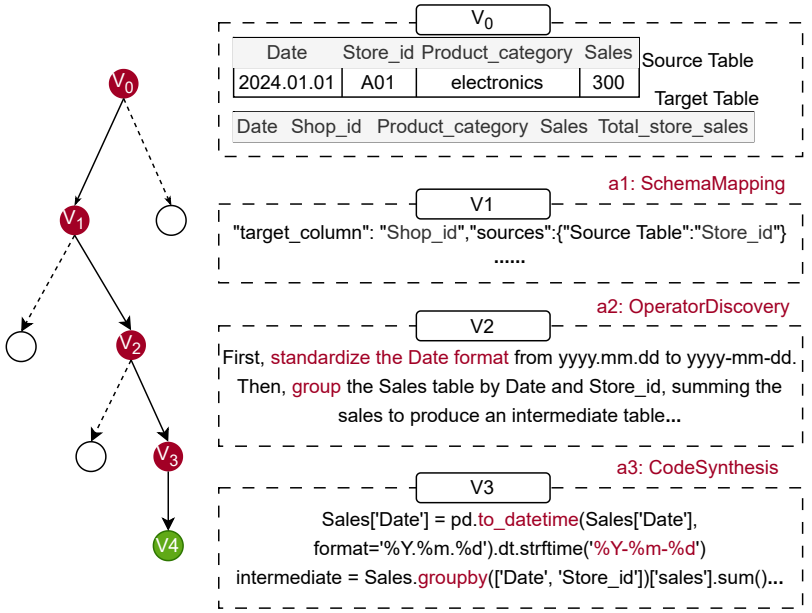


Fig. 4. An example of the tree-structured search process.

approach for ADP, which orchestrates the MCTS process via open-source LLMs and executable feedback effectively.

We formulate the FPG process as a tree-structured search problem defined on a directed tree $\mathcal{T} = (V, E)$, where each node $v_i \in V$ represents a partial reasoning state and each edge $e_i \in E$ means an action defined in DPAS. A complete root-to-leaf path corresponds to a synthesized data preparation pipeline that produces a target-compliant output. This tree-structured formulation enables BAT to perform structured, step-by-step reasoning, starting from symbolic actions of DPAS and ending with concrete executable code. Next, we introduce the settings of both nodes and edges of the search tree utilized in FPG.

Nodes (V) — Reasoning States. Each node $v_i \in V$ represents an intermediate transformation state, capturing the transformation logic constructed up to that node.

Edges (E) — Reasoning Actions. Each edge $e_i \in E$ corresponds to an action decision of DPAS. These include actions such as selecting a schema mapping and choosing a data operator (e.g., `Groupby`, `Join`). Edges transition the state from one node to another and collectively determine a final candidate pipeline.

Figure 4 illustrates an example of ADP as a tree-structured search process. The source table contains four columns, termed as `Date`, `Store_id`, `Product_category`, and `Sales`. The target table includes a renamed column `Shop_id` and a new column `Total_store_sales`, which aggregates daily sales totals for each store. The ADP process is modeled as a search tree, where:

- The root node v_0 represents the initial inputs, i.e., the source table with few-shot data instances and the target table containing only the metadata (i.e., table schema and column descriptions).
- The intermediate node v_1 performs the `SchemaMapping` action to establish correspondences between source and target columns. Next, Node v_2 applies the `OperatorDiscovery` action to identify a sequence of transformation operators, including operations such as `Date Formatting` and `GroupBy`. Due to space limitations, we omit subsequent inferred operators. Then, node v_3 utilizes the `CodeSynthesis` action to generate a data preparation pipeline.

Algorithm 1: Fundamental Pipeline Generator (FPG)

Input: A set of source tables $\mathbb{T}_l$, a target table T_r, an LLM f_{LLM}, a rollout budget N, and a max depth $|d|$

Output: Set of valid transformation pipelines $\mathcal{P}_{\text{found}}$

1 Initialize root node $v_0 \leftarrow (\mathbb{T}_l, T_r)$;
2 Initialize empty pipeline set $\mathcal{P}_{\text{found}}$;
3 **for** $iteration = 1$ **to** N **do**
4 $\quad v_i \leftarrow \textsc{Select}(v_0)$;
5 $\quad C(v_i) \leftarrow \textsc{Expand}(v_i, f_{\text{LLM}})$;
6 $\quad v'_{|d_m|} \leftarrow \textsc{Simulate}(v_i, C(v_i), f_{\text{LLM}})$;
7 $\quad \text{reward} \leftarrow \textsc{Evaluate}(v'_{|d_m|})$;
8 $\quad \textsc{Backpropagate}(v'_{|d_m|}, \text{reward})$;
9 $\quad$ **if** *pipeline is correct via evaluating reward* **then**
10 $\quad\quad \mathcal{P}_{\text{found}} \leftarrow \mathcal{P}_{\text{found}} \cup \textsc{GetPipeline}(v'_{|d_m|})$;
11 **return** $\mathcal{P}_{found}$

- The final node v_4 denotes the termination. At this node, the generated pipeline is assigned a reward.

Thereafter, we are ready to introduce the details of the search process of FPG. Algorithm 1 presents its pseudo-code. It takes as inputs a set of source tables $\mathbb{T}_l$ with few-shot data instances, a target table T_r containing only schema definitions, an LLM f_{LLM}, and a rollout budget N. FPG performs N rollouts to construct and evaluate candidate pipelines. Each rollout consists of four canonical phases, i.e., *Selection*, *Expansion*, *Simulation*, and *Backpropagation*.

Selection. The selection phase begins the MCTS rollout by traversing the current search tree from the root node v_0 toward a leaf or expandable node v_i (Line 4 in Algorithm 1). At each node v, the algorithm determines the next action $a_i \in \mathcal{A}$ and chooses the child node v' by maximizing an acquisition function based on the *Upper Confidence Bounds applied to Trees* (UCT) [26]. Formally:

$$\text{UCT}(v, a) = Q(v_i, a_i) + c \cdot \sqrt{\frac{\log N(v_i)}{N(v_i, a_i)}} \tag{2}$$

Here, $Q(v_i, a_i) \in \mathbb{R}$ is the cumulative reward for taking action a_i at node v_i, where $\mathbb{R}$ denotes the real number space. $N(v_i)$ is the visit count of node v_i, $N(v_i, a_i)$ is the number of times action a_i has been applied at v_i. $c \in \mathbb{R}^+$ is a tunable constant that controls the trade-off between exploration and exploitation. $\mathbb{R}^+$ represents the set of positive real numbers. Notably, if there exists any child node that has not been visited before (i.e., $N(v_i, a_i) = 0$), the algorithm prioritizes selecting such nodes.

This selection strategy encourages the search to favor paths that have historically yielded high reward while also encouraging exploration of less-visited action branches. The logarithmic dependence on $N(v_i)$ ensures diminishing optimism as more rollouts visit the same node, preventing over-exploration of suboptimal paths. The traversal continues recursively until an unexpanded node is reached. This node will be the frontier for subsequent expansion.

Expansion. Once the selection phase identifies a frontier node v_i, the algorithm proceeds to the expansion phase (Line 5 in Algorithm 1). The goal of expansion is to enumerate and instantiate all valid next actions from v_i. This phase produces a candidate set of child nodes for the subsequent simulation phase.

