Weak-to-Strong Prompts with Lightweight-to-Powerful LLMs for High-Accuracy, Low-Cost, and Explainable Data Transformation

Changlun Li
HKUST(GZ)
Guangzhou, China
cli942@connect.hkust-gz.edu.cn

Chenyu Yang
HKUST(GZ)
Guangzhou, China
cyang662@connect.hkust-gz.edu.cn

Yuyu Luo
HKUST(GZ)/HKUST
Guangzhou, China
yuyuluo@hkust-gz.edu.cn

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

Nan Tang*
HKUST(GZ)/HKUST
Guangzhou, China
nantang@hkust-gz.edu.cn

ABSTRACT

Data transformation poses significant challenges due to the wide diversity in input data formats and different requirements. Existing approaches—including human-driven, algorithmic, and large language model (LLM)-based solutions—each exhibits trade-offs in terms of cost, accuracy, and the range of supported transformations. To address these limitations, we propose **MegaTran**, a novel framework for generating accurate and cost-effective data transformation code. **MegaTran** employs a two-stage process: **Weak2StrongPrompt**, which converts a user's weak prompt (a loosely specified user input) into a strong, structured prompt, and **Prompt2Code**, which generates transformation code based on this refined prompt. In **Weak2StrongPrompt**, a fine-tuned lightweight LLM predicts the transformation type and generates a detailed task description from the user's input. In **Prompt2Code**, a powerful LLM generates the corresponding transformation code, guided by two key optimizations: (1) *Sanity-check Reflection with checklist*, which iteratively debugs and refines the code by addressing errors; and (2) *Lazy-RAG*, a retrieval-augmented generation technique that retrieves relevant code snippets or documentation from external resources (*e.g.,* GitHub, DataPrep) to enhance code quality. Extensive experiments show that **MegaTran** achieves results varying from **+2.2%** to **+26.1%** accuracy improvement compared with SoTA methods.

PVLDB Reference Format:
Changlun Li, Chenyu Yang, Yuyu Luo, Ju Fan, and Nan Tang*.
Weak-to-Strong Prompts with Lightweight-to-Powerful LLMs for High-Accuracy, Low-Cost, and Explainable Data Transformation. PVLDB, 18(8): 2371 - 2384, 2025.
doi:10.14778/3742728.3742734

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/HKUSTDial/megatran.

Nan Tang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742734

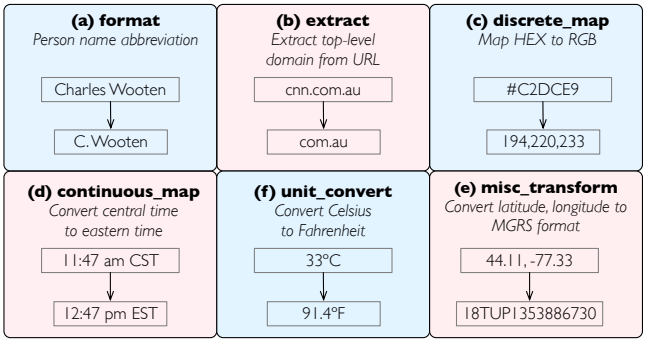

Figure 1: Typical data transformation examples.

1 INTRODUCTION

Data transformation [7, 9, 13, 14, 29, 30, 38, 56, 59, 69], which refers to the process of converting data into a standardized or normalized format in this paper, is essential for ensuring consistency and preparing the data for effective analysis or processing.

Data transformation can take *various types* [31, 38], which includes but is not limited to (see Figure 1): (a) formatting values, such as abbreviating a name; (b) extracting substrings, such as extracting a domain from a URL; (c) discrete value mappings, such as converting HEX values to RGB values; (d) continuous value mappings, such as converting from CST to EST; (e) unit conversions, such as changing temperatures from Celsius to Fahrenheit; and (f) other miscellaneous transformations.

Existing Solutions and Their Limitations. For data transformation tasks, scientists typically know which columns to transform, can provide a few example cases, and have a general sense of the transformation (see Figure 2 Spec). Additionally, they often utilize various tools, such as profiling tools like Trifacta or Tableau, or consult existing code libraries and community resources like DataPrep [53] and GitHub, as shown in Figure 2 Tools . Based on this information, existing solutions are categorized as follows.

Human Experts manually write code (*e.g.,* Python or domain-specific languages) for data transformation. This incurs high human costs ($$$) but it offers high accuracy (+++).

Alg Various algorithms have been proposed based on input-output pairs (referred to as Transform-by-Example [1, 23, 30, 32, 36]) or

