Text-to-Pipeline: Bridging Natural Language and Data Preparation Pipelines [Experiment, Analysis & Benchmark]

Yuhang Ge
Zhejiang University
yuhangge@zju.edu.cn

Yachuan Liu
Zhejiang University
liuyachuan@zju.edu.cn

Zhangyan Ye
Zhejiang University
yzyet@zju.edu.cn

Yuren Mao
Zhejiang University
yuren.mao@zju.edu.cn

Yunjun Gao
Zhejiang University
gaoyj@zju.edu.cn

Abstract

Data preparation (DP) transforms raw data into a form suitable for downstream applications, typically by composing operations into executable pipelines. Building such pipelines is time-consuming and requires sophisticated programming skills, posing a significant barrier for non-experts. To lower this barrier, we introduce *Text-to-Pipeline*, a new task that translates NL data preparation instructions into DP pipelines, and PARROT, a large-scale benchmark to support systematic evaluation. To ensure realistic DP scenarios, PARROT is built by mining transformation patterns from production pipelines and instantiating them on 23,009 real-world tables, resulting in ~18,000 tasks spanning 16 core operators. Our empirical evaluation on PARROT reveals a critical failure mode in cutting-edge LLMs: they struggle not only with multi-step compositional logic but also with semantic parameter grounding. We thus establish a strong baseline with *Pipeline-Agent*, an execution-aware agent that iteratively reflects on intermediate states. While it achieves state-of-the-art performance, a significant gap remains, underscoring the deep, unsolved challenges for PARROT. It provides the essential, large-scale testbed for developing and evaluating the next generation of autonomous data preparation agentic systems.

PVLDB Reference Format:
Yuhang Ge, Yachuan Liu, Zhangyan Ye, Yuren Mao, and Yunjun Gao. Text-to-Pipeline: Bridging Natural Language and Data Preparation Pipelines [Experiment, Analysis & Benchmark]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/A11en0/Text-to-Pipeline.

1 Introduction

Data preparation (DP) refers to the process of transforming raw data into a form suitable for downstream applications such as business intelligence (BI) and machine learning (ML) [5, 11, 45, 46]. As a core component of modern data management, tabular DP plays a central role in supporting workflows in data warehouses and BI systems [5]. Preparing tabular data typically involves multiple operations such as filtering [30], joining [8], grouping [41], and reshaping [13]. These operations are often composed into pipelines where each step incrementally transforms the table and feeds the result into the next [20, 41, 42]. However, building correct and efficient pipelines is time-consuming and requires sophisticated programming skills, which is challenging even for experienced data engineers, as they need to compose pipelines in a large compositional space. Moreover, this poses a significant technical barrier for non-experts and prevents them from participating in DP.

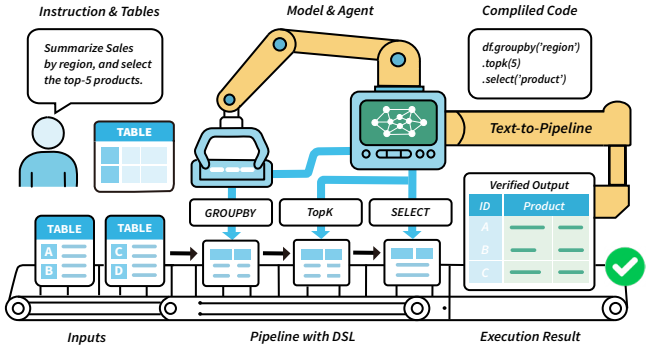

Figure 1: Task overview of *Text-to-Pipeline*.

To lower this barrier, a natural language (NL) interface that allows users to complete DP tasks by writing NL instructions is a natural choice. This paradigm has been explored in related tasks such as Text-to-SQL [21, 44], spreadsheet formula generation [29, 49], or code generation [39, 47]. However, these tasks primarily focus on database queries or cell-wise logic, not the multi-step, stateful transformations central to DP pipelines. In parallel, other automated pipeline construction methods [20, 41, 42] are not driven by NL; instead, they rely on structured supervision such as input-output table pairs [4, 13, 42] or schema graphs [20], making them inapplicable when only NL instructions are available. This leaves a critical gap for NL-driven pipeline generation. Simply tasking LLMs to generate monolithic scripts in general-purpose languages like Pandas is also insufficient, as this approach lacks the modularity, formal verification, and robust state-tracking necessary to guarantee correctness over complex, multi-step transformations.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

To bridge this gap, we introduce *Text-to-Pipeline* task: translating NL instructions into executable DP pipelines over tabular data. We formalize it as symbolic program generation in a domain-specific language (DSL), which can be compiled into executable backend code such as Pandas or SQL. DSL offers a structured representation, backend flexibility, and stronger support for verification and evaluation than directly generating code. As shown in Fig. 1, an instruction "*Summarize sales by region, and select the top-5 products.*" and input tables, the system generates a pipeline in DSL, e.g., `GroupBy`, `Topk`, `Select`, which is compiled into backend code and executed to produce the final table. To support this task, we build PARROT, a large-scale benchmark with ~18,000 multi-step DP tasks spanning 16 core operators, using 23,009 real tables from six public sources. Constructing such a benchmark poses significant challenges: First, pipelines must be structurally realistic, reflecting real-world transformation patterns. Second, semantically valid (i.e., executable by construction on the given tables' schema). Third, faithfully aligned with diverse, natural NL instructions.

To address these challenges, we design a rigorous five-stage synthesis framework. We first extract transformation patterns from production pipelines to ensure our tasks align with practical usage. Next, we introduce a novel propose-then-validate synthesis process that separates structural realism from semantic validity. A *Proposer*, guided by empirically-derived Markov transition matrices, samples structurally realistic operator chains. Then, a *Validator*, powered by a formal Schema Propagation Mechanism (SPM), ensures each proposed operation is semantically valid given the current table state. This guarantees all generated pipelines are executable by construction. Finally, after compiling the DSL to executable code and generating intent-aligned NL using LLMs, we perform multi-phase validation, including a rigorous review by six PhD-level human experts, to ensure gold-standard data quality.

Our evaluation of PARROT validates our core hypothesis: formalism is critical. Targeting our symbolic DSL achieves **62.88%** accuracy, vastly outperforming direct code generation like Pandas (**33.8%**). Yet, even with a robust DSL, **even** cutting-edge LLMs like GPT-4o exhibit a critical failure: they achieve high program validity (**81.12%**) but their execution accuracy drops to **71%**, producing programs that *run* but yield the *wrong answer*. These findings highlight that direct prompting generation fails at the core challenges of multi-step compositional logic and semantic parameter grounding. To address this, we propose *Pipeline-Agent*, an execution-aware agent that iteratively reasons over intermediate states, achieving a state-of-the-art performance of **76.17%**. Despite this, the significant remaining gap underscores that these two challenges are the deep, unsolved problems posed by this task, calling for new solutions. In summary, our key contributions are:

- **Task:** We formalize the *Text-to-Pipeline* task, translating natural language into executable, multi-step data preparation pipelines.
- **Benchmark:** We construct PARROT, a large-scale, high-quality benchmark of ~18,000 instances, using a novel synthesis framework that guarantees pipeline validity and realism.
- **Findings:** We demonstrate that our DSL formalism is critical (+29 points over Pandas) and that SOTA LLMs fail on multi-step compositional logic and semantic parameter grounding.
- **Baseline:** We propose Pipeline-Agent, an SOTA agent that serves as a foundation for future research.

Finally, PARROT provides a challenging testbed for developing the next generation of intelligent data preparation agentic systems.

2 Related Work

NL-Driven Program Generation. Prior research has extensively studied how to translate NL into executable programs. Specifically, Text-to-SQL methods [21, 23, 25, 44] map NL queries to SQL statements, emphasizing semantic understanding and schema linking. These approaches are typically benchmarked on datasets such as WikiSQL [50], Spider [44], Spider 2.0 [21], and BIRD [25]. Text-to-Formula [22, 29, 49] techniques like NL2Formula [49], SheetCopilot [22], and SpreadsheetBench [29] focus on converting NL into spreadsheet formulas or targeted cell-level edits, suitable for localized spreadsheet manipulations. Additionally, general Text-to-Code frameworks [6, 10, 15, 48] translate NL instructions into scripts in languages like Python, C++, and Java, addressing diverse standalone programming tasks tested on HumanEval [6] and APPS [15]. Although these paradigms leverage NL, these tasks typically target the level of SQL queries, cell-wise formulas for spreadsheets, or logic functions for general programming. In contrast, *Text-to-Pipeline* focuses on generating multi-step, executable pipelines for data preparation, where the objective is to transform input tables into expected outputs through schema-aware DSL programs.

Automatic Data Pipeline Generation. Automating data pipeline construction is often framed as program synthesis. Early methods rely on manual coding or visual tools [1, 2], while example-driven approaches [3, 12–14, 17, 18, 34, 51] require input-output (IO) table pairs and struggle with multi-step logic. Subsequent work [4, 35, 36, 42] extends to multi-step synthesis but still depends on output supervision. Auto-Tables [26] and Auto-Prep [20] remove this constraint using self-supervised learning and mining existing operator traces, respectively. However, none of these methods support open-ended pipeline generation directly from NL instructions. In parallel, several human-interactive systems assist users during pipeline construction. EDAssistant [27] supports in-situ code search, and Auto-Suggest [41] mines notebook patterns to recommend transformations. ChatPipe [7] enables conversational construction with LLMs. These systems rely on user interaction or code context, focusing on usability over automation. AutoPrep [9] uses a multi-agent framework for question-aware preparation in TableQA, but does not support general-purpose pipeline generation. In contrast, our *Text-to-Pipeline* task with PARROT targets fully automatic pipeline generation from NL instructions, enabling end-to-end generation without human feedback or IO supervision.

3 Task Definition

We formally define the *Text-to-Pipeline* task as translating a natural language instruction over an input table set into an executable, multi-step data preparation pipeline. We represent this pipeline not as direct code, but as a symbolic program in a domain-specific language (DSL). This DSL representation is backend-agnostic, allowing it to be compiled into various executable codes.

Problem Setup. Let $\mathcal{X}$ denote the space of all possible input table sets, $\mathcal{Y}$ the space of output tables, $\mathcal{L}$ the space of natural language

instructions, and $\mathcal{P}$ the space of DSL programs. An input instance consists of a set of one or more tables $\mathbf{x} = \{x_1, \ldots, x_m\} \in \mathcal{X}$. Each instance is associated with a reference output table $y \in \mathcal{Y}$. Given an instruction $\ell \in \mathcal{L}$ and an input table set $\mathbf{x} \in \mathcal{X}$, the objective is to learn a mapping $f : \mathcal{L} \times \mathcal{X} \to \mathcal{P}$ that yields a symbolic program $p = f(\ell, \mathbf{x})$, which can be compiled and executed to produce an output $\hat{y}$:

$$\hat{y} = \texttt{Exec}(c, \mathbf{x}) \quad \text{where } c = \texttt{Compile}(p). \tag{1}$$

Here, $\texttt{Compile}(p)$ is the compiled executable code, Exec denotes the execution engine (e.g., Pandas or SQL), and $\hat{y}$ is expected to be equivalent to the reference output y. In summary, each instance of this *Text-to-Pipeline* can be represented as a five-tuple $(\mathbf{x}, \ell, p, c, y)$.

Task Scope. The scope of *Text-to-Pipeline* focuses on data preparation (e.g., cleaning, integration, transformation) over a given set of tables. This mirrors common enterprise workflows where data discovery is handled by upstream systems (e.g., data catalogs), and users then perform multi-step transformations on the discovered tables, as seen in tools like Trifacta [2] or Power Query [1].

Pipeline Structure. Each program $p \in \mathcal{P}$ is a left-to-right chain of k symbolic operators: $p = o_1 \circ o_2 \circ \cdots \circ o_k$, where each operator $o_i \in O$ is drawn from a core set covering common DP actions:

$$O = \{\texttt{groupby}, \texttt{sort}, \texttt{join}, \texttt{dropna}, \texttt{filter}, \ldots\}. \tag{2}$$

Each operator o_i is parameterized by structured arguments (e.g., column names, aggregation functions). For a complete list of the 16 operators and their categories, refer to Tab. 1. Despite its modular design, synthesizing such pipelines is non-trivial. The space of valid operator sequences grows combinatorially with length, and each step is constrained by schema compatibility, parameter validity, and cross-step dependencies. Moreover, subtle interactions among operators (e.g., column renaming before aggregation) can significantly affect program correctness and execution outcomes.

Why DSL over Direct Code. Compared to generating Pandas code directly, DSL offers three key advantages: (1) *Better planning and verification*: DSL supports step-wise reasoning, schema validation, and error tracing. It enables fine-grained evaluation (e.g., operator accuracy) and is easier to synthesize at scale. (2) *Stable structure*: DSL uses a fixed set of operations with clear parameter formats, avoiding the syntactic variance of Pandas (e.g., multiple ways to filter or group data). This improves model learnability and consistency. (3) *Backend flexibility*: DSL can be compiled to Pandas, SQL, or Spark, making it adaptable to different runtime environments. In contrast, Pandas ties the output to Python execution. Experiments further confirm this choice, showing consistent performance advantages over direct Pandas and SQL generation (see Tab. 7).

Evaluation Metrics. We employ three primary metrics to provide a multifaceted view of model performance, evaluating both execution correctness and program structure fidelity.

- **Execution Accuracy (EA)**: Given input tables $\mathbf{x}$ and a generated program $\hat{p}$, we execute $\hat{p}$ to obtain $\hat{y} = \texttt{Exec}(\hat{p}, x)$. EA measures the proportion of samples where $\hat{y} \overset{\star}{=} y$, i.e., the predicted output $\hat{y}$ matches the ground truth y up to canonical equivalence (e.g., row/column permutations, floating-point tolerance):

$$\text{EA} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(\hat{y}_i \overset{\star}{=} y_i),$$

where N is the number of test samples, and $\mathbb{I}(\cdot)$ is the indicator function.