Instruction	You are a highly meticulous and intelligent schema matcher specialized in data preparation. Your task is to analyze a set of **source tables** and a **target table**, and identify how each column in the target table semantically corresponds to column(s) in the source table(s).
Tips	The target table may be derived from the source tables using operations such as **join**, **union**, **groupby**, **pivot**, **unpivot**, **rename**, **column arithmetic**, **date formatting**, **adding or dropping columns**. A target column may: - Correspond to one or more columns from a single source table, - Be the result of combining or aggregating columns, - Be derived from columns across multiple source tables (e.g., through a join or union). .....
Table info	Table Schema: {table_schema_dict}
Output Format	Please respond with your reasoning process and a JSON object structured as follows: ```json [{{ "target_column": "target_column_name1", "sources": {{"table_name1": ["column1", "column2", ...]}}, }}]``` .....

Fig. 5. An example of an action prompt template.

Let $\mathcal{A}_{\text{valid}}(v_i) \subseteq \mathcal{A}$ be the set of admissible action types at node v_i, constrained by the transformation constraints (as detailed in Table 2). For each action type $A_j^i \in \mathcal{A}_{\text{valid}}(v_i)$, the algorithm queries the LLM to generate its parameters:

$$\theta_i = f_{\text{LLM}}(\textsc{Prompt}(v_i, A_j^i)) \tag{3}$$

where θ_i is a structured object that specifies the action's arguments (e.g., join keys, aggregate functions). As illustrated in Figure 5, we design a standardized action prompt template to facilitate this generation. Taking the `SchemaMapping` action as an example, the prompt begins with a detailed instruction that describes the task. Then, there are the carefully designed tips that guide the model in identifying semantic correspondences between source and target columns. This is followed by information about the table. To ensure the generated output is syntactically valid and aligned with downstream processing, the prompt concludes with an explicit specification of the expected response format.

Each parameterized action is defined as a tuple $a_i = (A_j^i, \theta_i)$. Applying a_i to the current node produces a successor state $v_i' = \delta(v_i, a_i)$, where δ is the state transition process. All generated child nodes $\{v_1', v_2', \ldots v_{|v'|}'\}$ are collected in a candidate set:

$$C(v_i) = \left\{ \delta(v_i, a_i) \;\middle|\; a_i = (A_j^i, f_{\text{LLM}}(\textsc{Prompt}(v_i, A_j^i))),\ A_j^i \in \mathcal{A}_{\text{valid}}(v_i) \right\} \tag{4}$$

The collected candidates $C(v_i)$ will be passed to the subsequent simulation phase for evaluation.

Simulation. In the simulation phase (Line 6 in Algorithm 1), the algorithm randomly selects a node v_i' from the candidate set $C(v_i)$. Then it recursively applies actions to expand the selected

node until a termination node is reached or the maximum rollout depth $d_{\max}$ is exceeded. The result of this simulation is a sequence of nodes, which represents a candidate path as follows:

$$Path = \{v'_i, v'_{i+1}, \ldots, v'_{|d_m|}\}, \quad \text{with} \quad v'_{i+1} = \delta(v'_i, a_i) \tag{5}$$

Here, $|d_m|$ denotes the m-th depth of the search tree, and $|d_m| \leq |d|$. δ is the state transition process as defined in the expansion stage.

At the end of the simulation phase, the terminal node $v'_{|d_m|}$ typically contains a complete data preparation pipeline $\mathcal{P}$. An intuitive approach for assessing the quality of such a pipeline is to take advantage of the reasoning capabilities of LLMs. To this end, we introduce a self-reward mechanism in which the LLM acts as a judge. Specifically, given a set of source tables $\mathbb{T}_I$, the target table T_r, and the generated pipeline $\mathcal{P}$, the LLM is prompted to evaluate whether $\mathcal{P}$ is likely to produce a correct transformation. Based on its reasoning, the LLM returns a scalar reward $r_{\text{LLM}} \in \{0, 0.5, 1.0\}$. The evaluation criteria of the self-reward mechanism are as follows:

- $r_{LLM} = 1$ if the program is fully correct and complete;
- $r_{LLM} = 0.5$ if the program is partially correct but incomplete;
- $r_{LLM} = 0$ if the program is malformed or incorrect.

Designing an effective self-reward mechanism is challenging. Prior studies [47] have demonstrated that prompts often led to unreliable evaluations. For instance, LLMs might assign high rewards to programs that are syntactically correct but semantically incorrect.

To address this, we design a structured reward prompt composed of four main parts. The first part, i.e., instruction, defines the model's role as a data preparation expert and clearly states its evaluation task. The second part, i.e., evaluation rules, introduces a clear reward rubric with three levels—exact match, partially correct, and incorrect—helping the model judge program quality. The third part, i.e., table information, provides the source table, target table schema, and generated pipeline. The final part, output format, specifies the expected structure of the model's response. This structured prompt reduces ambiguity and solves the problems faced above.

Backpropagation. Following the simulation phase, the final reward r_{LLM} assigned to the pipeline $\mathcal{P}$ is propagated back along the search path. The goal of this phase (Line 7 in Algorithm 1) is to update value estimates and visitation counts for all intermediate node–action pairs encountered in the current rollout. These updates help refine the UCT-based search policy for future iterations. Let the search path be denoted by a sequence of state–action pairs:

$$\Pi = \{(v'_0, a_0), (v'_1, a_1), \ldots, (v'_{|d_m|}, a_{|d_m|})\} \tag{6}$$

where each $v'_i \in V$ is a node, and $a_i = (A^i_j, \theta_i)$ is the applied action at that node. For each pair $(v'_i, a_i) \in \Pi$, we maintain the following three statistics: (i) $Q(v'_i, a_i)$ is the cumulative reward for taking action a_i at node v'_i; (ii) $N(v'_i)$ is the number of times node v'_i has been visited; and (iii) $N(v'_i, a_i)$ is the number of times action a_i has been taken from v'_i. We update these quantities as follows:

$$Q(v'_i, a_i) \leftarrow Q(v'_i, a_i) + r \tag{7}$$

$$N(v'_i) \leftarrow N(v'_i) + 1 \tag{8}$$

$$N(v'_i, a_i) \leftarrow N(v'_i, a_i) + 1 \tag{9}$$

These updates ensure that nodes along promising paths—i.e., those associated with higher reward values—are more likely to be revisited in subsequent rollouts, while underperforming branches are naturally downweighted. The revised statistics directly impact the UCT score in Equation (2), enabling adaptive exploration over the search space in ADP tasks.