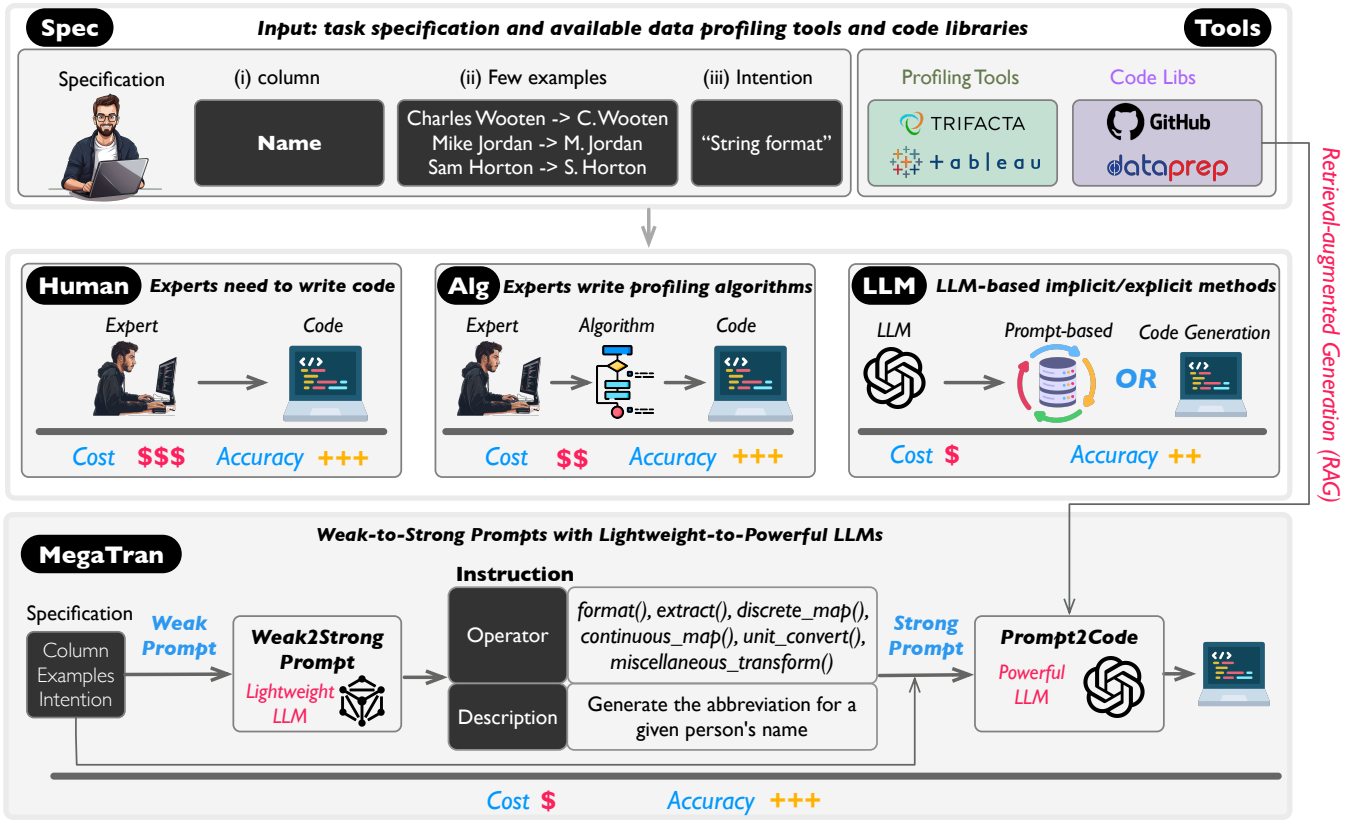

Figure 2: Different data transformation approaches.

unpaired input and output examples (referred to as Transform-by-Pattern [37, 79]). They typically incur medium costs ($$), and offer high accuracy (+++). Note, however, that the experts need to write different algorithms for different types of transformations, and the supported transformation types are limited.

LLM Recently, large language models (LLMs) [5, 74] have demonstrated significant potential across various tasks, such as in-context learning [20, 54, 73, 78] and code generation [3, 8, 25, 45, 51]. LLMs can either be directly prompted to transform data (*i.e.,* implicit transformation) [6, 39, 50, 55, 84] or leveraged for the generative capabilities to produce transformation code (*i.e.,* explicit transformation). However, the former approach lacks explainability, making it difficult to adopt, while the latter tends to be error-prone, with generated code often overfitting and lacking generalizability. These methods typically incur low costs ($) with medium accuracy (++).

Our Goals: High-accuracy, Low-cost, and Explainable. *High-accuracy* means ensuring that the transformed data closely aligns with the ground truth, with minimal deviation from the expected result. *Low-cost* refers to both minimizing human effort and reducing the monetary costs associated with prompting LLMs, making the transformation process more efficient and affordable. *Explainable* means providing transparency into the transformation process, not only identifying which values have been modified (*i.e.,* the "what") but also clarifying the rationale behind each transformation decision (*i.e.,* the "why").

Challenges. Leveraging LLMs for code generation faces two key challenges that must be addressed to achieve high accuracy.

(1) Weak Prompts. The first challenge is the loosely specified user input (*i.e.,* weak prompts), which can lead to LLMs generating incorrect or overly specific code. In contrast, when users provide clear, explicit instructions (*i.e.,* strong prompts), it significantly improves the quality of the generated code by reducing uncertainty in the interpretation of the task.

Example 1. [*Weak Prompts vs. Strong Prompts.*] *Consider the example in Figure 3 from a benchmark dataset, which involves converting an Excel serial date into a human-readable format, with three sample cases shown on the top left.*

[*Weak Prompt to Bad Code.*] *A weak prompt, characterized by a loosely specified instruction, leads to suboptimal results even when using a powerful LLM such as GPT-4o. In this scenario, the generated Python code is inadequate—it overfits to the few provided examples and fails to generalize to all possible cases.*

[*Strong Prompt to Good Code.*] *In contrast, a strong prompt includes a well-specified transformation type and a detailed task description. With this additional context, LLMs can produce high-quality and generalizable code that effectively handles a broader range of inputs.*

Example 1 highlights the important role of strong prompts in enabling LLMs to generate robust and reliable transformation code. However, it is important to acknowledge that providing strong