- **Program Validity (PV)**: The proportion of generated programs $\hat{p}$ that can be successfully compiled into executable code $c = \texttt{Compile}(\hat{p})$ *and* be executed ($\hat{y} = \texttt{Exec}(c, \mathbf{x})$) without any runtime errors, regardless of output correctness:

$$\text{PV} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(\texttt{Valid}(\hat{p}_i)),$$

where $\texttt{Valid}(\hat{p}_i)$ returns true if $\hat{p}_i$ compiles and executes successfully.

- **Operator Accuracy (OA)**: This metric provides a complementary, order-agnostic view, measuring the proportion of correctly identified operators. It evaluates the selection of the correct *set* of operations, while EA evaluates the sequence. For a generated program $\hat{p}$ and ground truth p, OA is computed as:

$$\text{OA} = \frac{1}{N} \sum_{i=1}^{N} \frac{|\text{set}(\hat{p}_i) \cap \text{set}(p_i)|}{|\text{set}(p_i)|},$$

where $\text{set}(p_i)$ denotes the set of operators in the ground truth program p_i.

4 Benchmark Design

To support systematic evaluation of *Text-to-Pipeline*, we synthesize a high-quality benchmark of about 18,000 instances through a carefully designed data construction framework. As shown in Fig. 2, the process involves five stages: (1) **Data Collection and Curation**, where we gather and standardize tables from diverse real-world sources; (2) **Operator Chain Construction**, where we sample realistic, schema-valid transformation pipelines using a Markov model guided by our Schema Propagation Mechanism (SPM); (3) **Rule-Based Code Compilation**, where we deterministically compile the abstract pipelines into executable code to generate the ground-truth output tables; (4) **Instruction Generation**, where we use large language models to create natural language instructions corresponding to the pipelines and their I/O tables; and (5) **Multi-phase Validation**, where we employ both automatic execution checks and human-in-the-loop verification to ensure data quality. We detail each of these five stages in the following subsections.

4.1 Stage 1: Data Collection and Curation

To simulate realistic DP scenarios, we collect tables from six public datasets covering diverse real-world domains.

- **Spider** [44] is a large-scale, human-annotated benchmark for Text-to-SQL, designed to test cross-domain generalization. It features 200 databases with multiple tables, requiring models to handle complex queries and intricate schema relationships.
- **Bird** [25] is a large-scale benchmark for Text-to-SQL focused on real-world business intelligence (BI) scenarios. It includes large databases with varied, noisy table structures and complex join conditions, mirroring challenges in enterprise environments.
- **Auto-Pipeline** [42] and **Auto-Tables** [26] are benchmarks designed to advance the automation of complex data preparation workflows. They focus on synthesizing multi-step transformations, such as joins, pivots, and unpivoting, often without user-provided examples (e.g., using a "by-target" paradigm).

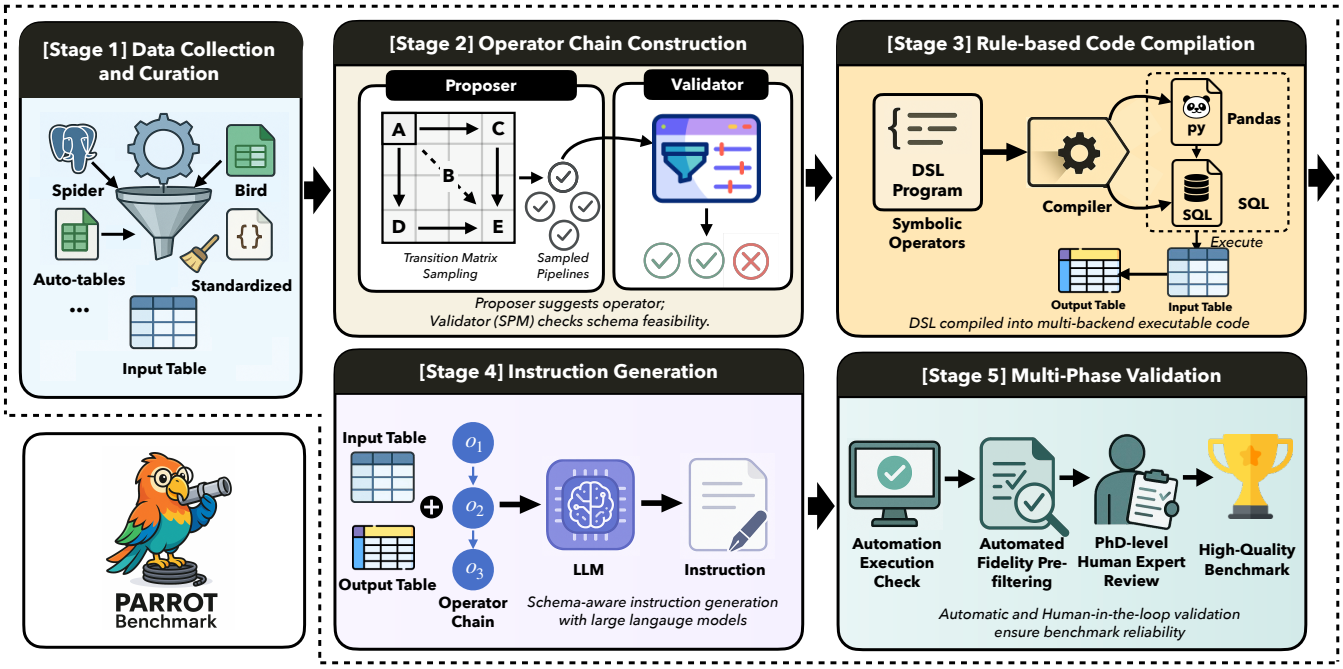

Figure 2: The data synthesis workflow of PARROT.

- **TableBench** [40] is a comprehensive benchmark for Table Question Answering (TableQA) that emphasizing complex reasoning across diverse domains such as fact checking, numerical reasoning, and data analysis.
- **LakeBench** [8] is a massive-scale (over 1TB) benchmark designed to evaluate methods for discovering joinable and unionable tables within data lakes. It features over 10,000 queries with ground truth annotations.

From these sources, we curated 23,009 tables, spanning domains such as blockchain, finance, healthcare, and education. These tables exhibit diverse structures, ranging from 2 to 120 columns, and include both wide and long formats. To ensure downstream compatibility, we then performed several curation steps to ensure quality and tractability. First, we performed structural validation, filtering out any empty or irregular tables that lacked a coherent schema. Second, to ensure manageable execution times for our benchmark tasks, we enforced a uniform row limit, truncating tables that exceeded 50 rows while preserving their original schema. Finally, we applied basic preprocessing to clean superficial data inconsistencies. This included steps such as trimming leading/trailing whitespace from string values and unifying common null representations (e.g., literal strings like "NA", "Null", or empty strings "") into a standard NaN marker. This curation process resulted in a robust and diverse collection of tables used for the pipeline synthesis stage.

4.2 Stage 2: Operator Chain Construction

This stage aims for synthesizing operator chains $p = o_1 \circ \cdots \circ o_k$ that are both structurally realistic and semantically valid. Our synthesis framework is designed as a propose-then-validate process. At each step of building the chain, this process performs two key actions: (1) **Propose**: the *Proposer*, trained on real-world operator sequences, suggests a set of structurally plausible next operators. (2) **Validate**: the *Validator*, powered by a formal schema engine, checks if a proposed operator is semantically executable given the current table schema. This iterative propose-then-validate loop guarantees that every synthesized pipeline p is executable by construction. We now detail the design of these two core components.

The Proposer. The Proposer's role is to ensure that generated operator sequences reflect common patterns found in real-world data preparation. To achieve this, we model operator transitions as a first-order Markov process. We first define our DSL with 16 core operators (see Tab. 1). We then construct an empirical transition matrix $\mathbf{P} \in \mathbb{R}^{|O|\times|O|}$. To build this matrix, we collected and parsed 1,200 data transformation scripts from open-source repositories (e.g., Kaggle, GitHub) [26, 41, 42]. Each script was parsed into an abstract syntax tree (AST), and high-level transformation calls were extracted and mapped to our normalized DSL operator set (e.g., `df.drop_duplicates` → `dedup`). We then recorded all observed consecutive operator pairs $(o_i \to o_j)$ to compute transition frequencies. These frequencies were normalized into conditional probabilities $P(o_j \mid o_i)$ using Laplace smoothing ($\alpha = 0.5$) to mitigate data sparsity. This resulting matrix $\mathbf{P}$ serves as our generative model for proposing structurally realistic operators.

The Validator. The Proposer (using $\mathbf{P}$) only ensures structural realism; it has no knowledge of schema and may propose an invalid operation (e.g., applying `groupby` to a non-existent column). The Validator's role is to prevent this. We designed the Validator around a core engine we call the **Schema Propagation Mechanism (SPM)**. The SPM formally defines the effect of any operator o_t

Table 1: Supported operators: categorized by task type, typical parameters, Pandas-style examples, and observed frequencies.

Operator	Typical Parameters	Example (Pandas-style)	Frequency
Data Cleaning			
`filter`	condition	`df.query("value != 1")`	8876
`dropna`	axis, how, subset	`df.dropna()`	3925
`deduplicate`	subset, keep	`df.drop_duplicates()`	7065
`cast`	column, dtype	`df["value"].astype("float")`	3888
Data Integration			
`join`	left, right, on, how	`df1.merge(df2, on="id", how="inner")`	3643
`union`	dataframes	`pd.concat([df1, df2])`	2198
Structural Reconstruction			
`groupby`	by, agg	`df.groupby("region").sum()`	9271
`pivot`	index, columns, values	`df.pivot("id", "type", "score")`	2933
`unpivot`	id_vars, value_vars	`df.melt(id_vars=["id"], value_vars=["value"])`	4365
`explode`	column	`df.explode("tags")`	2135
`transpose`	-	`df.transpose()`	2271
`wide_to_long`	stubnames, i, j	`pd.wide_to_long(...)`	588
Assisted Operations			
`sort`	by, ascending	`df.sort_values("time")`	8588
`topk`	columns, k	`df.head(5)`	4074
`select`	columns	`df[["name", "value"]]`	5762
`rename`	columns	`df.rename(columns={"old": "new"})`	5937

on a given schema $\mathcal{S}_t$ at each step t, producing a new output schema $\mathcal{S}_{t+1}$. Specifically, the schema is represented as a structured object $\mathcal{S}_t = \{(c_i, \tau_i)\}_{i=1}^{n}$, where c_i is the column name and τ_i is its data type. Each DSL operator is associated with a deterministic transformation rule $\delta : \mathcal{S}_t \to \mathcal{S}_{t+1}$ that specifies how it modifies the schema. For example, `groupby` introduces new aggregation columns and may drop non-grouped fields; `join` merges two schemas, applying name disambiguation (e.g., suffixes `_x`, `_y`) if overlapping columns exist; `rename` updates column names while preserving types; `pivot` restructures column names based on index and values; and `dropna` does not modify the schema structure. During chain construction, we maintain a running schema $\mathcal{S}_t$, updating it at each step via $\mathcal{S}_{t+1} = \delta(o_t, \mathcal{S}_t)$. This propagation strategy enables robust error prevention: before an operator o_t is added to the chain, the Validator checks if its required arguments (e.g., column names) are present and well-typed in $\mathcal{S}_t$. If the check fails, the operator is rejected. For multi-source operations (e.g., `join`, `union`), the SPM tracks a set of schemas $\{\mathcal{S}_t^{(1)}, \mathcal{S}_t^{(2)}, \dots\}$ and applies compatibility checks, such as matching join keys.

Schema-Aware Synthesis Algorithm. We now combine the Proposer and Validator into a unified algorithm, as summarized in Algo. 1. The initial operator o_1 is drawn uniformly. At each subsequent step i, the Proposer *proposes* a candidate operator o_i by sampling from $\mathbf{P}(\cdot \mid o_{i-1})$. This candidate is then passed to the Validator (SPM) for verification.

This validation step is critical: the SPM not only checks if o_i is applicable but also guides the selection of valid parameters (e.g., columns) for o_i. If the operator is successfully validated and parameterized, it is appended to the chain p, and the SPM updates the schema via $\mathcal{S}_i = \delta(o_i, \mathcal{S}_{i-1})$. If validation fails (e.g., no valid columns can be found), the operator o_i is simply discarded, and the algorithm proceeds to the next iteration $i + 1$. The chain length k is sampled from a truncated geometric distribution over $[1, 8]$. We also adopt a three-level difficulty scheme based on program length (e.g., Easy (≤ 3 ops), Medium (4-6 ops), and Hard ($\geq$7 ops)), as detailed in Sec. 5. This entire process guarantees that all synthesized pipelines p are executable by construction. The detailed definitions and illustrative examples for each level are provided in App. A.5.

4.3 Stage 3: Rule-Based Code Compilation

After yields a validated symbolic pipeline p, we compile it into executable code c. This serves two purposes: (1) generating the final output table y (by executing c), which is required for instruction generation in Stage 4.4, and (2) creating the executable ground truth for evaluation. Since p has been validated by the SPM, the compiler can be a simple, deterministic, and stateless template engine.