Algorithm 2: Execution-aware Pipeline Optimizer (EPO)

Function `ExecutionAwareReward`$(\mathcal{P}, \mathbb{T}_I, T_r)$**:**
  $\hat{T}_r \leftarrow$ `Execute`$(\mathcal{P}, \mathbb{T}_I)$;
  reward $\leftarrow$ `sim`$(\hat{T}_r, T_r)$;
  return *reward*;
Function `SimulationCache`$(v_i, A^i_j, \mathcal{M}, f_{LLM})$**:**
  if $(v_i, A^i_j) \in \mathcal{M}$ **then**
    return $\mathcal{M}[(v_i, A^i_j)]$;
  $\theta_i \leftarrow f_{\text{LLM}}$(`PROMPT`$(v_i, A^i_j)$);
  $\mathcal{M}[(v_i, A^i_j)] \leftarrow \theta_i$;
  return θ_i;
Function `EarlyTermination`$(\mathcal{P}, \mathcal{P}_{found}, K, \mathbb{T}_I, T_r)$**:**
  if *pipeline is correct via evaluating reward* **then**
    $\mathcal{P}_{\text{found}} \leftarrow \mathcal{P}_{\text{found}} \cup \mathcal{P}$;
    if $|\mathcal{P}_{found}| \geq K$ **then**
      return `True`;
  return `False`;

3.3 Execution-aware Pipeline Optimizer

The fundamental pipeline generator builds data preparation pipelines and scores them using a self-reward strategy. This approach treats the LLM as a self-evaluator: given the source tables $\mathbb{T}_I$, the schema S_r from the target table T_r, and the generated pipeline $\mathcal{P}$, the model infers a scalar reward that reflects its confidence in the pipeline's correctness. To further enhance the framework, particularly in scenarios where execution signals are available, we develop the *execution-aware pipeline optimizer* (EPO). Specifically, EPO includes two key optimizations: (i) execution-aware reward optimization, and (ii) search accelerator via simulation cache and early termination. These enhancements are modular and apply during both the expansion and simulation phases of MCTS. A complete description of the procedure is provided in Algorithm 2.

Execution-aware reward optimization. Recall that the fundamental pipeline generator (FPG) involves a self-reward strategy to estimate the correctness of a pipeline $\mathcal{P}$. This reward is solely based on LLM's reasoning ability. It may give incorrect results due to hallucination issues. This is especially common in multi-step pipelines, where small logical errors are easy to miss [16]. Therefore, we incorporate execution feedback into the reward calculation, referring to it as the *execution-aware reward* (line 1 in Algorithm 2). As mentioned earlier, BAT generates a DP pipeline $\mathcal{P}$ based on the current search state. Once a complete pipeline is synthesized, it can be executed to output a table $\hat{T}_r$. Formally, $\hat{T}_r = \textsc{Execute}(\mathcal{P}, \mathbb{T}_I)$. We then compute the similarity between $\hat{T}_r$ and T_r based on their schema similarity. Formally, $sim(\hat{T}_r, T_r) = \frac{|\hat{S}_r \cap S_r|}{|S_r|}$, where $\hat{S}_r$ denotes the set of column names in the output table $\hat{T}_r$, and S_r denotes the set of column names specified in the target table T_r. Specifically, the schema similarity metric evaluates both column names and data types, and penalizes pipelines resulting in type mismatches (e.g., String vs. Integer), thereby ensuring the robustness of the execution-aware reward optimization.

This reward provides a grounded and deterministic signal that reflects whether the pipeline produces valid outputs. As to be verified by our experimental study in Section 4.3, we find that

the execution-aware reward consistently outperforms the self-reward, especially under noisy or ambiguous conditions.

Search accelerator. To improve the search efficiency of FPG, we introduce two strategies, namely *simulation cache* (line 5 in Algorithm 2) and *early termination* (line 11 in Algorithm 2), respectively. The simulation cache maintains a parameter prediction cache $\mathcal{M}$. It stores the result of each LLM call for a context $(v_i, A_j^i) \in V \times \mathcal{A}$. In this way, when the same context is encountered in future rollouts, the cached parameter θ_i is reused, circumventing redundant LLM inference calls. Formally:

$$\theta_i = \mathcal{M}[(v_i, A_j^i)] \quad \text{if cached, else } \theta_i = f_{\text{LLM}}(\textsc{Prompt}(v_i, A_j^i)) \tag{10}$$

Furthermore, to avoid unnecessary exploration once the search has produced sufficient successful candidates, EPO incorporates an early termination criterion. Let $\mathcal{P}_{\text{found}}$ denote the set of pipelines with reward $r = 1$. If the number of such pipelines exceeds a user-defined threshold K, the search process is halted.

Together, simulation cache and early termination form a pair of synergistic strategies. They collectively reduce redundant exploration in automatic data preparation tasks, leading to a more efficient and targeted search process.

Discussion. Recall that we have proposed two reward strategies, namely, the self-reward strategy and the execution-aware reward strategy. Alternatively, another reward strategy is the *hybrid reward strategy*, which combines the rewards from both execution feedback and the LLM itself. However, we observe that this strategy achieved an intermediate performance between the execution-aware reward strategy and the self-reward strategy. More detailed experimental results can be found in Section 4.3. This is due to the inherent instability of LLM. Even when informed by execution feedback, the reward judgments of LLM cannot always guarantee correctness. Therefore, we adopt the proposed execution-aware reward in BAT.

Schema similarity is insufficient to guarantee strict pipeline correctness. However, strict correctness verification is infeasible in our "target-instance-free" task. Consequently, the primary objective shifts from finding a strictly verified solution to identifying the most semantically plausible approximate solution to minimize subsequent human intervention, as detailed in Section 1. Adopting the insight from [50], which illustrates that implicit constraints are often sufficient to effectively guide the possible pipeline generation in the absence of examples, BAT constructs a *composite implicit constraint* that synergizes schema similarity with LLM semantic reasoning based on source table data and target metadata. This synergy filters out mismatches and hallucinations, ensuring pipeline reliability in most cases. Moreover, as shown in Section 4.3, we conduct an idealized experiment where target instances are accessible for reward computation. The results show only a marginal gap between instance-level rewards and schema-similarity-based rewards. This demonstrates that, when coupled with LLM-driven reasoning, schema similarity is sufficient in most cases.

4 Experiments

Datasets. Due to the sensitive nature of enterprise data and proprietary business logic, public datasets for this new task are scarce. Consequently, following recent SOTA works [42, 50], we utilize two real-world ADP benchmarks currently available in the public domain. The statistics are summarized in Table 3.

- **Auto-Pipeline Benchmark**. This benchmark was introduced by [50]. It features multi-step pipeline synthesis using 700 ADP tasks from GitHub and industries. Each task includes a target schema specified in a 'by-target' format, along with up to 10 transformation steps.
- **Smart Building Benchmark**. This dataset is originally released in [42], which aims to build DP pipelines for energy data. It comprises 105 real-world tasks involving complex structural transformations (such as aggregation, pivot, and column-wise computation). In this dataset,

Table 3. Statistics of the datasets used in experiments.

Benchmark	# of pipelines	avg. # of input files	avg. # of input cols	avg. # of input rows
Auto-Pipline	481	3.4	8.3	8446.4
Smart Building	105	1	4.9	28.1

some target tables contain columns that are not relevant to the source tables directly. This better reflects real-world scenarios, where target schemas often include extraneous or system-specific fields beyond the scope of available data.

Evaluation Metrics. To evaluate the effectiveness of synthesized pipelines, we use two evaluation metrics as follows:

- **Execution Accuracy (EX).** Recall that our goal is to generate an executable and correct DP pipeline rather than a candidate pipeline. The former can actually inject data from sources into the specific target table, while the latter may generate a new table that is semantically similar to the target table (as described in Section 2.1). To this end, we define EX as a strict indicator of end-to-end correctness. Given a generated pipeline $\mathcal{P}$, we execute $\mathcal{P}(\mathbb{T}_I)$ to produce an output table $\hat{T}_r$, and compare it with the ground truth, i.e., target table T_r. A result is marked as correct (EX = 1) only if $\hat{T}_r$ is identical to T_r in both schema and data instances. This evaluation goes beyond schema alignment to ensure the correctness of the pipeline. Note that the data instances of T_r are used solely during evaluation; the generation process has no access to target instances. The strict equality comparison rules out false positives from type mismatches.
- **Column Similarity (CS).** Similar to the previous study [42], we define CS to measure schema-level similarity between the predicted output table $\hat{T}_r$ and the ground-truth target table T_r. A column pair is considered matched if its name exactly matches a column in the target schema. This metric is particularly useful for identifying partially correct pipelines. For example, a pipeline may fail to perform a group-by aggregation but still recover correctly named columns such as `Date` or `Store_id`. In such cases, CS highlights a possible meaningful pipeline even when the full output is incorrect (i.e., EX = 0).