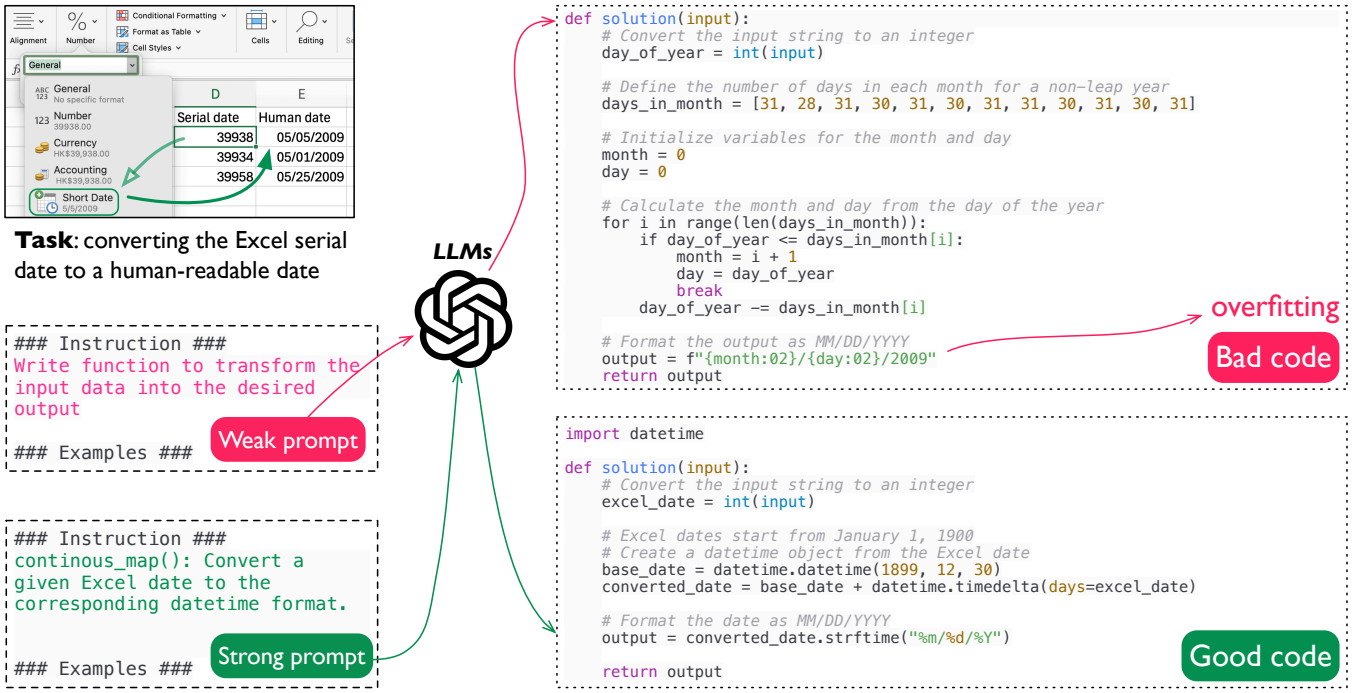

Figure 3: Bad code generation with weak prompts vs. good code generation with strong prompts.

prompts or precise instructions may not always fall within the user's expertise or responsibility.

(2) Data Challenges and Knowledge Gaps. Data transformation tasks are frequently challenged by the inherent complexity of diverse data formats, structures, and contextual semantics, as well as the need for domain-specific knowledge. Even with strong prompts, code generated by powerful LLMs can still suffer from errors or suboptimal performance due to these multifaceted challenges.

MegaTran In response to the above two challenges, we propose a novel two-stage framework, namely **MegaTran**, for the generation of data transformation code, as shown in Figure 2.

Weak2StrongPrompt. The first stage transforms user-provided weak prompts (Challenge 1) into strong prompts by generating precise transformation types and detailed task instructions.

Prompt2Code. The second stage focuses on generating transformation code while addressing data-related challenges (Challenge 2), such as errors arising from data complexity or missing domain knowledge. To ensure robust code generation, we design an error-driven reflection mechanism that enables the LLM to self-debug and iteratively refine its output. To further address domain-specific knowledge gaps, we introduce a novel Lazy-RAG approach. Unlike Eager-RAG methods that always retrieve external examples or documentations [83], Lazy-RAG selectively retrieves relevant information based on error analysis and task context.

Contributions. We summarize our contributions as follows:

(1) We define the problem of data transformation code generation, categorize six common transformation operations, and introduce our **MegaTran** framework in Section 2.

(2) We develop the **Weak2StrongPrompt** module by fine-tuning a lightweight LLM and constructing a task-specific fine-tuning dataset in Section 3.

(3) We propose two optimization techniques for **Prompt2Code**: (i) a *Sanity-check Reflection with checklist* module to enable error-driven self-debugging and iterative refinement in Section 4, and (ii) a *Lazy-RAG* strategy that selectively retrieves external knowledge to address domain-specific knowledge gaps in Section 5.

(4) We perform extensive experiments to demonstrate the effectiveness of **MegaTran**, which significantly outperforms the state-of-the-art solutions (varying from **+2.2%** to **+26.1%** accuracy improvement on five distinct datasets), in Section 6.

2 PRELIMINARY AND SOLUTION OVERVIEW

2.1 Preliminary

Data Transformation. *Data transformation* refers to the process of modifying or converting the values in a specific column of a relational table to a predefined format or standardized representation.