Compilation workflow. We implement a compiler to translate DSL programs into executable code. The process maps each operator to backend-specific templates, ensuring correct parameter binding, schema handling, and code generation. Given a DSL program $p = [o_1, o_2, \dots, o_k]$, the compiler performs the following:

(1) **Parameter binding:** For each operator o_i, extract its parameters and map them to template slots;

Algorithm 1 Operator Chain Construction

Require: Curated table set $\mathbf{x}$, Transition Matrix $\mathbf{P}$, Schema Propagation Mechanism (SPM)
Ensure: Valid operator chain $p = \{o_1, o_2, \ldots, o_k\}$

Sample chain length $k \sim \text{TruncGeom}(1, 8)$
Sample first operator o_1 uniformly from O
Initialize chain $p \leftarrow [o_1]$
Get initial schema $\mathcal{S}_0 \leftarrow \texttt{GetSchema}(\mathbf{x})$
for $i = 2$ to k **do**
  // Propose a structurally realistic operator
  Sample $o_i \sim \mathbf{P}(\cdot \mid o_{i-1})$
  // Validate operator and bind parameters using SPM
  $is\_valid, o_i \leftarrow \texttt{SPM.ValidateAndBind}(o_i, \mathcal{S}_{i-1})$
  if *is_valid* is `True` **then**
    Append o_i to chain p
    $\mathcal{S}_i \leftarrow \texttt{SPM.UpdateSchema}(\mathcal{S}_{i-1}, o_i)$ // Propagate schema
  end if
  // If not valid, operator is discarded and loop continues
end for
return p

(2) **Schema-aware quoting:** Apply column name escaping (e.g., `_quote(col)`) to handle special characters;
(3) **Code merging:** Concatenate code lines using a left-to-right chaining convention (e.g., method chaining in Pandas);
(4) **Execution trace annotation:** Optionally insert line comments to denote DSL operator source (useful for debugging).

Backend support. While Pandas serves as the primary backend, the compiler is modular and supports alternate targets such as SQL or Spark through switchable template registries. Each backend maintains an operator-template mapping, enabling flexible deployment without modifying the symbolic layer.

Example. Given the DSL sequence:

```
[
  {
    "op": "groupby",
    "params": {
      "by": ["region"],
      "agg": { "sales": "sum" }
    }
  },
  {
    "op": "sort",
    "params": { "by": "sales", "ascending": false
    }
  }
]
```

The compiler generates the following Pandas code:

```
df.groupby('region')['sales']
    .sum()
    .reset_index()
    .sort_values(by='sales', ascending=False)
```

This translation preserves the semantics of the symbolic pipeline while ensuring correctness and interpretability.

4.4 Stage 4: Instruction Generation

With the symbolic pipeline p from Stage 2 and the corresponding output table y (generated by executing the compiled code from Stage 3), the next step is to generate a natural language instruction ℓ. Using the operator chain and an input-output table preview, $\{\mathbf{x}, y, p\}$, we generate instructions via a two-step process designed to ensure both semantic fidelity and linguistic diversity.

First, we prompt an LLM to generate a schema-aware, structured pipeline description grounded in the transformation. This initial prompt includes the task description, table schema, sampled rows, and in-context demonstrations to anchor the operator-language mappings and ensure semantic preservation, that is, making sure the generated text accurately reflects the underlying operations. Second, this structured draft is passed to a style-controlled refinement step, which converts the draft into a fluent, user-centric instruction. This two-step process is crucial as it decouples the goal of semantic grounding (Step 1) from that of stylistic expression (Step 2), which helps avoid templating bias.

For this entire process, we use `GPT-4o-mini` via OpenAI's API (`gpt-4o-mini-2024-07-18`) with a temperature of 0.7. This non-zero temperature is intentionally set to encourage linguistic diversity and mitigate the risk of deterministic, model-specific phrasing. The effectiveness of this de-biasing strategy is empirically validated in our benchmark analysis (see Sec. 5, Tab. 4), where PARROT demonstrates significantly higher lexical diversity (Distinct-n) and lower redundancy (Self-BLEU) than prior benchmarks. All prompts used for data synthesis are listed in App. B.1.

4.5 Stage 5: Multi-phase Validation

To ensure the quality and reliability of PARROT, every generated instance $(\mathbf{x}, \ell, p, c, y)$ undergoes a rigorous multi-phase validation process, combining automated checks with expert human review.

Phase 1: Automatic Execution Validation. First, we perform an automatic execution check. The compiled code c is executed on the input tables $\mathbf{x}$ to produce a predicted output $\hat{y}$. This $\hat{y}$ is then compared against the ground-truth y for canonical equivalence, denoted $\hat{y} \overset{\star}{=} y$. This check is robust to superficial formatting and ordering differences. Specifically, it verifies equivalence by: (1) sorting both tables by all columns to make row order invariant, (2) re-indexing columns to make column order invariant, and (3) applying a floating-point tolerance (e.g., 10^{-5}) for all numeric fields. Any instance where $\hat{y} \overset{\star}{=} y$ evaluates to `False` is discarded.

Phase 2: Automated Fidelity Pre-filtering. Second, we assess instruction fidelity. We employ an LLM-based "judge" to pre-filter samples, checking the semantic alignment between the natural language instruction ℓ (from Stage 4) and the symbolic operator chain p (from Stage 2). Using zero-shot classification prompts, the judge LLM scores the semantic consistency. Samples with low computed alignment scores are automatically discarded, reducing the burden on human annotators.

Phase 3: Expert Manual Validation (for Dev/Test Sets). Finally, to build a gold-standard evaluation set, we manual review of the data intended for the development and test sets, ensuring the highest possible data quality.

Table 2: Benchmark comparison across different tasks and execution properties. PARROT supports multi-step DSL programs grounded in natural language, with execution verified across multiple backends. Task types: TS = Text-to-SQL, TF = Text-to-Formula, P = Pipeline Synthesis, TP = Text-to-Pipeline. The symbol '-' indicates that it does not provide codes or instructions.

Benchmark	Task Type	#Inst.	#Tabs	#Ops	Avg. Steps	Atomic Exec	NL-Driven	Multi-Backend
Spider [44]	TS	10,181	1,056	27	4.78	✘	✓	✘
BIRD [25]	TS	12,751	693	27	6.52	✘	✓	✘
SpreadsheetBench [29]	TF	912	2,729	10	-	✘	✓	✘
NL2Formula [49]	TF	70799	21670	57	10.2	✘	✓	✘
Auto-Tables [26]	P	244	244	8	1.11	✓	✘	✓
Auto-Pipeline [42]	P	716	4,680	12	4.10	✓	✘	✓
PARROT(Ours)	TP	17,168	23,009	16	4.24	✓	✓	✓

- **Sampling:** We employed a stratified sampling strategy to select 3,000 instances for this review, drawing samples proportionally across both the three difficulty levels (Easy, Medium, Hard) and the 16 core operators. This ensures the reviewed subset mirrors the full benchmark's distribution.
- **Process:** We recruited and trained six graduate-level experts (MSc/PhD students in data science and NLP) with detailed annotation guidelines. Each expert scored instances on a 3-point scale across three criteria: (1) Instruction Accuracy (faithfully reflects the pipeline?), (2) Operator Coverage (all key steps implied?), and (3) Semantic Clarity (unambiguous?).
- **Outcome:** The inter-annotator agreement reached 91.4% (Cohen's $\kappa = 0.82$), confirming high reliability. Only samples that passed all automated checks (Phases 1 & 2) and received unanimous approval on all three criteria from our experts were retained for the final development and test sets. The remaining set of automatically validated data serves as the training set. To facilitate this rigorous human-in-the-loop process, we developed an interactive visualization platform to assist experts in inspecting and debugging data, which is detailed in App. C.

Table 3: Statistics of the PARROT.

Statistics	Number
Total Tasks	**17,168 (100%)**
Single Tab.	11,327 (66.0%)
Multi-Tab.	5,841 (34.0%)
Train / Dev / Test	14,388 / 1,387 / 1,393
Input Table	
Avg. Columns Per Tab.	6.7
Avg. Row Count	134.2
Num / Cat / Mixed	44.3% / 42.5% / 13.2%
Chain Complexity	
Easy ($\leq$ 3 ops)	32.36%
Medium (4~6 ops)	49.07%
Hard ($\geq$7 ops)	18.57%
Avg. Chain Length	4.24
Instructions	
Avg. Characters	192.3
Avg. Tokens	100.5

5 Benchmark Characteristics

We present a benchmark comparison in Tab. 2 and a statistical overview in Tab. 3. Compared with prior works, PARROT provides large-scale, NL-driven multi-step programs with broad operator coverage, verified atomic execution, and multi-backend support.

5.1 Compositional and Parameter Complexity

The complexity of PARROT is two-fold: the *compositional* depth of the programs and the *parametric* complexity of individual operators.

Compositional Complexity. Fig. 3 (left) illustrates the distribution of transformation chain lengths. The benchmark is centered on non-trivial sequences, with 49.07% of instances containing 4 to 6 operations and a significant tail of 18.57% exceeding 7 steps. As shown in Tab. 3, the average chain length of 4.24 is substantially higher than prior pipeline synthesis benchmarks like Auto-Tables [26] (1.11) and comparable to Auto-pipeline [42] (4.10), but over a dataset that is orders of magnitude larger and driven by natural language. This distribution accurately reflects the compositional nature of real-world tasks. The core challenge here is not merely length, but the necessity for models to manage long-range dependencies and schema propagation, where the validity of an operation at step t depends critically on the schema produced at step $t-1$.

Parameter Complexity. Beyond program depth, as illustrated in Fig. 4, the parameter complexity varies significantly. While operations like `rename` and `select` are simple, structural operations like `join`, `pivot`, and `groupby` exhibit high parameter complexity. This presents a critical challenge, as these operations require models to generate structured arguments, not just single values. For instance, a `groupby` requires identifying *both* the grouping keys ('by=...') *and* a dictionary of aggregation functions ('agg=...'). A `join` requires specifying join keys ('on=...') and the join type ('how=...'). This transforms the task from simple sequence generation into a hierarchical prediction problem, requiring the model to deeply understand table semantics (e.g., key vs. value columns) to correctly populate these complex, structured parameters.

5.2 Operational and Structural Diversity

As shown in Fig. 5, PARROT supports 16 operations with distinct frequencies, capturing the breadth of data preparation requirements

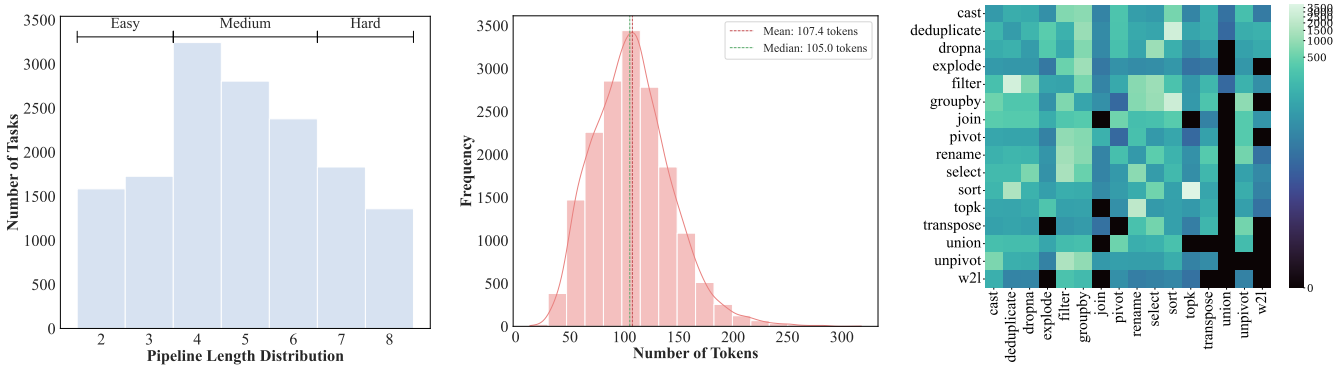

Figure 3: Left: pipeline length distribution over three difficulty levels. Middle: instruction length distribution by token frequency. Right: operation transition matrix. The abbreviation "w2l" stands for the wide-to-long operator.

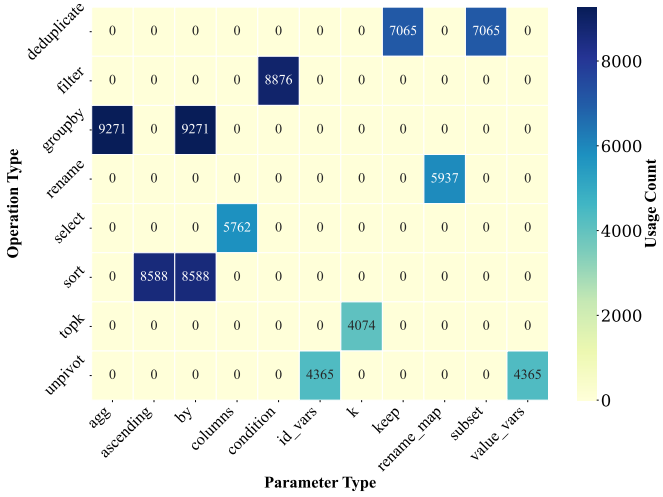

Figure 4: Heatmap of parameter usage across different operations. Darker colors indicate higher parameter complexity.