To evaluate efficiency, we report the execution time (in minutes) for each ADP approach.

Implementation Details. We detail the hyperparameters used in BAT as follows. We adopt the open-source Qwen-Coder family (i.e., 7B, 14B, and 32B) primarily due to its high resource efficiency and deployment feasibility. Unlike larger proprietary models that incur prohibitive costs, Qwen-Coder delivers competitive performance with significantly lower computational overhead, making it ideal for our pipeline generation task (with Qwen-Coder-32B as the default). Also, much larger LLMs, including GPT-OSS-120B and DeepSeek-V3.2, are employed to evaluate the generalizability of BAT. In the fundamental pipeline generator (FPG), we configure MCTS with the rollout budget $N = 10$ to ensure sufficient exploration. The maximum search depth $|d|$ is set to 5, aligning with the number of actions defined in our DP Action Sandbox (DPAS). The exploration constant c is set to 1, as our sensitivity analysis in Section 4.3 confirms that variations in c do not significantly impact the overall quality of the synthesized pipeline. In the execution-aware pipeline optimizer (EPO), the early termination threshold is set to $K = 2$, as it achieves the best balance between accuracy and efficiency (see Section 4.3 for detailed analysis). We adopt the execution-aware reward strategy to evaluate pipeline correctness. Unless explicitly specified, all hyperparameters are set to their default values. In addition, though the BAT framework is designed to be a local deployment, due to limitations in our experimental environment, we use the Qwen models via API calls. Importantly, all models used are open-source and self-hostable, ensuring that

our implementation is consistent with the setting mentioned before. All experiments are conducted on a Linux Server with an Intel(R) Xeon(R) Silver 4310 CPU (2.10GHz) and 128GB RAM. The programs were all implemented in Python. All details of the full prompts and the source code of BAT can also be found in https://github.com/ZJU-DAILY/BAT.

Methods Compared. We compare BAT against five representative methods as follows:

- **SQLMorpher** [42]. SQLMorpher is an end-to-end LLM-based ADP system. It constructs task-specific prompts (optionally incorporating data hints) and iteratively refines the generated transformations based on the SQL execution feedback. In our evaluation, we *do not use any data hints* in SQLMorpher, as the provision of data hints often relies on domain experts, which contradicts the goal of self-service data preparation, as discussed in Section 1. This setting ensures a fair comparison with other methods that operate purely on the source table and target schema.
- **Chain-of-Table** [48]. This approach decomposes complex transformations into a sequence of sub-tasks. It iteratively updates an intermediate table by applying one transformation at a time rather than generating an entire program at once. To ensure a fair and comprehensive comparison, we adapt the Chain-of-Table baseline by extending its operator library to support all of the transformation operators defined in BAT (Section 2.2).
- **Program-of-Thoughts**(PoT) [5]. PoT is a paradigm that separates reasoning from computation by prompting language models to generate executable code steps instead of relying solely on natural language reasoning. Specifically, we adapt PoT to the ADP context by prompting the LLM to express the transformation logic as executable Python programs (e.g., using pandas operators). This allows the baseline to ground its reasoning in actual execution, making it a robust competitor for synthesizing multi-step data preparation pipelines.
- **ReAct** [52]. ReAct integrates reasoning and acting by interleaving plans with actions and observations. In the context of ADP, this approach allows the LLM to reason over the execution results and refine its generated pipeline based on those results. By incorporating inspection of the transformation results, ReAct improves execution correctness in ADP pipelines.
- **FunctionCalling** [40]. FunctionCalling constrains the LLM's action space by exposing a predefined set of transformation functions via structured JSON. Although it is developed for general tool-augmented tasks, its modular design makes it an appropriate method for ADP, where transformations can be expressed as well-typed function calls. Therefore, we replace the original function library with the DP transformation operators, as illustrated in Section 2.2. This enables FunctionCalling to be applied for solving the task of ADP.

4.1 Main Results

Table 4 summarizes the overall results on both the Auto-Pipeline and Smart Building benchmarks.

Effectiveness. First, we observe that BAT consistently outperforms all SOTA competitors in terms of both execution accuracy (EX) and column similarity (CS). More specifically, it achieves up to 4.85× improvements in EX. This demonstrates that our tree-structured search process, coupled with execution-aware optimization, facilitates generating more reliable DP pipelines. Second, we also observe that the performance of BAT is positively correlated with the model size of LLM. BAT achieves the best performance using the LLM with 32B parameters. This indicates that larger LLM models provide stronger reasoning capability in the ADP task. Note that BAT-14B also delivers competitive performance with higher CS than other competitors. These results show that BAT can significantly enhance the capabilities of LLM in ADP tasks, even at smaller scales. Third, among the baselines, ReAct shows strong performance, particularly on the Auto-Pipeline benchmark. Its ability to interleave reasoning and acting contributes to improved effectiveness in ADP tasks. FunctionCalling performs poorly across both benchmarks, especially on Smart Building (EX =

Table 4. Overall results on Auto-Pipeline and Smart Building benchmarks.

Method	LLM Model	Auto-Pipeline Benchmark			Smart Building Benchmark		
		EX(%)	CS(%)	Time(min)	EX(%)	CS(%)	Time(min)
SQLMorpher	Qwen2.5-Coder-32B	71.46	75.22	0.50	<u>55.45</u>	58.54	0.42
Chain-of-Table	Qwen2.5-Coder-32B	56.13	58.77	0.12	43.81	48.03	**0.16**
PoT	Qwen2.5-Coder-32B	77.03	79.97	**0.09**	51.43	56.55	<u>0.20</u>
ReAct	Qwen2.5-Coder-32B	79.83	84.40	<u>0.10</u>	51.43	60.47	0.22
FunctionCalling	Qwen2.5-Coder-32B	18.27	50.29	0.28	19.61	30.08	0.40
BAT-32B	Qwen2.5-Coder-32B	**88.57**	**89.69**	1.16	**68.57**	**84.67**	3.24
BAT-14B	Qwen2.5-Coder-14B	<u>82.95</u>	<u>85.74</u>	1.20	48.57	<u>60.84</u>	3.17
BAT-7B	Qwen2.5-Coder-7B	47.19	49.80	1.47	17.14	35.19	4.34

[1] We report Execution Accuracy (EX↑), Column Similarity (CS↑), and average runtime (Time ↓, in minutes).
[2] **Bold** indicates current optimal values, while <u>underscored</u> values are suboptimal.