Code Generation for Data Transformation. This refers to the task of automatically generating program code that can perform data transformation on a given dataset [19, 42, 46]. Given a relational table and a transformation task specification (such as converting values in a column to a standardized format or mapping values to a new domain, and a few examples), the goal is to generate code that accurately implements the required transformation.

2.2 Data Transformation Operators

We categorize data transformation tasks into several common types based on their function and characteristics. Note that these data

transformation types are aligned with a work from Microsoft [32]. Please refer to Figure 1 for examples.

(1) `format`: Formatting involves transforming data values into a unified and consistent format, typically without involving arithmetic operations or complex transformations.

The primary goal of formatting is to standardize the appearance or representation of data, such as converting dates into a consistent format (*e.g.,* MM/DD/YYYY), normalizing text case (*e.g.,* converting names to title case), or adjusting numeric formats (*e.g.,* formatting numbers with a fixed number of decimal places).

(2) `extract`: This process extracts specific parts of a value from a larger or more complex value.

This process is often achieved using techniques such as regular expressions, which allow for identifying and isolating substrings based on patterns. For example, extracting the domain name from a URL or pulling out the area code from a phone number are typical extraction tasks.

(3) `discrete_map`: It involves converting values based on a finite, predefined mapping relation, similar to a dictionary lookup.

This process is commonly used to map categorical values to corresponding output values. For example, converting product codes to product names or translating country abbreviations to full country names are typical discrete mapping tasks. The transformation is deterministic, where each input value is mapped to a specific output based on the predefined mapping.

(4) `continuous_map`: This involves converting values through mathematical calculations, typically applied to numeric data. This transformation is used to map one set of continuous values to another based on a defined relationship or formula.

For example, converting timezone is an instance of continuous mapping. Unlike discrete mapping, which is based on fixed lookups, continuous mapping often involves arithmetic operations or mathematical formulas to transform the values.

(5) `unit_convert`: Unit conversion refers to the process of converting a value from one unit of measurement to another.

This transformation is common when dealing with physical quantities that can be expressed in different units, such as temperature, length, weight, or volume. For example, converting temperatures from Celsius to Fahrenheit, or converting distances from kilometers to miles, are typical unit conversion tasks.

(6) `misc_transform`: There are many other types of data transformations that do not fall under the specific categories defined above. These transformations, which may involve more complex or context-specific operations, are grouped together as miscellaneous transformations. This category encompasses any remaining transformation types that do not fit neatly into the predefined categories, such as aggregations, reshaping, or conditional value adjustments. All such transformations are classified as miscellaneous due to their diversity and specialized nature.

2.3 Solution Overview

Recall Figure 2 **MegaTran** , starting with a user specification (*e.g.,* text, JSON) that describes the column to be transformed, a few examples, and a loosely specified transformation intent (*e.g.,* "convert date", "extract domain", or "format phone number"), **MegaTran** employs a two-stage process, **Weak2StrongPrompt** and **Prompt2Code**, to generate high-quality data transformation code.

Weak2StrongPrompt: Enhancing User Prompts. It begins with the user specification, treating it as a weak prompt, and refines it by determining the appropriate *transformation type* (*i.e.,* Operator) and generating a more detailed *task description* (*i.e.,* Description). Such a novel weak-to-strong paradigm enhances prompting for code generation and plays a pivotal role in improving the quality of generated code. By providing a well-defined transformation type and clear instructions, **Weak2StrongPrompt** streamlines subsequent steps, as evidenced by the empirical results in Section 6.

Predicting the transformation type and generating task-specific instructions is inherently simpler than directly generating code. To this end, **MegaTran** employs a *lightweight LLM* for this purpose.

To train the lightweight LLM, we first *construct a fine-tuning dataset* encompassing a diverse set of transformation tasks, each annotated with its corresponding transformation type. It consists of two stages: (i) **Offline Stage:** We employ Supervised Fine-Tuning (SFT) to adapt a base LLM using curated fine-tuning dataset; (ii) **Online Stage:** The fine-tuned LLM is then used to analyze the user specification, predict the appropriate transformation type, and generate a precise, task-specific instruction to guide the subsequent code generation process.

Prompt2Code: Code Generation and Optimization. To produce accurate and reliable transformation code based on strong prompts, we employ a *powerful LLM* (*e.g.,* GPT-4). Powerful LLMs excel at handling complex tasks, such as generating precise code solutions tailored to the desired data transformation. To enhance the quality of the generated code, ensuring it is both syntactically correct and semantically aligned with the underlying data requirements, we introduce two key optimization techniques:

(O1) ***Sanity-check Reflection with Checklist.*** This technique improves code accuracy by enabling the LLM to review and iteratively refine its own outputs. To focus on the reflection process, we use an error-driven checklist that systematically identifies and categorizes issues. Whenever the generated code encounters a runtime error, such as `ValueError` or `SyntaxError` in Python, the reflection mechanism is triggered. Guided by the checklist, the LLM analyzes the errors, revises the code, and re-evaluates the updated solution, ensuring continuous improvement.

(O2) ***Lazy-RAG (Retrieval-Augmented Generation).*** To address gaps in domain-specific knowledge or missing external context, Lazy-RAG selectively retrieves relevant information. For example, if the generated code depends on third-party libraries not explicitly mentioned in the user input, Lazy-RAG dynamically fetches relevant documentation or examples from a knowledge base.