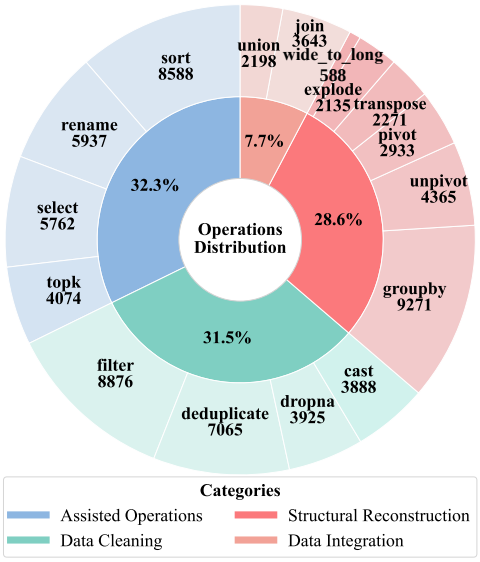

Figure 5: Operation distribution.

in real-world pipelines. Aggregation, reshaping, and integration together account for 48.6% of operator usage, while selection and ordering operations provide complementary functionality. This distribution effectively reflects empirically observed patterns in data science practice. Fig. 3 (right) shows the operator transition graph among the most frequently occurring operations, revealing diverse and non-linear patterns among common operations. The dense connectivity and heterogeneous edge weights underscore the rich compositional patterns present in multi-step pipelines, necessitating sophisticated reasoning capabilities for models to successfully predict operation sequences that maintain schema compatibility and semantic coherence.

To further examine the diversity of the table, Fig. 6 reports the distribution of column types in input versus output tables. Notably, output tables exhibit an increased proportion of textual columns and reduced numeric fields, reflecting structural transformations such as pivoting or aggregations that change schema layouts. These statistics collectively ensure that PARROT presents operational

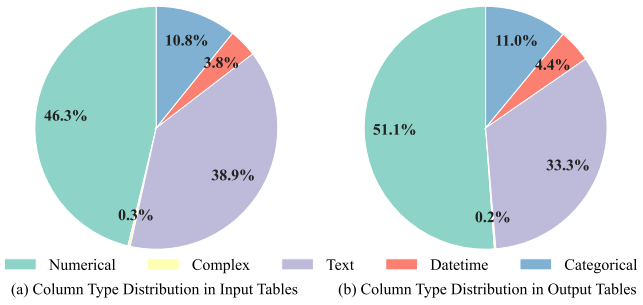

Figure 6: Column type distributions in input vs. output tables.

and structural diversity, challenging models across both symbolic planning and schema reasoning.

Table 4: Lexical diversity comparison across datasets.

Dataset	Distinct-1	Distinct-2	Self-BLEU-4
Spider [44]	0.39	0.62	0.81
NL2Formula [49]	0.42	0.68	0.77
PARROT (Ours)	**0.58**	**0.74**	**0.61**

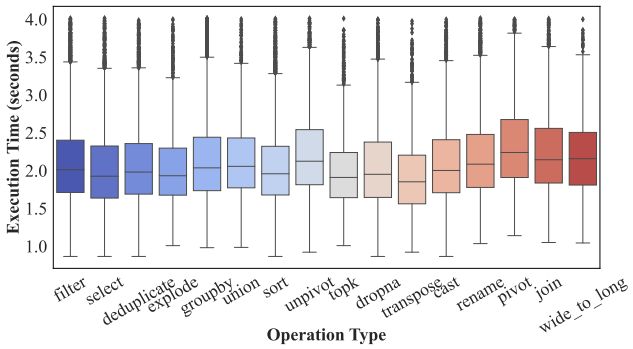

Figure 7: Distribution of per-operation execution times across all tasks, with the top 5% of outliers removed

5.3 Instruction Characteristics

Fig. 3 (mid) illustrates the length distribution of instructions in PARROT, with an average of 107.4 tokens. The distributions are right-skewed and bimodal, reflecting both concise directives and longer, context-rich descriptions. Instruction complexity correlates strongly with transformation chain length, suggesting that linguistically complex prompts often entail more compositional operations. To quantify lexical and semantic diversity, we compute standard generation diversity metrics, including Distinct-n and Self-BLEU: (1) **Distinct-n** [24]: Measures the proportion of unique n-grams in the instruction corpus. Higher values indicate richer lexical variety. We report Distinct-1 and Distinct-2. (2) **Self-BLEU** [52]: Measures overlap between an instruction and the rest of the corpus. Lower Self-BLEU implies lower redundancy and more diverse phrasing. Tab. 4 summarizes the results. Compared to other instruction-driven datasets such as Spider and NL2Formula, PARROT exhibits significantly higher lexical diversity and lower redundancy, reflecting its open-ended, LLM-generated language design. These results confirm that PARROT instructions exhibit greater lexical richness and structural variation, which helps benchmark model generalization to diverse user intents and phrasing styles. Importantly, while the instructions are synthesized for control and consistency, they are grounded in thousands of noisy real-world tables, ensuring models must interpret instructions grounded in noisy and diverse schemas.

5.4 Operator-Level Execution Characteristics

To quantify the computational costs of different operations, we analyze the per-operation execution time distribution across all tasks. Fig. 7 shows a box plot of execution times for each operation type, excluding the top 5% of outliers. The analysis reveals a clear computational hierarchy: data-intensive operations such as `unpivot` and `union` exhibit both higher median execution times, while lightweight operations like `select` and `dropna` typically complete within 2 seconds. This computational heterogeneity highlights that operation choice and ordering can markedly affect total execution time, motivating operation-aware scheduling and synthesis.

6 Pipeline-Agent

The *Text-to-Pipeline* task presents two fundamental challenges that standard prompting methods are ill-equipped to handle: multi-step compositional logic and semantic parameter grounding. First, models must plan a correct, order-dependent sequence of operations. Second, they must correctly ground parameters (e.g., column names, aggregation functions) to a table schema that is dynamically evolving with each transformation step.

Standard Tool Calling APIs [32] typically execute one-shot instructions, lacking the dynamic state maintenance required for sequential transformations. Plan-and-Solve [37], which separates the planning phase from the execution phase, cannot easily leverage intermediate execution feedback to correct or adapt the plan. Similarly, while Chain-of-Tables [38] focuses on static table reasoning, it lacks robust support for dynamic state tracking and adaptive tool chaining, making it less suitable for the iterative transformation tasks defined in our benchmark. To overcome these limitations, we introduce Pipeline-Agent, a unified framework designed to iteratively solve complex data preparation tasks by tightly coupling step-by-step reasoning with modular tool execution in a state-aware, feedback-driven interaction cycle. This agent's architecture is composed of three key components, which we elaborate on below.

Cognitive Orchestrator. This component is the agent's planning and reasoning engine, specifically designed to tackle the compositional logic challenge. Specifically, we adopt a ReAct-style [43] approach, which is essential for handling the dynamic state changes inherent in *Text-to-Pipeline*. Unlike static Plan-and-Solve methods that commit to a full plan upfront, our orchestrator interleaves reasoning and execution. Each "Thought" is a structured reasoning step where the agent must (a) analyze the user's remaining intent, (b) inspect the current table state (the "Observation"), and (c) formulate a single, executable "Action". This "Action" is a structured JSON call to a specific tool. After each action, the agent observes the outcome, such as the updated table schema or an execution error, which it as feedback for its next planning step. This allows the agent to dynamically refine its multi-step plan, correct mistakes, and handle complex dependencies based on observed table changes.

Modular Toolsets. This component represents the agent's actionable capabilities, defined as a set of deterministic, modular tools. Each tool corresponds directly to one of the 16 core operations in our DSL (see Tab. 1), such as `filter` or `groupby`. All tools adhere to a standard interface (e.g., `transform(df, **args)`) and accept structured JSON arguments. This design is the cornerstone of the agent's reliability: by forcing the LLM to generate only structured parameters rather than raw code, we mitigate the risks of direct code generation (e.g., LLM-hallucinated syntax, injection vulnerabilities, or use of non-existent `pandas` functions). This abstraction ensures that the agent's actions are constrained, predictable, and robustly callable, preventing the LLM from generating syntactically incorrect or semantically unsafe code.

Table 5: Performance of baseline models on the PARROT test set across difficulty levels. We report EA, PV, and OA in percentages (%). Best results per category are in bold.

Model	Execution Accuracy (EA)				Program Validity (PV)				Operator Accuracy (OA)			
	Easy	Medium	Hard	Overall	Easy	Medium	Hard	Overall	Easy	Medium	Hard	Overall
Zero-shot LLMs (non-reasoning)												
GPT-4o-mini	75.17	59.55	49.79	62.88	87.92	73.83	62.34	76.38	67.08	70.44	69.6	69.22
GPT-4o	**89.03**	**79.06**	**72.38**	**71.00**	89.04	**79.07**	**72.38**	**81.12**	64.24	**70.70**	**74.44**	**69.27**
Gemini-2.5-Pro	80.09	62.09	52.72	66.26	**91.5**	77.65	67.78	80.40	**68.75**	64.54	67.72	66.95
DeepSeek-V3	78.52	63.79	56.07	67.19	89.49	77.09	68.62	67.31	66.70	68.32	72.62	68.54
Zero-shot LLMs (reasoning)												
GPT-o3-mini	**82.10**	68.88	**67.36**	**72.86**	92.39	81.47	79.92	84.70	**67.86**	**70.51**	**72**	**69.91**
GPT-o4-mini	81.87	**69.17**	64.02	72.36	**94.85**	**83.31**	**82.01**	**86.79**	64.47	68.76	68.61	67.36
DeepSeek-R1	77.18	58.84	41.84	61.81	89.49	72.70	55.23	75.09	67.45	68.06	67.39	67.75
Fine-tuned LLMs												
Qwen2.5-Coder-1.5B	67.79	59.41	50.63	60.59	77.85	70.58	64.02	71.79	67.11	70.33	66.55	68.65
Qwen2.5-Coder-3B	83.67	69.73	**68.62**	74.01	**91.05**	80.91	**82.01**	84.35	82.03	81.76	**81.93**	81.87
Qwen2.5-Coder-7B	**84.12**	**69.87**	68.20	**74.15**	91.5	**82.18**	81.59	**85.07**	**82.96**	**82.96**	81.59	**82.53**

Stateful Validator. This component acts as the feedback mechanism that grounds the agent's reasoning in executable reality. It serves two functions: execution and observation. First, it receives the "Action" (e.g., `{'op': 'groupby', ...}`) from the orchestrator, invokes the corresponding *Modular Tool*, and executes it on the current table state $\mathbf{df}_t$. Second, it generates the "Observation" for the next reasoning step. This observation is not the full table data; it is a concise summary of the new state $\mathbf{df}_{t+1}$, including: (a) the updated table schema (column names and types), (b) a few sample rows (e.g., $df.head(5)$) to ground the LLM's understanding of the data values, and (c) a descriptive error message if the execution failed. This component effectively acts as the runtime-equivalent of the *Schema Propagation Mechanism (SPM)* (Sec. 4.2), ensuring that every step in the agent's plan is validated against a real, dynamically evolving schema, rather than a hypothesized one. In summary, these three components form a robust, self-correcting system. The agent's "mind" (Cognitive Orchestrator) is decoupled from its "hands" (Modular Toolsets), while the "nerves" (Stateful Validator) provide a high-fidelity feedback loop. It allows Pipeline-Agent to chain complex operations, correct errors in real-time, and dynamically adapt its plan based on intermediate results, thus overcoming the static, one-shot limitations of prior methods.

7 Experiments and Analysis

7.1 Experimental Setup

We conduct a comprehensive evaluation of various models on the PARROT benchmark to assess their capabilities in tackling the *Text-to-Pipeline* task. Our experimental setup is detailed below.

LLM Baselines. To establish a comprehensive performance baseline on PARROT, we evaluate a diverse set of LLMs, encompassing both zero-shot inference capabilities of proprietary models and the performance of fine-tuned open-source models. These models represent the current state-of-the-art in code generation and natural language understanding: (1) **Zero-shot LLMs:** We utilize several leading API-based models, including GPT-4o [31], GPT-4o-mini [31], Gemini-2.5-Pro [33], and DeepSeek-V3 [28]. These models are prompted with the task instruction and table schema without any task-specific fine-tuning. (2) **Fine-tuned LLMs:** We fine-tune a series of strong open-source code generation models: Qwen2.5-Coder-1.5B [16], Qwen2.5-Coder-3B [16], and Qwen2.5-Coder-7B [16]. These models are trained on the PARROT training set, which is derived from the PARROT benchmark, to adapt them specifically to the *Text-to-Pipeline* task.

Structured Generation Approaches. We evaluate three distinct target output formalisms for generating executable data pipelines to understand their efficacy in representing and synthesizing complex transformations: (1) **Text-to-Code (Pandas)**: Direct generation of executable Pandas code. (2) **Text-to-SQL**: Generation of SQL statements, assessing the adaptability of SQL-centric approaches to broader data preparation tasks not typically addressed by SQL. (3) **Text-to-Pipeline**: Our primary approach, where models generate operation sequences in our DSL, subsequently compiled to Pandas. The DSL is architected for modular planning, type safety through enforcement, and schema-aware validation, aiming for a more robust generation pathway.

Planning and Agent-based Approaches. To assess the capabilities of more sophisticated reasoning strategies for multi-step pipelines, we evaluate several planning and agent-based paradigms: (1) **Tool Calling API [32]**: LLMs are instructed to generate the full multi-step execution plan or program for a PARROT task in a single pass, simulating a direct tool-use scenario. (2) **Plan-and-Solve [37]**: This approach first generates a high-level operation plan (e.g., a sequence of operations), then synthesizes the executable program based on that plan. (3) **Chain-of-Tables [38]**: This strategy involves evolving and manipulating intermediate tabular states throughout the reasoning chain to guide the transformation process for tasks from PARROT. (4) **Pipeline-Agent**: Our proposed agent that iteratively predicts an operation, executes it on the current table, and reflects on the result. By leveraging intermediate states,

Table 6: Evaluation results of agent methods on the PARROT test set. We report EA, PV, and OA in percentages (%). Best results per category are in bold. OA is omitted for Chain-of-Tables as it does not produce atomic operation sequences.