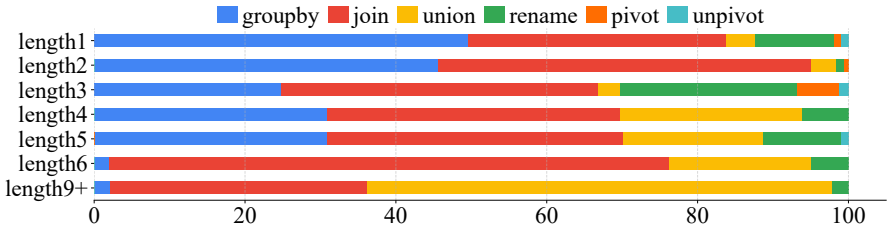


Fig. 6. Operator distribution of Auto-Pipeline benchmark.

19.61%). This result suggests that a function invocation scheme is insufficient for complex multi-step reasoning tasks. In addition, all methods, including BAT, perform better on Auto-Pipeline than on Smart Building. This is due to the characteristics of the Smart Building benchmark. It contains ambiguous and irregular schema structures. In addition, they also include schema noise in the target table. Such characteristics increase the difficulty of identifying correct DP pipelines. All methods struggle in this setting, especially when no external hints are provided.

Furthermore, we evaluated the effectiveness of each method on pipeline generation tasks with varying difficulty levels. We follow [50] to use *pipeline length* to indicate the difficulty of pipeline generation. Longer length denotes a more challenging task. Note that the datasets in the Smart Building benchmark can not be grouped by pipeline length and are therefore excluded from this analysis. Figure 7 shows the corresponding results. It is observed that BAT-32B maintains strong performance across most lengths, with a decline at length 9+. In contrast, SQLMorpher performs relatively well on length 9+ tasks, due to its strength in handling union operations, which are more common in longer pipelines. However, it performs poorly on shorter pipelines that require more precise and diverse transformations, such as GroupBy, Rename, and Pivot. This observation is consistent with the operator distribution shown in Figure 6. Here, short pipelines are dominated by Join and GroupBy, whereas complex pipelines (length 9+) exhibit a surge in Union operations. These results suggest that SQLMorpher performs well in specific scenarios but lacks general versatility. Overall, BAT demonstrates more stable and robust performance across varying levels of length.

Efficiency. We now turn our attention to the evaluation of efficiency, with results shown in Table 4. We can observe that the runtime of both BAT and the competitors can be completed in minutes. However, the runtime of BAT is slightly longer than the other competitors. Notably, although the running time of BAT takes a little bit longer, BAT achieves up to 4.85× improvements in the

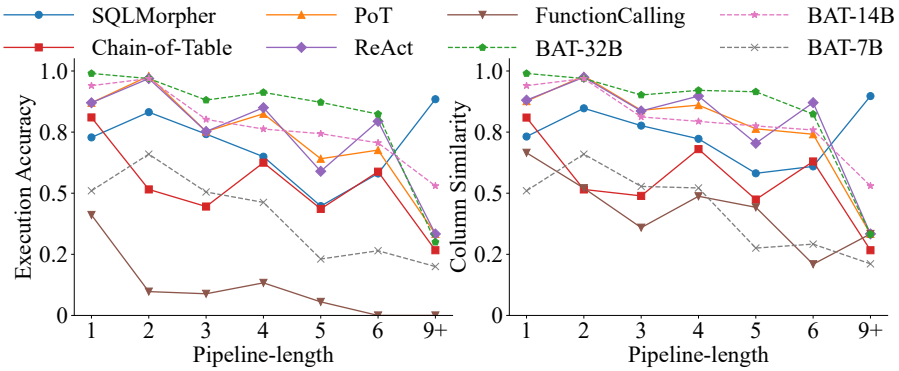


Fig. 7. Execution accuracy (EX) and column similarity (CS) results vs. different pipeline length.

Table 5. Effect of simulation cache (SC) and early termination (ET) on EPO, by reporting execution accuracy (EX), column similarity (CS), and running time.

Configuration	Auto-Pipeline Benchmark			Smart Building Benchmark		
	EX(%)	CS(%)	Time(min)	EX(%)	CS(%)	Time(min)
w/ SC & ET	88.57	89.69	1.16	68.57	84.67	3.24
w/o SC & ET	86.69	87.55	1.87	69.38	82.39	4.75
w/o SC	88.36	88.87	1.40	69.52	84.24	4.12
w/o ET	87.44	89.54	1.66	68.57	85.39	3.91

end-to-end data preparation pipeline execution accuracy. We want to emphasize that this efficiency-accuracy trade-off is worthwhile. In enterprise systems, DP pipelines usually run weekly, daily, or hourly [22]. Therefore, a pipeline generation time in minutes is sufficient for most DP scenarios. Under such a condition, the improvement of accuracy is particularly critical in practice.

4.2 Ablation Study

We conduct ablation studies to evaluate the impact of two critical components in BAT, including the *execution-aware pipeline optimizer* (EPO) and the *transformation constraints* within the DPAS.

Impact of EPO components. First, we evaluate the effectiveness of EPO, which consists of two components: *simulation cache* (SC) and *early termination* (ET). We compare four configurations: (i) with both SC and ET enabled (w/ SC & ET), (ii) with both disabled (w/o SC & ET), (iii) with only SC disabled (w/o SC), and (iv) with only ET disabled (w/o ET). Table 5 reports the results.

As can be observed, removing either SC or ET results in increased runtime on all benchmarks. This verifies that both SC and ET contribute significantly to the efficiency of BAT. In particular, the absence of the simulation cache leads to redundant LLM queries during the tree-structured search process, as parameter predictions must be recomputed for the same node states. Similarly, without early termination, the search continues exploring even after high-quality pipelines have already been found. Besides, we can observe that both EX and CS vary slightly across different configurations. Though there are some fluctuations, the variation in EX does not exceed 1%, and the CS differences are minor. These results demonstrate that EPO successfully improves efficiency through SC and ET components without materially compromising pipeline quality.

Table 6. Effect of transformation constraints.

Setting	Auto-Pipeline Benchmark			Smart Building Benchmark		
	EX(%)	CS(%)	Time(min)	EX(%)	CS(%)	Time(min)
Constrained (BAT-32B)	**88.57**	**89.69**	**1.16**	68.57	**84.67**	**3.24**
Unconstrained (BAT-32B)	88.44	89.19	5.87	**69.52**	81.80	10.17

Table 7. Effect of reward strategy.

Benchmark	Reward strategy	EX(%)	CS(%)	Time(min)
Auto-Pipeline	Self reward	83.78	86.29	1.27
	Execution-aware reward	**88.57**	**89.69**	**1.16**
	Hybrid reward	87.94	88.64	1.29
Smart Building	Self reward	44.76	54.34	3.49
	Execution-aware reward	**68.57**	**84.67**	**3.24**
	Hybrid reward	59.05	69.67	3.54

Effect of transformation constraints. Second, we examine the role of transformation constraints in DPAS by comparing BAT with an unconstrained variant in which these constraints are removed. As shown in Table 6, removing transformation constraints has a negligible effect on the quality of the synthesized pipelines. However, the unconstrained variant incurs a substantial efficiency penalty, with runtime increasing by approximately 5× on Auto-Pipeline and 3× on Smart Building, respectively. This slowdown stems from the significantly expanded search space, which introduces many redundant or semantically invalid transformation sequences. These results demonstrate that transformation constraints serve as effective pruning heuristics in DPAS.