Interleaving Sanity-check Reflection and Lazy-RAG. These two techniques are seamlessly integrated into the code generation process of **Prompt2Code** to ensure accuracy and completeness. As the workflow illustrated in Figure 4, we break it down into 3 steps: (i) The strong prompt will guide LLM to generate the intermediate code. (ii) The intermediate code will be executed to check whether it can pass all test cases. If passed, it will be returned as the final code. If not, the code optimization techniques will be connected. (iii)

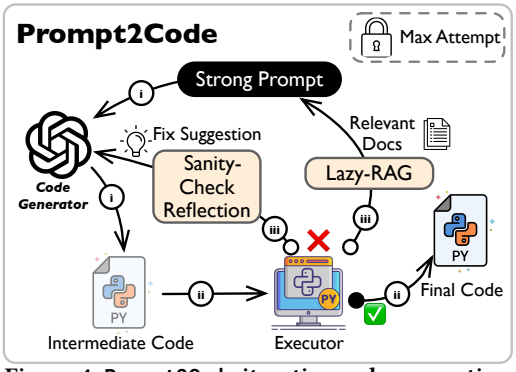

Figure 4: `Prompt2Code` iterative code generation.

(O1): The Sanity-check Reflection module will analyze the error and provide a fix suggestion for the next attempt (Section 4); (O2) The Lazy-RAG module will query the relevant documents and inject them as additional context (Section 5).

Discussion on Generalizability. The framework demonstrates strong generalizability through: (1) Extensible support for transformation operators when it comes to adding new operators for a certain group of transformation tasks; (2) Scalable design where new error types can be incorporated via checklist updates; (3) Additional code primitives can be integrated through vector database expansion; (4) Our empirical experiments across five distinct datasets with hundreds of unseen transformation tasks demonstrate consistent performance improvements without task-specific tuning.

3 WEAK-TO-STRONG PROMPTS

3.1 Weak Prompts and Strong Prompts

Weak Prompt. A *weak prompt*, or a user specification, $\mathbf{S}$, typically provided in natural language format, serves as the primary interface for interacting with our system. It comprises the three components $\mathbf{S} = (C, E, I)$, representing the column specification, examples, and intent specification, respectively.

[C: Column Specification.] Indicates the column containing the values that require transformation.

[E: Examples (*a.k.a.* Demonstrations).] The user is encouraged to provide a few examples to guide the transformation. We classify these examples into two types:

(1) *Instance-level examples*: These are concrete *input-output* pairs that explicitly demonstrate the desired transformation. They help specify the exact mapping or pattern required for the task.
(2) *Abstract-level examples*: These examples provide a broader context, including metadata and semantic information about the source column and the transformed column. They might specify data types, format requirements, or semantic constraints without detailing individual value mappings.

To illustrate these example types, we present three examples corresponding to cases (a), (b), and (c) in Figure 1.

Note that users have the flexibility to provide instance-level examples, abstract-level examples, or a combination of both, depending on the level of detail and context they could provide.

Table 1: Level of granularity on user-provided examples.

Level	Input	Output
Instance	Charles Wooten	C. Wooten
Abstract	a person's full name	the abbreviated name
Instance	cnn.com.au	com.au
Abstract	URL	top-level domain
Instance	#C2DCE9	194,220,233
Abstract	hexadecimal representation	r,g,b values with value range in 0-255

[I: Transformation Intent Specification.] It conveys a general transformation intent, such as normalizing human names or standardizing phone numbers.

Instruction. An *instruction* $\mathbf{I}$, inferred from the lightweight LLM, acts as a crucial intermediary, enhancing the user-provided chat input. It is composed of two key elements, $\mathbf{I} = (O, D)$:

[O: Transformation Operator.] An *operator* identifies a specific type of transformation, ensuring precise alignment between the user's intent and the subsequent code generation process.

[D: Description.] The description provides detailed coding directives to guide the strong LLM effectively.

Experimental findings (see Exp-3: Ablation Study in Section 6) highlight that directly using weak prompts results in suboptimal performance, whereas incorporating structured, instruction-guided strong prompts significantly improves accuracy and reliability.

Strong Prompt. A *strong prompt* inherits weak prompt with few examples and generated instruction (operator and task description). More details of **`Weak2StrongPrompt`** are shown in Figure 5.

Example 2. [*Generated Instructions.*] *Consider the six examples in Figure 1, the corresponding instruction outputs are as follows: (a)* `format`*: generate the abbreviation for a person's name. (b)* `extract`*: extract the top-level domain from the URL. (c)* `discrete_map`*: convert a hexadecimal color code to RGB representation. (d)* `continuous_map`*: convert central time to eastern time. (e)* `unit_convert`*: convert Celsius to Fahrenheit. (f)* `misc_transform`*: convert a given latitude and longitude into MGRS format.*

Discussion on the Choice of Operators. To address the diverse range of data transformation tasks, we define a set of functional operators that encompass the most common transformation scenarios, as detailed in Section 2.2. Among these, the `unit_convert` operator is designed to handle a broad category of tasks involving conversions between different data types, such as converting Celsius to Fahrenheit. This design choice is supported by findings from He et al. [32], who analyzed a wide variety of transformation tasks and identified the pattern "convert A to B" as a frequently queried use case in the Bing search engine.

3.2 Offline Fine-Tuning

We fine-tuned a lightweight LLM to acquire the specific knowledge in data transformation by curated samples [44]. Such a weak model performs a predictive task to translate user intent into accurate instructions for code generation. In practice, the expert user can keep track of the newly occurred transformation, and then update the predictive model from time to time.