Model	Execution Accuracy (EA)				Program Validity (PV)				Operator Accuracy (OA)			
	Easy	Medium	Hard	Overall	Easy	Medium	Hard	Overall	Easy	Medium	Hard	Overall
Tool Calling	**71.62**	58.36	47.67	60.48	86.48	66.53	58.13	71.07	67.79	47.15	35.40	51.31
Plan-and-Solve	63.69	43.19	30.23	47.40	74.52	52.53	38.37	57.00	61.04	42.83	30.09	46.36
Chain-of-Tables	50.67	17.90	9.30	26.27	81.76	74.32	79.07	77.39	-	-	-	-
Pipeline-Agent												
- GPT-4o-mini	70.27	61.08	54.65	62.72	88.51	76.26	67.44	78.41	67.56	52.04	41.02	54.79
- GPT-4o	**77.70**	**78.21**	**67.44**	**76.17**	96.62	**88.33**	**82.56**	**89.82**	**78.04**	**72.32**	**65.91**	**72.92**
- Deepseek-V3	66.89	60.70	48.84	60.49	**91.21**	78.99	70.93	81.26	68.47	58.18	47.53	59.42

Table 7: Performance comparison of different target generation formalisms on the PARROT test set. We report EA, PV, and OA in percentages (%). Best results per category are in bold.

Model	Execution Accuracy (EA)				Program Validity (PV)				Operator Accuracy (OA)			
	Easy	Medium	Hard	Overall	Easy	Medium	Hard	Overall	Easy	Medium	Hard	Overall
Text-to-SQL	10.81	2.59	0	3.05	**93.92**	67.31	55.81	73.31	-	-	-	-
Text-to-Code	48.64	32.59	28.57	33.8	70.27	57.77	53.74	58.45	-	-	-	-
Text-to-Pipeline	**75.17**	**59.55**	**49.79**	**62.88**	87.92	**73.83**	**62.34**	**76.38**	**67.08**	**70.44**	**69.6**	**69.22**

it enables context-aware planning and handles schema evolution in complex PARROT tasks.

Implementation Details. For zero-shot LLM evaluations, we utilized consistent prompt templates. Each prompt included: (1) a clear definition of the *Text-to-Pipeline* task. (2) the input table (column names and data types). (3) 10 sample rows from the input table to provide context on data values, and (4) the natural language instruction. We used a temperature of 0.7 for evaluation purposes. For fine-tuned models, we performed supervised fine-tuning on the PARROT training set. Models were trained for 3 epochs using the AdamW [19] optimizer with a learning rate of 1e-3 and a batch size of 16. A learning rate scheduler of linear decay with warmup was employed. Early stopping was triggered based on loss on a dedicated validation split of PARROT to prevent overfitting. All fine-tuning experiments were conducted on a cluster of 4 NVIDIA 4090 (24GB) GPUs. We use the GPT-4o-mini as the default LLM backbone unless specified otherwise. For Text-to-Pandas, the DSL-to-Pandas compilation is rule-based and deterministic. For Text-to-SQL, the generated SQL is subsequently executed via the SQLite engine. We have provided partial comments for operator types that are not supported by SQL. More experimental details and prompts design can be found in App. A and App. B.

7.2 Evaluation Results

Performance across Difficulty Levels. Tab. 5 reports the performance of LLM baselines across Easy, Medium, and Hard tasks in PARROT . Zero-shot models with explicit reasoning prompts (e.g., GPT-o3-mini) outperform non-reasoning variants, achieving **72.86%** EA vs. **71.00%** (GPT-4o). GPT-4o (non-reasoning) performs well on Easy (**89.03%**) and Medium (**79.06%**) tasks, while showing a struggle with competitive Hard tasks (EA **71.00%**). Fine-tuned models deliver the largest gains. Qwen2.5-Coder-7B achieves **74.15%** overall EA and **68.20%** on Hard tasks—outperforms leading closed-source LLMs (Deepseek and GPT series) despite having fewer parameters. It also attains top PV (**85.07%**) and OA (**82.53%**), highlighting the quality of our supervised data. Across all models, performance consistently declines from Easy to Hard tasks, with a sharper drop in EA than PV. This suggests that while many models produce syntactically valid outputs (high PV), ensuring execution correctness in complex, multi-step settings remains a core challenge. For instance, Qwen2.5-Coder-3B's EA drops from **83.67%** (Easy) to **68.62%** (Hard), reflecting the compositional difficulty inherent.

Impact of Structured Target Generation. Our analysis confirms a core hypothesis of this work: targeting our DSL vastly outperforms direct code generation. As shown in Tab. 7, it achieves an overall EA of **62.88%**, exceeding Text-to-Code (Pandas, **33.80%**) by **+29.08** points and Text-to-SQL (**3.05%**) by a wide margin. This advantage holds across difficulty levels, particularly on Easy (**75.17%**) and Medium (**59.55%**) tasks. Although Text-to-SQL yields high PV (**73.31%**), its low EA indicates poor semantic grounding in complex, multi-step tasks beyond standard SQL patterns. Text-to-Pipeline also achieves the highest OA (**69.22%**), reflecting stronger structural fidelity. OA is not reported for Text-to-Code and Text-to-SQL since their outputs do not use atomic operations as our DSL. These results highlight the DSL's effectiveness in enabling more accurate planning and execution for compositional data preparation tasks.

Efficacy of Planning and Agent-based Approaches. Structured planning with agent-based methods significantly improves multi-step reasoning. As shown in Tab. 6, our proposed Pipeline-Agent (GPT-4o) achieves the highest overall EA of **76.17%**, outperforming Tool Calling API (**60.48%**) and Plan-and-Solve (**47.40%**) by large

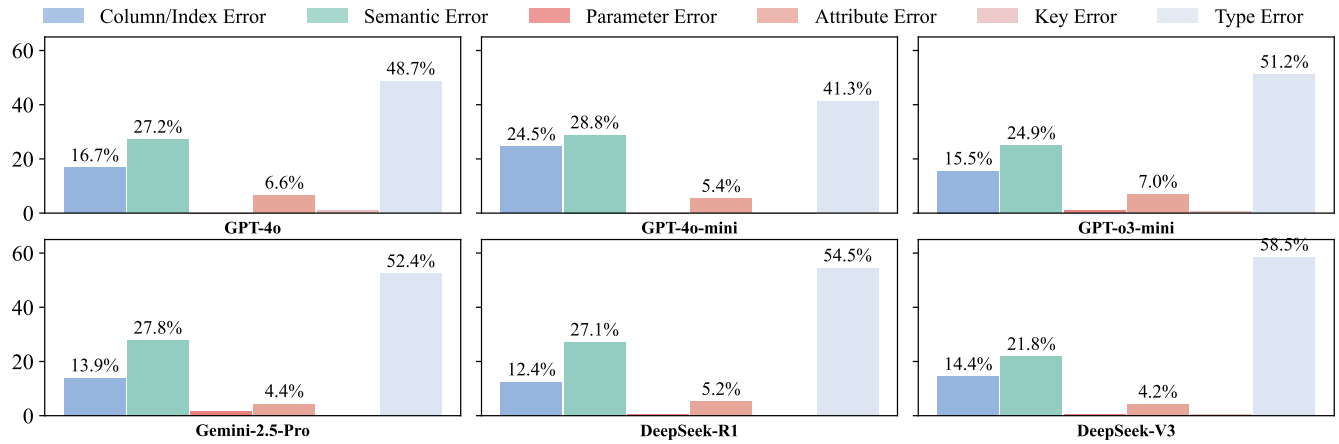

Figure 8: Distribution of error types across six large language models.

margins. Even with the weaker GPT-4o-mini backbone, Pipeline-Agent still outperforms both baselines (**62.72%** EA), confirming its robustness. Chain-of-Tables, while exhibiting strong PV on Hard tasks (**79.07%**), suffers from poor EA (**26.27%**), likely due to unstable intermediate manipulations. Notably, upgrading the Pipeline-Agent's backbone from GPT-4o-mini to GPT-4o yields a substantial **+13.45** point EA gain, demonstrating its ability to leverage stronger models effectively. Across difficulty levels, Pipeline-Agent (GPT-4o) maintains high accuracy: **77.70%** (Easy), **78.21%** (Medium), and **67.44%** (Hard). Its high OA (**72.92%** with GPT-4o, **54.79%** with GPT-4o-mini) further highlights its strength in structuring valid transformation sequences. Since Chain-of-Tables produces non-symbolic intermediate states, OA is not reported for this method. These results underscore that agent-based strategies with explicit planning are key to tackling the compositional challenges, especially when paired with capable LLMs.

7.3 Error Analysis and Case Study

Error Analysis. We conduct a comparative error analysis across six leading LLMs, as illustrated in Fig. 8, to understand common failure modes. We summarize key failure modes below, which we find map directly to our two identified core challenges: (1) **Type errors** consistently account for the largest proportion across all models, ranging from 41.3% (GPT-4o-mini) to 54.5% (DeepSeek-R1). These errors typically stem from applying transformations to columns with incompatible data types, such as aggregating over non-numeric fields or sorting categorical columns without conversion. Models with weaker schema-tracking capabilities (e.g., DeepSeek-R1/V3) exhibit higher type error rates. This is a primary manifestation of failed semantic parameter grounding. (2) **Semantic errors** constitute the second largest category, with models like GPT-4o and Gemini-2.5-Pro showing around 27% semantic failures. These reflect incorrect or missing operations, broken logic chains, or hallucinated steps that deviate from the task intent. The relatively lower semantic error rates of DeepSeek models may indicate a conservative generation style, albeit at the cost of lower execution accuracy. This directly reflects a failure in multi-step compositional logic.

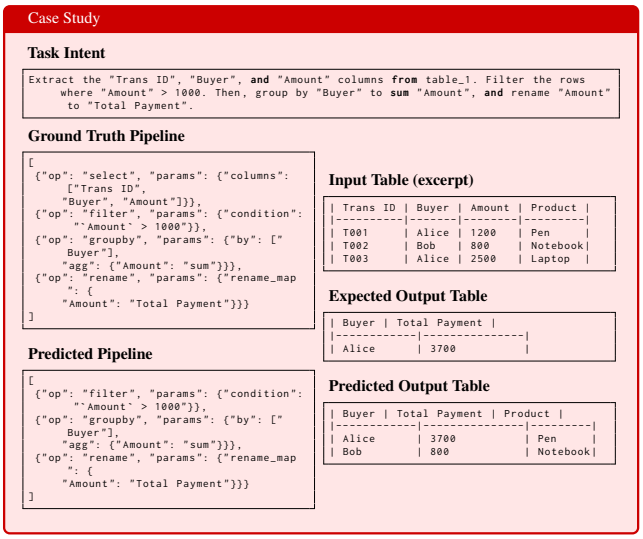

Figure 9: Case study with zero-shot prompting.

(3) **Column/index errors** are more prominent in GPT-o3/4-mini and Gemini-2.5-Pro, often resulting from misaligned references due to schema evolution (e.g., renaming or selection). This suggests limitations in maintaining coherent schema state across steps. This is another key failure mode of semantic parameter grounding under dynamic schema changes. (4) **Attribute errors** (e.g., wrong aggregation function or sorting order) appear in 3%–7% of cases across models, indicating shallow mapping between instruction semantics and operator parameters. Overall, these results highlight the *diverse failure modes* of different LLMs in *Text-to-Pipeline*, and emphasize the importance of schema tracking, operator grounding, and long-range reasoning in achieving robust program synthesis. This also represents a fine-grained failure in semantic grounding.

Case Study. We examine a failure case from zero-shot prompting, shown in Fig. 9, where the model is instructed to perform a series of operations: extract relevant columns (`select`), filter rows by value

(`filter`), aggregate results (`groupby`), and finally rename a column (`rename`). The user instruction states:

> *"Extract the 'Transaction ID', 'Buyer Name', and 'Amount Paid' columns from table_1. Filter the rows where 'Amount Paid' is greater than 1000. Then, group the data by 'Buyer Name' and calculate the total sum of 'Amount Paid' for each buyer. Finally, rename 'Amount Paid' to 'Total Payment'."*

The ground truth program includes all four operations in the specified order. However, the model-generated pipeline omits the initial `select` step and begins directly with `filter`, followed by `groupby` and `rename`. This mistake illustrates the critical failure modes as we mentioned before: (1) Failure in multi-step compositional Logic. The model fails to generate the correct, order-dependent sequence, omitting the initial `select` step entirely. It fails to reason that this column-scoping operation is a necessary compositional precursor, forcing downstream operators to process an incorrect (wider) schema and deviating from the required plan. (2) Failure in semantic grounding. While the model correctly parameterizes the operations it generates (e.g., 'by=["Buyer"]'), it fails to ground the entire instruction—specifically, it ignores the "Extract..." directive. This failure to map the full semantic intent of the NL to a complete symbolic program is a fundamental grounding failure. This case study demonstrates that even when a model avoids simple parameter errors, it can still fail compositionally and fail to ground the user's complete intent, thus validating the challenges of PARROT.

8 Conclusion

In this work, we defined the *Text-to-Pipeline* task and constructed PARROT, a large-scale benchmark designed to support systematic evaluation. Our experiments validated PARROT's role as a critical and difficult testbed. We demonstrated that the choice of representation is crucial, as direct code generation fails markedly (e.g., 33.8% EA for Pandas), while our symbolic DSL provides a more robust target. Furthermore, our in-depth error analysis pointed to multi-step compositional logic and semantic grounding as the key bottlenecks. While our proposed *Pipeline-Agent* achieves the SOTA, its significant remaining performance gap underscores the depth of the challenge. Future work could focus on developing agents that better track dynamic table states throughout the pipeline. For instance, reinforcement learning could be employed to train a policy, using rich execution errors as negative reward signals. PARROT provides the community with a critical benchmark for developing the next generation of data preparation agents.