4.3 Further Analysis

We present further analysis to provide deeper insight into the behavior, robustness, and limitations of BAT. Specifically, we examine: (i) the effect of different reward strategies during search; (ii) the reliability of BAT compared to its idealized variant; (iii) the effect of the early termination threshold on search efficiency and quality; (iv) the sensitivity of BAT to the exploration constant in MCTS; (v) the generalizability of BAT across diverse LLM models; (vi) the scalability of BAT with respect to table size and table number; and (vii) failure patterns through detailed error analysis.

Effect of reward strategies. To investigate the impact of different reward strategies on the execution-aware pipeline optimizer EPO, we compare the following three reward strategies: the self-reward strategies, the execution-aware reward strategy, and the hybrid reward one. Detailed information can be found in Section 3.3. The results are summarized in Table 7. First, we observe that the execution-aware reward consistently outperforms the other two strategies in terms of both execution accuracy and column similarity. These results confirm that incorporating execution feedback provides a more reliable and precise evaluation of pipeline correctness. Second, as can be observed, the hybrid reward strategy performs better than the self-reward one. This suggests that combining LLM reasoning with execution feedback facilitates more reliable pipeline generation. The feedback helps correct potential mistakes that the LLM might overlook when reasoning alone. Third, strategies that rely on LLMs, such as hybrid reward and self-reward, generally take longer runtimes. This is because they involve additional inference steps during evaluation.

Reliability of BAT under metadata-only setting. We evaluate the robustness of our approach by comparing it with an "idealized" instance-aware setting. We denote our method as BAT w/o instance (metadata-only), and introduce an idealized variant, BAT w/ instance (Ideal), where target instances

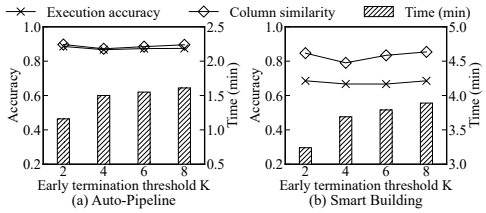


Fig. 8. Effect of early termination threshold.

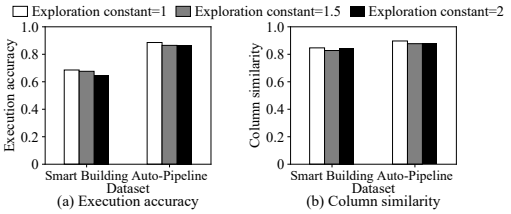


Fig. 9. Effect of exploration constant.

are assumed to be available for reward computation. As depicted in Figure 10, the performance gap remains remarkably narrow across different LLM models. This observation empirically confirms that the semantic implicit constraints utilized in our framework effectively compensate for the absence of instance-level feedback, validating BAT's reliability in metadata-only environments.

Effect of early termination threshold. We investigate the impact of the early termination threshold K on EPO. As shown in Figure 8, increasing K first leads to a drop in performance, followed by a recovery. This trend is more pronounced on the Smart Building benchmark, while the Auto-Pipeline benchmark remains relatively stable. The initial performance drop occurs because the expanded search process includes lower-quality pipeline candidates. Performance eventually recovers as the search explores more options and identifies higher-quality candidates – but at a significant runtime cost. Therefore, we set $K = 2$ as the default value, as it achieves the best balance between accuracy and efficiency.

Effect of exploration constant. We further investigate the effect of the exploration constant c of MCTS for the fundamental pipeline generator (FPG) by using three settings: 1, 1.5, and 2. As shown in Figure 9, execution accuracy and column similarity exhibit minor fluctuations as the constant increases, but the variation remains within a narrow range. In summary, the exploration constant does not significantly affect the overall pipeline quality.

Generalizability across diverse LLMs. To address concerns regarding whether the performance of BAT is specific to the Qwen-Coder family, we extended our evaluation to include two other state-of-the-art open-source LLMs: GPT-OSS-120B and DeepSeek-V3.2. As summarized in Table 8, BAT consistently achieves the highest execution accuracy across both model architectures on both benchmarks. These findings confirm that the effectiveness of our proposed tree-search framework and execution-aware optimization is model-agnostic and robust across different LLM families.

Effect of table size and number. We focus our evaluation on the Auto-Pipeline benchmark, as it exhibits substantial variance in both table number and schema complexity. The Smart Building benchmark is excluded, as it focuses on single-table tasks with column sizes concentrated in a narrow range and therefore does not reflect scalability. Here, *table size* is defined as the number of columns rather than row counts. This is because our metadata-driven synthesis relies on schema information and few-shot samples; row counts do not impact the LLM's reasoning complexity. To ensure rigorous comparison, we group tasks into five balanced intervals.

The results in Figure 11 reveal the following two key insights. (i) **Impact of table number**: BAT exhibits a modest performance decline as the number of tables increases, a trend expected given the escalating structural complexity. However, performance stability varies by model size: the fluctuations observed in smaller models (i.e., BAT-7B and BAT-14B) are attributable to their limited capacity in handling complex multi-table reasoning. (ii) **Impact of table size**: As can be observed, while smaller models exhibit fluctuations due to the aforementioned limited capacity, larger models show only minor performance variations across different column counts, indicating a weak correlation between execution accuracy and schema width. Notably, the lower performance

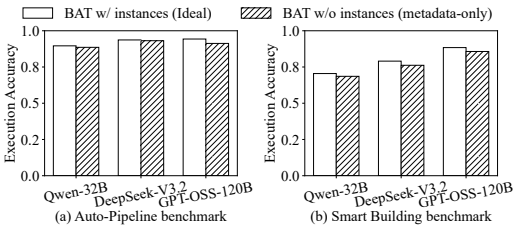


Fig. 10. BAT w/ instances vs. BAT w/o instances.

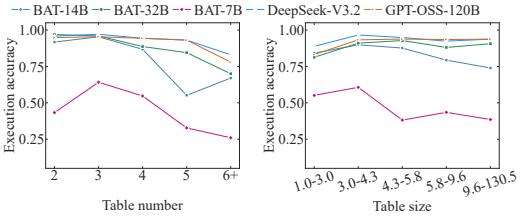


Fig. 11. Effect of table number and table size.

Table 8. Generalizability analysis.

Method	Auto-Pipeline Benchmark			Smart Building Benchmark		
	EX(%)	CS(%)	Time(min)	EX(%)	CS(%)	Time(min)
Model: GPT-OSS-120B						
SQLMorpher	91.14	**93.77**	0.38	70.19	71.05	0.36
Chain-of-Table	80.87	81.00	0.12	67.62	71.04	0.25
PoT	88.98	89.31	**0.07**	73.33	78.96	0.18
ReAct	90.44	90.97	0.09	77.14	84.76	0.21
FunctionCalling	27.74	56.30	0.15	55.21	63.92	**0.12**
BAT (Ours)	**91.27**	91.27	0.54	**85.71**	**90.58**	1.02
Model: DeepSeek-V3.2						
SQLMorpher	89.41	**94.36**	0.38	74.76	76.24	0.36
Chain-of-Table	84.41	85.28	0.12	55.24	59.50	0.25
PoT	88.15	89.31	**0.07**	63.81	69.37	0.18
ReAct	88.77	90.28	0.09	74.29	82.15	0.21
FunctionCalling	28.71	54.28	0.15	22.30	33.25	**0.12**
BAT (Ours)	**93.14**	93.28	0.54	**76.19**	**82.47**	1.02

observed when table sizes fall within the 1–3 range is attributed to heightened semantic ambiguity. Tasks with fewer columns often lack sufficient semantics, but they may still require intricate transformations. Indeed, we would like to emphasize that the performance of pipeline synthesis is governed by semantic complexity rather than table size.