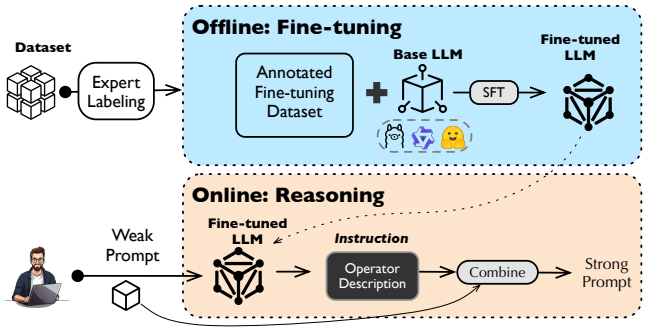

Figure 5: The `Weak2StrongPrompt` module.

Supervised Fine-Tuning. Supervised fine-tuning (SFT) has been widely used in the field of LLM fine-tuning [16, 48]. It is particularly well-suited for our framework as it allows us to train the LLM on paired examples of user inputs and desired instructions. Through SFT, we create a specialized model that can effectively bridge the gap between natural language descriptions and structured transformation instructions. The training process involves presenting the model with ($\mathbf{S}$: *specification*, $\mathbf{I}$: *Instruction*) pair, which contains the specification $\mathbf{S}$ from user side and the instruction $\mathbf{I}$ that represents the desired structured output (O : *Operator*, D : *Description*). This supervised approach ensures that the model learns to consistently map user inputs to well-formed instructions that can guide the subsequent code generation process.

Fine-Tuning Dataset Construction. We constructed a dataset based on the TDE Benchmark [32], which contains over 200 transformation tasks curated from popular and representative questions. To adapt this dataset for fine-tuning, we refactored the original samples in plain-text format into Alpaca-like pairs [70], mapping the user specifications ($\mathbf{S}$) to corresponding instructions ($\mathbf{I}$). Upon reviewing the original dataset, we observed that approximately 30% of the tasks lacked sufficient information, such as vague descriptions, external links without context, or missing textual details. To address this, we modified the user specifications by supplementing or refining them based on the provided examples to ensure clarity and completeness. Since the original dataset did not include detailed instructions, we manually annotated each task with its corresponding operator (O) and description (D), leveraging domain expertise to ensure accuracy.

We define information completeness through a systematic approach. For each data transformation task, we build a fine-tuning sample following the **completeness criteria** below:

- **Chat**: The task briefing of the transformation. We harness the task description provided in the TDE benchmark. If not provided, we will create one based on the examples.
- **Examples (*a.k.a.* tuples)**: Typical input-output examples of the transformation. All examples are collected from the TDE benchmark.
- **Context**: Context information related to the input and output of the transformation *e.g.*,, data type, format, etc.
- **Instruction**: It manifests the instruction $\mathbf{I}$ as the expected output of the fine-tuned LLM, which consists of operator (O) and description (D) for current task.

Algorithm 1 Sanity-check Reflection with Checklist

Input: a prompt and a checklist
Output: generated code **code***, execution log **log**

```
code* ← ∅; log ← ∅; sugg = ∅; cnt = 0; k = 3
do
    code ← LLM(prompt, sugg);
    error ← Executor(code)
    if error == ∅ then
        code* ← code
        break
    else if error ∈ checklist then
        sugg ← LLM_Reflection(code, error)
    else
        log ← Log(error)
        break                          ▷ will send to Lazy-RAG module
while (cnt++ < k)
return code*, log
```

This systematic approach ensures consistent annotation quality while maintaining task authenticity, which took 18 hours in total.

3.3 Online Reasoning

During online reasoning, the user can simply feed a specification $\mathbf{S}$ to the fine-tuned lightweight LLM, which will infer a suitable instruction $\mathbf{I}$ [71, 82, 85]. Then, a strong prompt will be constructed by integrating the weak prompt and the generated instruction. Consequently, the refined strong prompt will be provided to the following **`Prompt2Code`** module for code generation.

4 SANITY-CHECK REFLECTION WITH CHECKLIST

Sanity-check. In programming analysis, a sanity-check is a basic verification step that ensures a program or system behaves according to fundamental expectations [22, 68]. It typically involves quick, simple tests to catch obvious problems before proceeding with more complex operations [80]. For example, checking if input values are within reasonable ranges or if essential resources are available. To mimic this programming practice in an LLM-based manner with less human overhead, we cooperate with the reflection mechanism to improve the quality of generated code.

Reflection for Self-Correction. The reflection mechanism has proven to be effective in enhancing the performance of LLMs, particularly in the domain of code generation [10, 58, 62]. During runtime execution, it allows the model to review and revise its own outputs and errors autonomously. This self-correcting capability is crucial for automated code generation and reducing the need for human intervention. By iteratively analyzing the generated code against expected outcomes and known error patterns, the reflection mechanism can refine the code, ensuring it meets the desired specifications.

Inspired by this programming practice and the self-reflection capabilities of LLMs, we design a Sanity-check Reflection module to guarantee the correctness of generated code. Similar to how developers use sanity-check to handle basic errors early, our module

performs error-driven verification of the generated code with the help of a pre-defined checklist.

Checklist. Inspired by the technique from ACL 2020 best paper [60], we design a checklist to handle common programming errors, such as syntax/value/import errors, for data transformation. To clarify, we harness the conceptual idea of which conducts behavioral testing of NLP models with Checklist, but extend the technique by introducing an error checklist and novel reflection mechanisms for the correctness of the code generation.