Acknowledgments

This work was supported by the [...] Research Fund of [...] (Number [...]). Additional funding was provided by [...] and [...]. We also thank [...] for contributing [...].

References

[1] [n.d.]. Power Query: Merge queries. https://support.office.com/en-us/article/Merge-queries-Power-Query-fd157620-5470-4c0f-b132-7ca2616d17f9.

[2] [n.d.]. Trifacta: Standardize Using Patterns. https://docs.trifacta.com/display/DP/Standardize+Using+Patterns.

[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] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *PACMPL* (2019).

[5] Chengliang Chai, Jiayi Wang, Yuyu Luo, Zeping Niu, and Guoliang Li. 2022. Data management for machine learning: A survey. *TKDE* 35, 5 (2022), 4646–4667.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv:2107.03374* (2021).

[7] 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*.

[8] Yuhao Deng, Chengliang Chai, Lei Cao, Qin Yuan, Siyuan Chen, Yanrui Yu, Zhaoze Sun, Junyi Wang, Jiajun Li, Ziqi Cao, et al. 2024. Lakebench: A benchmark for discovering joinable and unionable tables in data lakes. *VLDB* (2024).

[9] 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. *arXiv:2412.10422* (2024).

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv:2002.08155* (2020).

[11] Raul Castro Fernandez, Aaron J Elmore, Michael J Franklin, Sanjay Krishnan, and Chenhao Tan. 2023. How large language models will disrupt data management. *VLDB* (2023).

[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 Narasayya, and Surajit Chaudhuri. 2018. Transform-data-by-example (tde) an extensible search engine for data transformations. *VLDB* (2018).

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

[15] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv:2105.09938* (2021).

[16] 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:2409.12186* (2024).

[17] Zhongjun Jin, Michael R Anderson, Michael Cafarella, and HV Jagadish. 2017. Foofah: Transforming data by example. In *SIGMOD*.

[18] 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).

[19] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv:1412.6980* (2014).

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

[21] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv:2411.07763* (2024).

[22] Hongxin Li, Jingran Su, Yuntao Chen, Qing Li, and ZHAO-XIANG ZHANG. 2023. Sheetcopilot: Bringing software productivity to the next level through large language models. *NeurIPS* (2023).

[23] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. Codes: Towards building open-source language models for text-to-sql. *SIGMOD* (2024).

[24] Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. 2015. A diversity-promoting objective function for neural conversation models. *arXiv:1510.03055* (2015).

[25] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *NeurIPS* (2023).

[26] Peng Li, Yeye He, Cong Yan, Yue Wang, and Surajit Chaudhuri. 2023. Auto-Tables: Synthesizing Multi-Step Transformations to Relationalize Tables without Using Examples. *VLDB* (2023).

[27] Xingjun Li, Yizhi Zhang, Justin Leung, Chengnian Sun, and Jian Zhao. 2023. Edassistant: Supporting exploratory data analysis in computational notebooks with in situ code search and recommendation. *TiiS* 13, 1 (2023), 1–27.

[28] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv:2412.19437* (2024).

[29] Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. 2024. SpreadsheetBench: towards challenging real world spreadsheet manipulation. *arXiv:2406.14991* (2024).

[30] Zan Ahmad Naeem, Mohammad Shahmeer Ahmad, Mohamed Eltabakh, Mourad Ouzzani, and Nan Tang. 2024. RetClean: Retrieval-Based Data Cleaning Using LLMs and Data Lakes. *VLDB* (2024).

[31] R OpenAI. 2023. Gpt-4 technical report. arxiv 2303.08774. *View in Article* 2 (2023), 13.

[32] 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.

[33] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv:2403.05530* (2024).

[34] Rishabh Singh. 2016. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *VLDB* (2016).

[35] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by output. In *SIGMOD*.

[36] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *SIGPLAN*.

[37] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv:2305.04091* (2023).

[38] 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. *arXiv:2401.04398* (2024).

[39] Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. 2024. Selfcodealign: Self-alignment for code generation. *NeurIPS* (2024).

[40] Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xeron Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, et al. 2025. Tablebench: A comprehensive and complex benchmark for table question answering. In *AAAI*.

[41] Cong Yan and Yeye He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *SIGMOD*.

[42] Junwen Yang, Yeye He, and Surajit Chaudhuri. 2021. Auto-pipeline: synthesizing complex data pipelines by-target using reinforcement learning and search. *VLDB* (2021).

[43] 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*.

[44] 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. *arXiv:1809.08887* (2018).

[45] Daochen Zha, Zaid Pervaiz Bhat, Kwei-Herng Lai, Fan Yang, Zhimeng Jiang, Shaochen Zhong, and Xia Hu. 2025. Data-centric artificial intelligence: A survey. *Comput. Surveys* 57, 5 (2025), 1–42.

[46] Haochen Zhang, Yuyang Dong, Chuan Xiao, and Masafumi Oyamada. 2024. Jellyfish: Instruction-Tuning Local Large Language Models for Data Preprocessing. In *EMNLP*.

[47] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv:2401.07339* (2024).

[48] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A survey on large language models for software engineering. *arXiv:2312.15223* (2023).

[49] Wei Zhao, Zhitao Hou, Siyuan Wu, Yan Gao, Haoyu Dong, Yao Wan, Hongyu Zhang, Yulei Sui, and Haidong Zhang. 2024. NL2Formula: Generating Spreadsheet Formulas from Natural Language Queries. In *EACL*.

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

[51] Erkang Zhu, Yeye He, and Surajit Chaudhuri. 2017. Auto-join: Joining tables by leveraging transformations. *VLDB* (2017).

[52] Yaoming Zhu, Sidi Lu, Lei Zheng, Jiaxian Guo, Weinan Zhang, Jun Wang, and Yong Yu. 2018. Texygen: A benchmarking platform for text generation models. In *SIGIR*.

A Details of Experiments

A.1 LLM Baselines

Zero-shot LLMs. To ensure consistent evaluation across models, we adopt a unified prompt format detailed in Section B.2. This schema-aware and instruction-driven template provides clear guidance on operation selection and output structure. During inference, all models are run with a temperature of 0.7 and a maximum output length of 4000 tokens. For models without a `temperature` parameter (e.g., `o3-mini`, `o4-mini`), this setting is omitted.

Fine-tuned LLMs. To align large language models with data preparation tasks, we fine-tune several instruction-tuned variants of Qwen2.5-Coder, including 1.5B, 3B, and 7B models. Training is performed on four NVIDIA RTX 4090 GPUs using `torchrun` with mixed precision (bf16). The batch size is set to 2 for the 1.5B model and 1 for the 3B and 7B models. Each model is trained for three epochs with a sequence length of 4096.

A.2 Structured Generation Approaches

Text-to-Code (Pandas). The Text-to-Pandas module converts natural language instructions into executable `pandas` code. It employs structured, context-aware prompts that help the model identify operation types (e.g., `filter`, `groupby`, `pivot`) and generate semantically correct, executable code based on the table schema. This design lowers the barrier to complex transformations and supports reproducible evaluation. Detailed templates are provided in Section B.3.

Text-to-SQL. For the Text-to-SQL setting, we simulate a realistic database environment using SQLite, where all benchmark tables are stored as normalized relational tables with explicit schemas and foreign key relationships to support joins. This setup ensures that model-generated SQL queries are executable and verifiable. Structured prompt templates (Section B.4) pair natural language instructions with schema descriptions to promote consistent and accurate SQL generation.

A.3 Planning and Agent-based Methods

Tool and Environment Setup. Each data preparation operation is implemented as a callable tool (Section A.4). Agents invoke these tools via LangChain's `tool calling` API within a controlled Python environment that provides access to input tables, schemas, and intermediate states.

Tool Calling Agent. This baseline directly maps user instructions to tool invocations using LLM reasoning. The input includes a table preview, natural language instruction, and standardized tool schema documentation. We evaluate GPT-4o, GPT-4o-mini, and DeepSeek-V3 under this setting.

Plan-and-Solve Agent. Following a two-phase paradigm, this agent first generates a high-level transformation plan (e.g., `filter` → `groupby` → `sort`) and then executes each operation sequentially through tool calls. This decouples reasoning from execution but lacks feedback integration.

Chain-of-Tables Agent. Inspired by [38], this agent explicitly maintains intermediate tabular states, enabling stepwise manipulation and reasoning across evolving tables. It adapts plan–execute–react loop to operate over our DSL-based transformation layer instead of SQL.

Pipeline-Agent. We propose Pipeline-Agent, a unified framework that couples reasoning and execution in a closed-loop cycle. At each step, the agent reasons about the current table state, selects the next operation, executes it through the corresponding tool, and incorporates feedback into subsequent reasoning. This tight integration of reasoning and execution enables robust handling of complex, multi-step transformations. We evaluate Pipeline-Agent with GPT-4o, GPT-4o-mini, and DeepSeek-V3 to analyze the impact of reasoning capacity on performance.

A.4 Tool Definitions and Interaction

Each tool in the Pipeline-Agent is a self-contained module that encapsulates a particular transformation logic. Tools expose a unified interface for execution and are compatible with structured reasoning inputs from LLMs. During the agent's reasoning process, these tools are dynamically selected and applied, enabling seamless integration into the closed-loop pipeline. The toolset spans essential operations such as filtering, grouping, sorting, pivoting, and more, supporting a wide range of data preparation needs.

Tools Definition

```
class BaseOpInput(BaseModel):
    table_names: str

class FilterInput(BaseOpInput):
    condition: str

class SortInput(BaseOpInput):
    by: List[str]; ascending: List[bool]

class PivotInput(BaseOpInput):
    index: str
    columns: str
    values: str
    aggfunc: str

class StackInput(BaseOpInput):
    id_vars: List[str]
    value_vars: List[str]

class ExplodeInput(BaseOpInput):
    column: str
    split_comma: bool

class WideToLongInput(BaseOpInput):
    subnames: List[str]
    i: List[str]
    j: str
    sep: str
    suffix: str

class UnionInput(BaseOpInput):
    left_table: str
```

```
    right_table: str
    how: str

class JoinInput(BaseOpInput):
    left_table: str
    right_table: str
    left_on: str
    right_on: str
    how: str
    suffixes: List[str]

class TransposeInput(BaseOpInput): pass

class DropnaInput(BaseOpInput):
    subset: List[str];
    how: str

class DeduplicateInput(BaseOpInput):
    subset: Union[List[str], None]
    keep: str

class TopKInput(BaseOpInput):
    k: int

class SelectInput(BaseOpInput):
    columns: List[str]

class CastInput(BaseOpInput):
    column: str
    dtype: str

class RenameItem(BaseModel):
    old_name: str
    new_name: str

class RenameInput(BaseOpInput):
    rename_items: List[str]

class AggItem(BaseModel):
    column: str
    agg_func: str

class GroupByInput(BaseOpInput):
    by: List[str]
    aggregations: List[AggItem]
```

A.5 Difficulty Level Definition and Illustrative Examples

Difficulty Level Definitions. To simulate tasks of varying complexity, we classify operator chains into three difficulty levels based on chain length and the semantic nature of operations involved:

- **Easy** (1–3 steps): These chains typically consist of atomic, table-local operations such as `filter`, `sort`, `select`, or `dropna`.
- **Medium** (4–6 steps): Chains in this category often involve combinations of aggregation and light integration, such as `groupby-agg`, `rename`, or simple `join` operations.
- **Hard** (7–8 steps): These chains incorporate multi-table joins, nested reshaping (e.g., `pivot` following `groupby`), and schema-evolving transformations requiring global reasoning.

This stratified scheme enables us to evaluate model performance across various compositional depths and reasoning challenges, while ensuring control over the distribution of task complexity in the benchmark.

Illustrative Examples. There are three examples for each level of difficulty. Each example consists of a natural language instruction, the corresponding DSL operator sequence, the compiled code, and input/output tables. These examples demonstrate different reasoning demands, such as filtering, sorting and multi-table aggregation like union and joining with nested operations.

Example (Easy)

```
{
  "instruction": "Sort the data in 'table_1' by
  ↪  'Civil Liberties' and 'President' in
  ↪  ascending order to organize the entries
  ↪  accordingly.",
  "input_table": "input_E001.csv",
  "output_table": "output_E001.csv",
  "transformation_sequence": [
  "op": "sort", "params": {"by": ["Civil
  ↪  Liberties","President"],"ascending":
  ↪  [true,true]},}
  ],
  "gold_code": "df.sort_values(by=['Civil
  ↪  Liberties', 'President'],
  ↪  ascending=[True, True]))"
}
```

Input Table (E001).

Year	Political Rights	Civil Liberties	Status	President
1972	6	6	Not Free	Hamani Diori
1973	6	6	Not Free	Hamani Diori
1974	7	6	Not Free	Hamani Diori
1975	7	5	Not Free	Seyni Kountché
1976	7	5	Not Free	Seyni Kountché
1977	7	5	Not Free	Seyni Kountché

Output Table (E001).