Error Analysis. To gain deeper insights into why certain pipelines fail, we conducted a comprehensive analysis of the failed cases on the Auto-Pipeline dataset. As shown in Figure 12, BAT-32B, ReAct, and PoT exhibit relatively balanced error distributions across major operators. Their failures are concentrated mainly in Join, GroupBy, and Union. It is primarily associated with intrinsically difficult pipelines involving complex interactions among multiple operators. In contrast, SQLMorpher, Chain-of-Table, and FunctionCalling display more skewed error distributions. For these methods, Join and GroupBy account for a disproportionately large share of failures, while Union contributes a comparatively smaller fraction. This pattern suggests that these approaches are more robust to structural schema alignment, as required by Union, but struggle with semantic reasoning. Across all methods, Pivot, Unpivot, and Rename contribute minimally to the total error count, largely due to their low frequency in the benchmark.

5 Related Work

ADP methods. There is a large body of prior ADP work. Transform-by-example (TBE) [1, 11, 13–15, 23, 24, 27, 43] enables users to specify desired data transformations by providing one or more illustrative examples. Transform-by-pattern [25] seeks to discover reusable transformation logic by

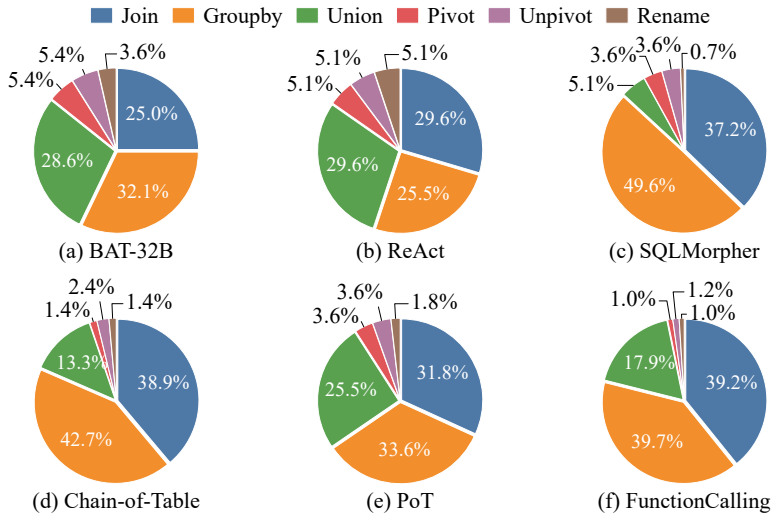


Fig. 12. Failure cases on the Auto-pipeline dataset.

analyzing large corpora of table columns and their structural patterns. Transform-for-joins [31, 54] focuses on automatically generating equi-joins between tables with semantically matching, but textually differing, key columns. Transform-by-target [49, 50] focuses on synthesizing end-to-end data pipelines that transform raw input tables into a predefined target table, leveraging functional dependencies and keys to guide the transformation process. The methods above synthesize transformation operators guided by the provided data examples, which heavily rely on direct access to the target table instances. Other methods, such as Auto-Tables [30] and Auto-Prep [29], utilize target table instances for training reliable ADP models. However, these supervised methods require a proportional amount of training data (e.g., Auto-Pipeline and AutoPrep require $\geq 50\%$ data), not few-shot examples, which is impractical in most real-world scenarios. Furthermore, relying on even 'few-shot examples' violates the strict privacy constraint in cross-department collaboration in our task, rendering these methods inapplicable. In summary, these studies either require target instances or involve labor-intensive training efforts, in contrast to the task addressed in this paper.

More recently, both academia and industries utilize LLMs to solve ADP problems, allowing data preparation without labor-intensive efforts, such as collecting supervision signals. SQLMorpher [42] tackles the ADP problem by prompting LLMs to generate SQL transformation scripts. Its prompt incorporates attribute descriptions and external data hints. It is the most comparable to our work in terms of both objective and formulation. However, our proposed BAT requires *zero* external knowledge for the guidance of pipeline synthesis, and achieves more comprehensive performance than that of SQLMorpher, as verified in the experiments in Section 4.1. In addition, ChatPipe[4] and Text-to-Pipeline[9] explore a different direction – focusing on interactive, language-driven pipeline synthesis. Similarly, Palantir Foundry, the famous operating system for the modern enterprise, provides a pipeline builder assistant, which leverages LLMs to fulfill user-specified single-step DP transformation requirements [21]. These approaches rely on explicit user intent. Thus, they are orthogonal to our problem setting.

LLM-based multi-step reasoning methods. In recent years, LLM-based multi-step reasoning methods have been widely applied to handle complex multi-step tasks [3, 7, 39, 51]. Although these methods were not specifically proposed for solving the ADP problem, they can be readily applied

to ADP, given its nature as a multi-step task. We now introduce several representative and state-of-the-art methods from this category. Program-of-Thoughts (PoT) [5] enhances complex reasoning by prompting language models to output executable code steps instead of plain text. Chain-of-Table [48] extends this by evolving intermediate table states, enabling LLMs to iteratively generate and apply operators, providing structured context throughout the transformation. ReAct [52] combines reasoning with action execution, incorporating feedback mechanisms like execution validation to improve the correctness of synthesized pipelines. FunctionCalling [40] constrains LLMs to invoke a pre-defined set of structured transformation operators with well-typed JSON arguments. These methods can be easily adapted to the ADP task by modifying their reasoning procedures and aligning their operator settings with DP transformation primitives. More detailed adaptations are described in Section 4.

6 Conclusions

In this paper, we present BAT, an open-source LLM-powered automatic data preparation framework, which is driven by Monte Carlo Tree Search (MCTS). In BAT, we first introduce a DP action sandbox (DPAS) to constrain the exploration space. Then, FPG generates multi-step pipelines via the proposed LLM-driven Monte Carlo tree search method. At each search step, a large language model (LLM) analyzes the source table and the target table schema to propose the next transformation action. Once a complete DP pipeline is generated, the LLM evaluates the reliability of the generated pipeline using a reward mechanism. Finally, BAT selects the most reliable pipeline as the result. To further improve search reliability and efficiency, we design an execution-aware pipeline optimizer (EPO). It incorporates execution-aware reward mechanisms to ensure the reliability and accelerates the search process via simulation caching and early termination. Comprehensive experiments confirm the superiority of BAT. In the future, we would like to incorporate user intent into automated data preparation. By allowing users to guide the process with natural language commands, we can ultimately make the system accessible to a much broader audience. We plan to develop more complex and large-scale datasets in the future.

Acknowledgments

This work was supported in part by the NSFC under Grants No. (62025206, U23A20296, 62502426), Zhejiang Province's "Jianbing" R&D Project under Grant No. (2025C01195). Yunjun Gao is the corresponding author of the work.

References

[1] Ziawasch Abedjan, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2016. DataXFormer: A robust transformation discovery system. In *ICDE*. 1134–1145.

[2] Louis T Becker and Elyssa M Gould. 2019. Microsoft power BI: extending excel to manipulate, analyze, and visualize diverse data. *Serials Review* 45, 3 (2019), 184–188.