Checklist Extension. Extending checklist to other programming languages involves a few simple steps: (i) **Error Identification**: Identify common runtime and syntax errors specific to the target programming language. This involves understanding the typical error messages and scenarios encountered in that language; (ii) **Checklist Adaptation**: Modify the existing checklist to include these language-specific errors. This can be done by collaborating with language experts or using existing documentation and resources; (iii) **Testing and Iteration**: Conduct thorough testing with sample code snippets in the new language to ensure that the adapted checklist and reflection mechanism work as intended. Iteratively refine the approach based on feedback and observed performance.

Overall, while there is some initial effort required to adapt the approach to a new programming language, the modular nature of our system makes this process straightforward. By focusing on the specific error patterns and leveraging the flexibility of the reflection mechanism, we can extend our approach to support a wide range of programming languages.

Algorithm. The algorithm for Sanity-check Reflection with checklist is shown in Algorithm 1. It takes a prompt and a checklist as input, and outputs the data transformation code and error logs. It first initializes all variables (line 1). It then iteratively checks and fixes the code until the maximum number of attempts is reached (lines 2-13). During each iteration, it first generates code with LLM (line 3) and captures error message by executing the code (line 4). If the code can be successfully executed (line 5), the code will be returned and the iteration will be terminated (lines 6-7). Otherwise, if there is an execution error captured by the checklist (line 8), it will use LLM to generate a fix suggestion (line 9), which will be used for the next round of iteration. However, if the execution error cannot be captured by the checklist (line 10), then LLM reflection cannot solve the problem. It will log the error message (line 11) and terminate the iteration (line 12). Finally, it will return the code and error logs (line 14).

Note that, there are three types of outputs of the algorithm:

(1) $\texttt{code}^* \neq \emptyset$: good code is generated
(2) $\texttt{code}^* == \emptyset$ and $\texttt{log} \neq \emptyset$: invoke the Lazy-RAG module
(3) $\texttt{code}^* == \emptyset$ and $\texttt{log} == \emptyset$: cannot generate good code

Example 3. *Let's illustrate with an example, as shown in Figure 6.*

(i) [Code Generation.] The powerful LLM will generate an "Intermediate code" given the "Strong prompt" for converting shoe sizes on the operator `domain_map`.

(ii) [Code Execution.] A code executor will execute the "Intermediate code". It will terminate if the code can pass all test cases (i.e., a final code). Otherwise, it will send an error to the checklist. In this case, it sends an error message.

(iii) [Reflection with Checklist.] If the error is registered in the checklist, it will send the error to the Reflector *to generate concrete fix suggestion. Otherwise, it will send the log to the Lazy-RAG module. In this case, the error* `SyntaxError` *is captured by the checklist and the reflector will provide a "Fix suggestion".*

(iv) [Code Refinement.] The code generator will incorporate the "Fix suggestion" and re-generate the code, and go back to step (i). In this case, there is no error in the Executor and a good "Final code" is generated.

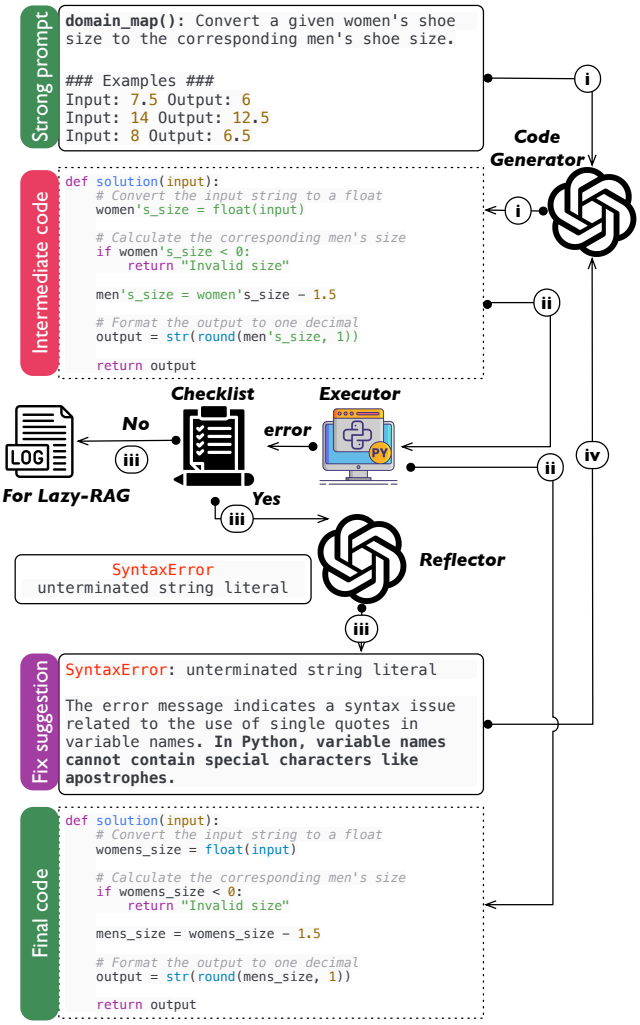

Figure 6: Sample workflow of Sanity-check Reflection.