Year	Political Rights	Civil Liberties	Status	President
1975	7	5	Not Free	Seyni Kountché
1976	7	5	Not Free	Seyni Kountché
1977	7	5	Not Free	Seyni Kountché
1972	6	6	Not Free	Hamani Diori
1973	6	6	Not Free	Hamani Diori
1974	7	6	Not Free	Hamani Diori

Example (Medium)

```
{
  "instruction": "Start by excluding the rows
  ↪  where the 'Year' is 2013. Then, remove
  ↪  duplicate rows in table_1, keeping the
  ↪  last occurrence for each duplicate.
  ↪  After that, group the resulting data by
  ↪  'Name', computing the minimum of 'Number
  ↪  of Contestants' for each group. Finally,
  ↪  sort the grouped data by 'Number of
  ↪  Contestants' in ascending order.",
  "input_table": "input_M001.csv",
  "output_table": "output_M001.csv",
  "transformation_sequence": [
    {"op": "filter", "params": {"column":
    ↪  "Year", "condition": "!= 2013"}},
    {"op": "duplicate", "params": {"subset":
    ↪  null, "keep": "last"}},
    {"op": "groupby", "params": {"by":
    ↪  ["Name"], "agg": {"Number of
    ↪  Contestants": "min"}}},
    {"op": "sort", "params": {"by": ["Number of
    ↪  Contestants"], "ascending": [true]}}
  ],
  "gold_code": "df.query('Year != 2013')
    .drop_duplicates(keep='last')
    .groupby('Name', as_index=False)
    .agg({'Number of Contestants': 'min'})
    .sort_values(by='Number of Contestants',
    ascending=True)"
}
```

Input Table (M001).

Name	Number of Contestants	Number of Approved	Year
University of Chile	253	125	2014
Pontifical Catholic University of Chile	202	118	2014
University of Concepción	108	46	2013
University of Chile	74	33	2015
Pontifical Catholic University of Chile	69	31	2015

Output Table (M001).

Name	Number of Contestants
Pontifical Catholic University of Chile	69
University of Chile	74

Example (Hard)

```
{
  "instruction": "Start by performing a right
  ↪  join between table_1 and table_2 on 'ship
  ↪  id' with suffixes '_left' and '_right'.
  ↪  Then, remove any rows with missing values
  ↪  across all columns. Next, explode the
  ↪  'location' column by splitting its values
  ↪  at commas. Group the data by the 'type'
  ↪  column, counting occurrences of 'speed
  ↪  knots' and calculating the mean of 'ship
  ↪  id'. Sort the grouped data by 'speed
  ↪  knots' and 'ship id' in descending order.
  ↪  Deduplicate the results based on 'speed
  ↪  knots' and 'ship id', keeping the first
  ↪  occurrence. Select the top two entries
  ↪  from the sorted results. Finally, rename
  ↪  the columns to 'category' for 'type',
  ↪  'velocity in nautical miles' for 'speed
  ↪  knots', and 'vessel identifier' for
  ↪  'ship id'.",
  "input_table": [
    "input_H001_ships.csv",
    "input_H001_ship_missions.csv"
  ],
  "output_table": "output_H001.csv",
  "transformation_sequence": [
    { "op": "join", "params": { "on": "ship
    ↪  id", "how": "right", "suffixes":
    ↪  ["_left", "_right"] } },
    { "op": "dropna", "params": { "how": "all"
    ↪  } },
    { "op": "explode", "params": { "column":
    ↪  "location", "split_comma": true } },
    { "op": "groupby", "params": { "by":
    ↪  ["type"], "agg": { "speed knots":
    ↪  "count", "ship id": "mean" } } },
    { "op": "sort_values", "params": { "by":
    ↪  ["speed knots", "ship id"],
    ↪  "ascending": [false, false] } },
    { "op": "deduplicate", "params": {
    ↪  "subset": ["speed knots", "ship id"],
    ↪  "keep": "first" } },
    { "op": "head", "params": { "n": 2 } },
    { "op": "rename", "params": { "rename_map":
    ↪  { "ship id": "vessel identifier",
    ↪  "speed knots": "velocity in nautical
    ↪  miles", "type": "category" } } }
  ],
  "gold_code": "df = (
    table_1.merge(table_2, on='ship id',
    ↪  how='right', suffixes=('_left',
    ↪  '_right'))
    .dropna(how='all')
```

```
    .assign(location=lambda df:
    ↪ df['location'].str.split(','))
    .explode('location')
    .groupby('type', as_index=False)
    .agg({'speed knots': 'count', 'ship id':
    ↪ 'mean'})
    .sort_values(by=['speed knots', 'ship id'],
    ↪ ascending=[False, False])
    .drop_duplicates(subset=['speed knots',
    ↪ 'ship id'], keep='first')
    .head(2)
    .rename(columns={
        'ship id': 'vessel identifier',
        'speed knots': 'velocity in nautical
        ↪ miles',
        'type': 'category'
    })"
}
```

Input Table 1: Ships (H001).

ship id	name	type	nationality	tonnage
1	Corbridge	Cargo ship	United Kingdom	3687
2	Farringford	Battle ship	United States	3146
3	Dromonby	Cargo ship	United Kingdom	3627
4	Author	Cargo ship	United Kingdom	3496
5	Trader	Battle ship	United Kingdom	3608
6	Ariadne	Cargo ship	United States	3035
7	Appam	Battle ship	United Kingdom	7781
8	Clan McTavish	Cargo ship	United States	5816

Input Table 2: Ship_missions (H001).

mission id	ship id	launched year	location	speed (knots)	fate
1	1	1930	Germany	25	Decommissioned
2	2	1930	Germany	25	Decommissioned
3	3	1930	Helsinki, Finland	23	Lost
4	5	1916	Norway	16	Retired
5	6	1931	Uusikaupunki, Finland	23	Decommissioned
6	7	1931	Uusikaupunki, Finland	23	Decommissioned
7	8	1932	Turku, Finland	23	Lost

Output Table (H001).

category	velocity in nautical miles	vessel identifier
Cargo ship	7	4.875
Battle ship	4	5.25

B Prompt Design

B.1 Prompts for Data Synthesis

In this section, we detail the prompts employed in our data synthesis framework. As previously mentioned in Section 4, we primarily use LLMs for generating and refining instructions, as well as for verifying the consistency between natural language instructions and the operator chain in the DSL. The specifics of these prompts are as follows:

Prompt for Instruction Generation

```
You are a data preparation expert. I have some
↪ related input tables and a target table,
↪ where the target table is obtained by
↪ transforming the input tables. Based on the
↪ transformation relationship between them,
↪ please generate a clear natural language
↪ instruction that describes how to transform
↪ the input tables into the target table.

The transformation operations and their
↪ detailed parameters are as follows:
{transform_chain_str}

Input Tables (First 10 Rows):
{input_table_str}

Target Table (First 10 Rows):
{target_table_str}

Please generate a clear and natural data
↪ preparation instruction in English. The
↪ instruction should explicitly describe the
↪ required transformation steps and clearly
↪ state the table names involved, without
↪ mentioning specific programming languages
↪ or function names. Use terminology from
↪ the data analysis domain and consider the
↪ purpose and effect of the operations.
Your instructions just need to clearly
↪ describe the conversion chain without
↪ describing additional operations.
Your instruction should follow the format:
Instruction: [Your data preparation
↪ instruction]
```

Prompt for Instruction Refinement

```
Based on the following data preparation task
↪ description, generate a natural language
↪ statement expressing the user's intent.
Concise, Action-Oriented Language: Focus on
↪ the core actions and remove unnecessary
↪ details. Keep the language clear and direct
↪ to highlight the transformation intent.
Clarification of Key Tables and Columns:
↪ Maintain essential table names and columns,
↪ but express them in a natural,
↪ straightforward way.
```

```
Simplified Descriptions of Complex Steps:
↪ Emphasize the main objectives (sorting,
↪ filtering, deduplication) without diving
↪ into excessive details, unless they are
↪ crucial for the context.
Necessary details need to be preserved such as
↪ the suffix of the join, the way the
↪ de-duplication operation is performed
↪ (first or last), etc.

Here are some examples:
---
Task Description: To transform the input tables
↪ into the target table, follow these steps:
1. Begin by performing an inner join between
↪ table_1 and table_2 using the allergy name
↪ column from table_1 and the allergy column
↪ from table_2. This will combine records
↪ from both tables where there is a match on
↪ these columns, while including the allergy
↪ name and allergy type from table_1 along
↪ with the stuid from table_2.
2. Next, group the resulting dataset by the
↪ allergy name (now included in the joined
↪ table) and aggregate the data by counting
↪ the number of unique stuid entries for each
↪ allergy name. This will give you the total
↪ number of students associated with each
↪ allergy.
3. After aggregating, sort the grouped data
↪ first by the count of stuid in ascending
↪ order and then by allergy name in
↪ descending order. This will organize the
↪ data based on the number of students and
↪ the names of the allergens.
4. From the sorted data, select the top 7
↪ entries based on the highest counts of
↪ students. This step ensures that we focus
↪ only on the most significant allergens.
5. Rename the columns in the resulting dataset
↪ by changing allergy name to allergen and
↪ stuid to student ID to make the column
↪ names more intuitive.
6. Apply a filter to retain only those records
↪ where the student ID (which now represents
↪ the count of students) is greater than or
↪ equal to 3. This will help in identifying
↪ the allergens that have a notable number of
↪ students associated with them.
7. Remove any duplicate entries from the
↪ filtered dataset to ensure that each
↪ allergen-student ID combination is unique.
8. Finally, perform a sort on the deduplicated
↪ data by student ID in ascending order and
↪ allergen in descending order to achieve
↪ the desired final format. Following these
↪ steps will yield a table that lists
↪ allergens along with the count of students
↪ associated with each, structured as
↪ specified in the target table.
User Intent: Start by performing an inner join
↪ between table_1 and table_2 on 'allergy
↪ name' and 'allergy', with suffixes '_left'
↪ and '_right'. Then, group the data by
↪ allergy name and count the number of
↪ 'stuid' entries for each allergen to
↪ determine the number of students associated
↪ with each allergy. After grouping, sort the
↪ data first by the student count in
↪ ascending order and then by allergy name in
↪ descending order. Select the top 7 entries.
↪ Rename the columns to change allergy name
↪ to allergen and stuid to student ID for
↪ clarity. Apply a filter to keep only the
↪ records where the student ID is 3 or
↪ greater. Deduplicate the data, keeping the
↪ first occurrence of each duplicate entry
↪ to ensure uniqueness. Finally, sort the
↪ deduplicated dataset by student ID in
↪ ascending order and allergen in descending
↪ order to produce the final result.
---
Task Description: First, combine the two input
↪ tables, table_1 and table_2, by performing
↪ a union operation to consolidate all
↪ records, including duplicates. Next, pivot
↪ the resulting table to reorganize the data,
↪ setting the station names (STN_NAM) as the
↪ index, the data provider (DATA_PVDR) as
↪ the columns, and using the minimum
↪ longitude (LONGITUDE) as the values. After
↪ pivoting, rename the column STN_NAM to
↪ Station Name. Then, filter the table to
↪ keep only the rows where the data provider
↪ is "NAV CANADA". Following this, remove any
↪ rows that contain missing values in the
↪ "NAV CANADA" column. Convert the data type
↪ of the "NAV CANADA" column to string. Next,
↪ ensure there are no rows where "NAV CANADA"
↪ is equal to itself (this condition might be
↪ meant for data cleansing or error checking).
↪ Finally, deduplicate the entries based on
↪ the "NAV CANADA" column while keeping the
↪ last occurrence of each duplicate. The
↪ result will be your target table with the
↪ columns DATA_PVDR and NAV CANADA.
```

```
User Intent: Begin by performing a union
↪ operation on table_1 and table_2 to
↪ consolidate all records, including
↪ duplicates. Then, pivot the resulting table
↪ with the station names (STN_NAM) as the
↪ index, the data provider (DATA_PVDR) as
↪ the columns, and use the minimum longitude
↪ (LONGITUDE) as the values. Rename the
↪ STN_NAM column to "Station Name" for
↪ clarity. Next, select the only column "NAV
↪ CANADA", and remove any rows with missing
↪ values in the "NAV CANADA" column. Convert
↪ the "NAV CANADA" column to a string data
↪ type and ensure that there are no rows
↪ where "NAV CANADA" is equal to itself.
↪ Finally, deduplicate the data based on the
↪ "NAV CANADA" column, keeping the last
↪ occurrence of each duplicate entry.
---
Task Description: First, reshape the data from
↪ the wide format to a long format by
↪ selecting the columns related to 'PUZZLE
↪ B' and 'PUZZLE A', while keeping the
↪ specified index columns intact. After
↪ transforming the data to a long format, you
↪ can apply the explode operation. This
↪ operation will split any column containing
↪ comma-separated values into individual
↪ rows. Next,transforms data from wide
↪ format to long format,it keeps the columns
↪ in id_vars unchanged and stacks the values
↪ from value_vars ("PUZZLE A" and "PUZZLE B")
↪ into two new columns: one for the variable
↪ names and another for the values.
User Intent: First,reshape the data by
↪ collapsing columns that start with "PUZZLE
↪ B" or "PUZZLE A" into a long format, while
↪ keeping the specified index columns
↪ ("Index", "Where are we?") unchanged. The
↪ original suffixes from the column names
↪ are extracted into a new column called var,
↪ using a space as the separator and matching
↪ suffixes with a word character pattern
↪ (\w+).Then, Explode the "PUZZLE B" column
↪ to create separate rows for each puzzle
↪ listed, ensuring that each puzzle is split
↪ by commas first. Next,transforms data from
↪ wide format to long format,the columns
↪ specified in id_vars ("Index", "Where are
↪ we?") remain unchanged and serve as
↪ identifiers for each row. The values in
↪ value_vars ("PUZZLE A" and "PUZZLE B") are
↪ then stacked into two new columns: one for
↪ the variable names and another for the
↪ values.
---

Now, based on the following task description,
↪ generate a user intent statement:
Transformation Chain: {transform_chain}
Task Description: {task_instruction}

Please output only the intent statement,
↪ without explanation or numbering.
```

Prompt for Instruction Verify

```
Task Background: The user has generated an
↪ initial natural language description from
↪ a transformation chain, and then used an
↪ LLM to generate a user intent statement
↪ based on that initial description.
1. **Transformation Chain**:
↪ {transform_chain_str}
2. **Initial Natural Language Description**:
↪ {instruction}
3. **Generated Intent**: {intent_text}

Task Requirement: Assume you are a data
↪ preparation expert. Based on the current
↪ intent, can you infer the correct
↪ conversion chain, including the details of
↪ the parameters?

Output Requirements:
- If the intent allows you to infer a complete
↪ and reasonable transformation chain,
↪ output:
{{
"is_valid": "true",
"intent": "{intent_text}"
}}
- Otherwise, output:
{{
"is_valid": "false",
"intent": "[Rewritten Intent]"
}}

Please return the result in strict JSON format
↪ with no additional explanations.
```

B.2 Prompts for Zero-shot LLMs

This section provides the detailed prompt template designed for zero-shot large language models (LLMs). The prompt is carefully constructed to guide the model in generating accurate and semantically faithful instructions without any fine-tuning. It incorporates schema information and explicit instructions to improve model understanding and output quality. This prompt serves as the basis for consistent evaluation across different zero-shot LLMs.