[3] Guoxin Chen, Minpeng Liao, Chengxi Li, and Kai Fan. 2024. Alphamath almost zero: process supervision without process. In *NeurIPS*, Vol. 37. 27689–27724.

[4] Sibei Chen, Hanbing Liu, Waiting Jin, Xiangyu Sun, Xiaoyao Feng, Ju Fan, Xiaoyong Du, and Nan Tang. 2024. ChatPipe: orchestrating data preparation pipelines by optimizing human-ChatGPT interactions. In *SIGMOD*. 484–487.

[5] 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* (2022).

[6] Rémi Coulom. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International Conference on Computers and Games*. 72–83.

[7] Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. 2024. Generating code world models with large language models guided by monte carlo tree search. In *NeurIPS*, Vol. 37. 60429–60474.

[8] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System. In *CIDR*.

[9] 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).

[10] Scott R Granter, Andrew H Beck, and David J Papke Jr. 2017. AlphaGo, deep learning, and the future of the human microscopist. *Archives of pathology & laboratory medicine* 141, 5 (2017), 619–621.

[11] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

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

[13] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek R. Narasayya, and Surajit Chaudhuri. 2018. Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations. *Proc. VLDB Endow.* 11, 10 (2018), 1165–1177.

[14] Yeye He, Kris Ganjam, Kukjin Lee, Yue Wang, Vivek Narasayya, Surajit Chaudhuri, Xu Chu, and Yudian Zheng. 2018. Transform-data-by-example (tde) extensible data transformation in excel. In *SIGMOD*. 1785–1788.

[15] Jeffrey Heer, Joseph M. Hellerstein, and Sean Kandel. 2015. Predictive Interaction for Data Transformation. In *CIDR*.

[16] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798* (2023).

[17] Alteryx Inc. 2025. Alteryx - data preparation. https://www.alteryx.com/glossary/data-preparation.

[18] Informatica Inc. 2025. Data Mesh Architecture for ELT. https://www.informatica.com/resources/articles/etl-strategies-for-ai-analytics.html.html.

[19] Informatica Inc. 2025. Enable Zero Trust with a Data Foundation. https://www.informatica.com/content/dam/informatica-com/en/collateral/brochure/enable-zero-trust-with-data-foundation_brochure_4378en.pdf.

[20] Informatica Inc. 2025. Simplify Data Integration with a CLAIRE Co-pilot. https://www.informatica.com/content/dam/informatica-cxp/techtuesdays-slides-pdf/SimplifyDataIntegrationwithaCLAIRECo-pilot.pdf.

[21] Palantir Inc. 2025. Palantir - AIP features in pipeline builder. https://www.palantir.com/docs/foundry/pipeline-builder/pipeline-builder-aip.

[22] Palantir Inc. 2025. Palantir foundry. https://www.palantir.com/docs/foundry/data-integration/data-pipeline.

[23] Zhongjun Jin, Michael R. Anderson, Michael J. Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *SIGMOD*. 683–698.

[24] Zhongjun Jin, Michael Cafarella, HV Jagadish, Sean Kandel, Michael Minar, and Joseph M Hellerstein. 2018. CLX: Towards verifiable PBE data transformation. *arXiv preprint arXiv:1803.00701* (2018).

[25] Zhongjun Jin, Yeye He, and Surajit Chauduri. 2020. Auto-transform: learning-to-transform by patterns. *Proc. VLDB Endow.* 13, 12 (2020), 2368–2381.

[26] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*. 282–293.

[27] Martin Koehler, Edward Abel, Alex Bogatu, Cristina Civili, Lacramioara Mazilu, Nikolaos Konstantinou, Alvaro Fernandes, John Keane, Leonid Libkin, and Norman W Paton. 2019. Incorporating Data Context to Cost-Effectively Automate End-to-End Data Wrangling. *IEEE Computer Architecture Letters* 01 (2019), 1–1.

[28] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *NeurIPS*, Vol. 35. 22199–22213.

[29] 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).

[30] 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.

[31] Arash Dargahi Nobari and Davood Rafiei. 2022. Efficiently transforming tables for joinability. In *ICDE*. 1649–1661.

[32] Pandas. 2025. Pandas operator: concat. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html.

[33] Pandas. 2025. Pandas operator: datetime. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_datetime.html.

[34] Pandas. 2025. Pandas operator: Groupby. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html.

[35] Pandas. 2025. Pandas operator: melt. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.melt.html.

[36] Pandas. 2025. Pandas operator: merge. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.merge.html.

[37] Pandas. 2025. Pandas operator: pivot. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pivot.html.

[38] Pandas. 2025. Pandas operator: rename. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.rename.html.

[39] Zhenting Qi, Mingyuan Ma, Jiahang Xu, Li Lyna Zhang, Fan Yang, and Mao Yang. 2025. Mutual reasoning makes smaller llms stronger problem-solvers. In *ICLR*.

[40] Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, et al. 2024. Tool learning with foundation models. *Comput. Surveys* 57, 4 (2024), 1–40.

[41] Rita L Sallam, Paddy Forry, Ehtisham Zaidi, and Shubhangi Vashisth. 2016. *Market Guide for Self-Service Data Preparation.* Technical Report. Gartner, Inc.
[42] 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 *BigData*. 1824–1834.
[43] Rishabh Singh. 2016. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proc. VLDB Endow.* 9, 10 (2016), 816–827.
[44] Theresa Stadler, Bristena Oprisanu, and Carmela Troncoso. 2022. Synthetic Data–Anonymisation Groundhog Day. In *USENIX Security Symposium*. 1451–1468.
[45] V Stafford. 2020. Zero Trust Architecture. *National Institute of Standards and Technology* 800, 207 (2020), 800–207.
[46] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI*. 452–466.
[47] Zhaoyang Wang, Weilei He, Zhiyuan Liang, Xuchao Zhang, Chetan Bansal, Ying Wei, Weitong Zhang, and Huaxiu Yao. 2025. Cream: Consistency regularized self-rewarding language models. In *ICLR*.
[48] Zilong Wang, Hao Zhang, Chun-Liang Li, Julian Martin Eisenschlos, Vincent Perot, Zifeng Wang, Lesly Miculicich, Yasuhisa Fujii, Jingbo Shang, Chen-Yu Lee, et al. 2024. Chain-of-table: Evolving tables in the reasoning chain for table understanding. In *ICLR*.
[49] Cong Yan and Yeye He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *SIGMOD*. 1539–1554.
[50] Junwen Yang, Yeye He, and Surajit Chaudhuri. 2021. Auto-Pipeline: Synthesize Data Pipelines By-Target Using Reinforcement Learning and Search. *Proc. VLDB Endow.* 14, 11 (2021), 2563–2575.
[51] 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. In *NeurIPS*, Vol. 36. 11809–11822.
[52] 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 *ICLR*.
[53] Xiaosong Yuan, Chen Shen, Shaotian Yan, Xiaofeng Zhang, Liang Xie, Wenxiao Wang, Renchu Guan, Ying Wang, and Jieping Ye. 2024. Instance-adaptive zero-shot chain-of-thought prompting. In *NeurIPS*, Vol. 37. 125469–125486.
[54] Erkang Zhu, Yeye He, and Surajit Chaudhuri. 2017. Auto-Join: Joining Tables by Leveraging Transformations. *Proc. VLDB Endow.* 10, 10 (2017), 1034–1045.

Received October 2025; revised January 2026; accepted February 2026