Prompts for Zero-shot LLMs

```
You are a data expert with extensive knowledge in data preparation pipelines.
Your task is to select operators based on user intent and use them to transform the source tables.

Important notes:
- After selecting the operators, ensure they can be correctly executed, especially keeping variable names consistent.
- Note: Except for the `join` and `union` operations, the result table name remains the same as the source table name. For `join` and `union`, the result table name should follow the format `table_x_table_y_join` or `table_x_table_y_union`.

Below are the available operators:
{
  "operators": [
    {
      "name": "join",
      "pandas_equivalent": "merge",
      "parameters": {
        "left_table": "left_table_name",
        "right_table": "right_table_name",
        "result_table": "table_x_table_y_join",
        "left_on": "left_column",
        "right_on": "right_column",
        "how": "",
        "suffixes": ["", ""]
      },
      "description": "Merge two datasets on a common column with specified input/output table names"
    },
    {
      "name": "union",
      "pandas_equivalent": "concat",
      "parameters": {
        "source_tables": ["table_1", "table_2"],
        "axis": 0,
        "result_table": "table_x_table_y_union",
        "ignore_index": true,
        "how": ["all", "distinct"]
      },
      "description": "Vertically concatenate multiple tables (similar to SQL UNION)"
    },
    {
      "name": "groupby",
      "pandas_equivalent": "groupby",
      "parameters": {
        "source_table": "source_table_name",
        "group_by": ["group_column_1", "group_column_2"],
        "aggregations": {
          "value_column_1": "aggregation_function",
          "value_column_2": "aggregation_function"
        },
        "result_table": "source_table_name"
      },
      "description": "Group data by specified columns and apply aggregation (similar to SQL GROUP BY)"
    },
    {
      "name": "pivot",
      "pandas_equivalent": "pivot_table",
      "parameters": {
        "source_table": "source_table_name",
        "index": ["index_column_1", "index_column_2"],
        "columns": ["column_to_expand"],
        "values": ["value_column"],
        "aggfunc": "aggregation_function",
        "result_table": "source_table_name"
      },
      "description": "Convert long-format data into wide-format (similar to Excel Pivot Table)"
    },
    {
      "name": "unpivot",
      "pandas_equivalent": "melt",
      "parameters": {
        "source_table": "source_table_name",
        "id_vars": ["fixed_column_1", "fixed_column_2"],
        "value_vars": ["column_to_unpivot_1", "column_to_unpivot_2"],
        "var_name": "variable",
        "value_name": "value",
        "result_table": "source_table_name"
      },
      "description": "Convert wide-format data into long-format (similar to SQL UNPIVOT)"
    },
    {
      "name": "explode",
```

```
    "pandas_equivalent": "pd.explode",
    "parameters": {
      "source_table": "source_table_name",
      "result_table": "source_table_name",
      "column": "list_column",
      "split_comma": True or false
    },
    "description": "Expand column values into
    ↪  separate rows (separate them by
    ↪  commas first if necessary)"
  },
  {
    "name": "filter",
    "pandas_equivalent": "query",
    "parameters": {
      "source_table": "source_table_name",
      "condition": "`column_name` operation
      ↪  value",
      "result_table": "source_table_name"
    },
    "description": "Filter rows based on
    ↪  conditions (similar to SQL WHERE)"
  },
  {
    "name": "sort",
    "pandas_equivalent": "sort_values",
    "parameters": {
      "source_table": "source_table_name",
      "by": ["column_1", "column_2"],
      "ascending": [true, false],
      "result_table": "source_table_name"
    },
    "description": "Sort data by specified
    ↪  columns (similar to SQL ORDER BY)"
  },
  {
    "name": "wide_to_long",
    "pandas_equivalent": "pd.wide_to_long",
    "parameters": {
      "source_table": "source_table_name",
      "subnames": ["subname"],
      "i": ["id_column"],
      "j": "var",
      "sep": "",
      "suffix": "",
      "result_table": "source_table_name"
    },
    "description": "Convert wide-format data
    ↪  to long-format"
  },
  {
    "name": "transpose",
    "pandas_equivalent": "transpose",
    "parameters": {
      "source_table": "source_table_name"
    },
    "description": "Transpose rows and
    ↪  columns of a table; no additional
    ↪  parameters needed"
  },
  {
    "name": "rename",
    "pandas_equivalent": "rename",
    "parameters": {
      "source_table": "source_table_name",
      "rename_map": "Dictionary mapping old
      ↪  column names to new names"
    },
    "description": "Rename columns based on
    ↪  the provided mapping"
  },
  {
    "name": "dropna",
    "pandas_equivalent": "dropna",
    "parameters": {
      "source_table": "source_table_name",
      "subset": ["List or single column name
      ↪  to check for missing values"],
      "how": "Deletion strategy: either 'any'
      ↪  or 'all'"
    },
    "description": "Remove rows with missing
    ↪  values in specified columns"
  },
  {
    "name": "deduplicate",
    "pandas_equivalent": "drop_duplicates",
    "parameters": {
      "source_table": "source_table_name",
      "subset": ["List or single column name
      ↪  to determine duplicates"],
      "keep": ["first", "last"]
    },
    "description": "Remove duplicate rows,
    ↪  keeping either the first or last
    ↪  occurrence in each group"
  },
  {
    "name": "topk",
    "pandas_equivalent": "head(k)",
    "parameters": {
      "source_table": "source_table_name",
      "k": "Number of top rows to retain"
    },
    "description": "Select the top k rows
    ↪  after sorting by index or specific
    ↪  criteria"
  },
```

```
    {
      "name": "select",
      "pandas_equivalent": "loc / bracket
      ↪  selection",
      "parameters": {
        "source_table": "source_table_name",
        "columns": "List of column names to
        ↪  keep"
      },
      "description": "Select specified columns
      ↪  from the table"
    },
    {
      "name": "cast",
      "pandas_equivalent": "astype",
      "parameters": {
        "source_table": "source_table_name",
        "column": "Column name to change data
        ↪  type",
        "dtype": "Target data type (e.g.,
        ↪  'int', 'float', 'str')"
      },
      "description": "Convert the data type of
      ↪  the specified column"
    }
  ]
}

Please output the transformation steps from
↪  the input tables to the target table using
↪  the above operations.
The output should follow the JSON format below:
```json
[
  {
    "name": "",
    "parameters": {
    }
  }
]
User Intent:
{USER_INTENT}
Input Tables (First 10 Rows):
{SOURCETABLE}
Please reason step-by-step based on the user
↪  intent, and then provide the result.
The final output should be in JSON format.
"""
```

B.3 Prompts for Text-to-Pandas

To ensure accurate generation of pandas code from natural language instructions, we designed structured prompt template tailored to various data preparation operations. The template guide the model in understanding user intent, interpreting table schemas, and producing syntactically and semantically correct code.

Prompt for Text-to-Pandas

```
I need you to convert natural language into
↪  Pandas code.

dataset schema:
{dataset_schema}

question: {task['question']}

Code Return Guidelines:
1. If you need to return a DataFrame as the
↪  result, assign it to a variable named
↪  'result'
2. If you modify an existing DataFrame, keep
↪  its original variable name
3. If you create a new DataFrame (other than
↪  the final result), use a clear variable
↪  name (e.g., df_temp)
4. Each input table is already loaded as a
↪  DataFrame. The variable name of each
↪  DataFrame is the same as the input table
↪  name, such as table_1, table_2, etc.

For join operations:
1. Use the appropriate merge/join method based
↪  on the requirement
2. Make sure to specify the correct 'on' or
↪  'left_on'/'right_on' parameters
3. Use the appropriate join type (inner, left,
↪  right, outer) as required
4. After joining, assign the result to the
↪  'result' variable.

Please generate Pandas code that solves this
↪  problem. Only return the code, no
↪  explanation.
Ensure the code is executable and follows the
↪  return guidelines above.
```

B.4 Prompts for Text-to-SQL

To support accurate and consistent SQL generation, we design structured prompt templates as following that pair natural language instructions with table schemas.
```

Prompt for Text-to-SQL

```
I need you to convert natural language questions into SQL queries.

The database schema is as follows:
{schema_prompt}

question: {sample["question"]}

Please generate an SQL query that can answer this question. Only return the SQL query, no explanation is needed. Ensure your SQL query is executable in SQLite.
```

C Visualization Platform

We develop an interactive visualization platform to help human-experts inspect and debug data, featuring a dashboard and task-level panels for description, code, and table comparison.

Dashboard Overview. This overview dash board (Fig. 10) displays metadata for each task, including status, execution time, predicted complexity, and involved operations (e.g., `groupby`, `sort`, `topk`).

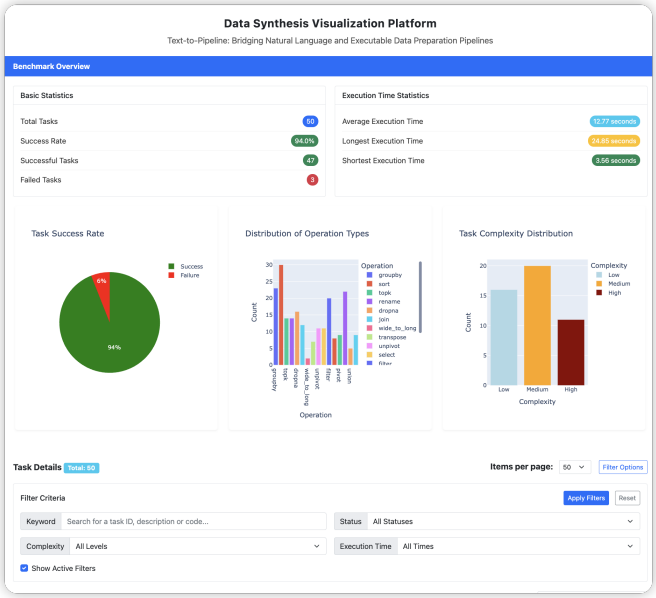

Figure 10: Benchmark Overview. The dashboard summarizes total task count, success rate, execution time statistics, operator distributions, and task complexity.

Task Description Panel. This panel (Fig. 11) shows the original natural language instruction, its interpreted transformation intent, and the corresponding symbolic transformation chain.

Code Implementation Panel. This panel (Fig. 12) presents the compiled Python (Pandas) implementation generated from the symbolic program.

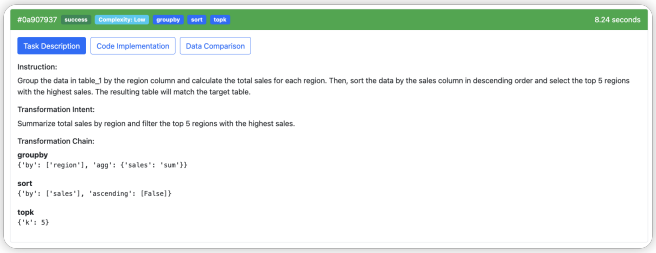

Figure 11: Task Description Panel. It displays the original natural language instruction, the rewritten instruction, and a structured transformation chain in DSL.

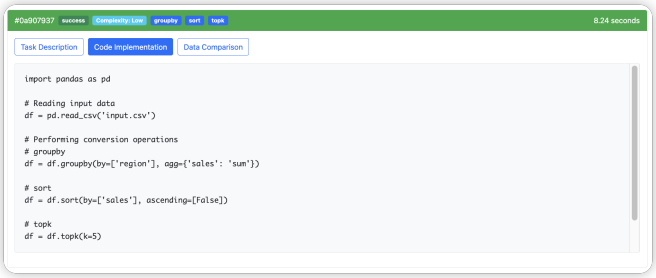

Figure 12: Code Implementation Panel. It presents the synthesized Python (Pandas) code that executes the DSL logic.

Table Comparison Panel. This panel (Fig. 13) provides side-by-side visualization of the input table, ground-truth output, and actual model execution result.

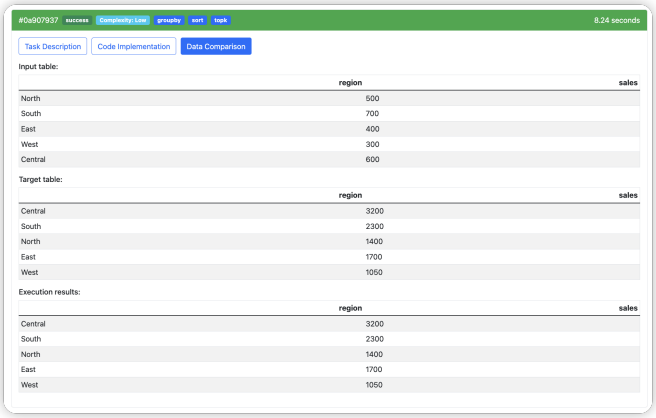

Figure 13: Data Comparison Panel. It visualizes the input table, target output, and actual execution result side-by-side for verification.