5 LAZY-RAG

Retrieval-Augmented Generation (RAG) is a technique that enhances language model outputs by incorporating relevant information from external knowledge sources during generation [2, 41]. In the context of code generation, RAG becomes particularly crucial as it allows the model to access up-to-date documentation, best

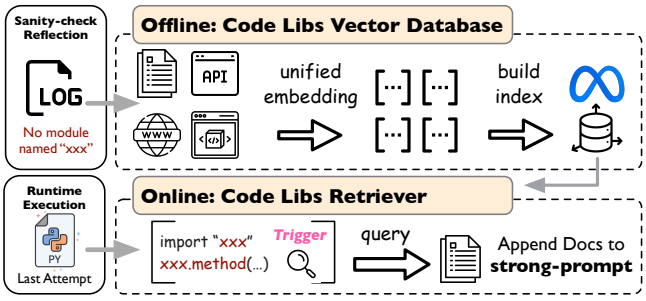

Figure 7: Lazy-RAG workflow.

practices, and proven code patterns that may not be present in its training data [66, 77].

The necessity of RAG for code generation stems from several challenges. First, programming languages, libraries, and APIs constantly evolve, making it difficult for pre-trained models to stay current. Second, many transformation tasks require specific domain knowledge or library-specific implementations that may be sparsely represented in the model's training data. By retrieving relevant documentation and code examples during generation, RAG helps maintain high accuracy and ensures the generated code follows current best practices and correct function usage.

Therefore, we utilize RAG to enhance our code generation process via two components: **code libs vector database** and **code libs retriever** (see Figure 7). This design balances efficiency with effectiveness, allowing us to maintain a comprehensive knowledge base while minimizing runtime overhead.

Eager-RAG vs Lazy-RAG. Our preliminary experiments reveal a critical insight about RAG usage in code generation. While traditional Eager-RAG techniques retrieve relevant code or documentation at every generation step, this approach can be counterproductive. We observe that constant retrieval often introduces noise and irrelevant information into the generation process, potentially degrading code quality rather than improving it.

For instance, when performing "datetime" related transformation, additional context from external sources may confuse the model as similar API names exist in different libraries. A concrete example is timezone conversion, where multiple libraries `datetime`, `pytz`, `dateutil`, and `pandas` all provide similar-sounding methods like `astimezone()`, `convert_tz()`, and `tz_convert()`. Eager-RAG might retrieve all these variations simultaneously, leading the model to mix APIs or suggest incompatible combinations. Our experiments (Refer to Exp-4, Table 5) show that Eager-RAG can lead to redundant generation attempts for simple tasks, increased inference time, and token consumption without accuracy improvements.

These observations motivate our Lazy-RAG approach, where retrieval is triggered only when previously logged packages are captured. This selective strategy helps maintain the balance between leveraging external knowledge and preserving the model's inherent capabilities.

Code Libs Vector Database. The code libs vector database serves as a knowledge repository that stores external package information and documentation, playing a crucial role in addressing Challenge 2 (Data Challenges and Knowledge Gaps). When the Sanity-check Reflection module encounters missing package errors or function call issues during code execution, it logs these incidents along with contextual information. Based on these logs, we manually curate relevant resources, including PyPI packages (*i.e.,* web pages), markdown files, and reStructuredText files from GitHub repositories, and best practices from developer communities, such as StackOverflow. Due to contributions from the open-source community, these code fragments are executable with detailed comments. Finally, we prepared a target list of packages with a simple script to help download locally and process the construction of the offline code lib vector database.

The vector database operates in offline mode, meaning it can be pre-built and optimized before the code generation process begins. It takes around 15 hours to complete the above works, including the script running time on downloading files and building the vector database. We employ a unified embedding approach where both package documentation and code snippets are embedded into the same vector space using a consistent encoder. This unified representation enables efficient similarity-based retrieval during the online generation phase. In terms of the embedding cost, we harness a smaller and highly efficient OpenAI model `text-embedding-3-small` to embed the chunks at a price of $0.00002 per 1K tokens.

Code Libs Expansion. Notably, the target list allows effective knowledge expansion, *i.e.,* adding a new package then using our automated ingestion scripts (refer to *assets/rag/pkg_info.json* and *scripts/build_vector_db.py* in artifact materials respectively). Consequently, the end user can refresh the code libs vector database rapidly.

Code Libs Retriever. This module operates during the online phase, lining up with the code libs vector database to inject external knowledge into the **`Prompt2Code`** process.

Algorithm. We present the workflow of the code libs retriever in Algorithm 2. It takes a prompt, a related code, and a list of missing logs as input, and outputs a new prompt. It first initializes a new prompt with the input prompt (line 1). Then, a trigger function checks the content of the input code if imported packages exist in pre-collected list (line 2). If found, it embeds the context as a query vector and retrieves the top-n relevant documents from the vector database (lines 3-4). Next, it combines the retrieved information with the original prompt (line 5). Finally, it returns the new prompt (line 6).

Remark. By design, the trigger function makes this retrieval process "lazy"; it only starts when certain packages or APIs are referred to, as opposed to trying to retrieve data at every attempt. The retrieved information is then used to augment the `strong-prompt` for the next attempt, providing the model with precise, relevant documentation and knowledge. This approach helps maintain a balance between leveraging external knowledge and avoiding information overload that could confuse the model.

6 EXPERIMENTS

The key questions we ask with our evaluation are:

- How is the overall performance of our approach in terms of accuracy and cost efficiency, compared to other baselines? (Exp-1, Exp-2)
- How is the ablation study? (Exp-3)

Algorithm 2 Code Libraries Retriever

Input: a prompt, a code, list of missing packages mlist
Output: new prompt **prompt***

```
prompt* ← prompt; top-n = 3
if Trigger(code, mlist) then
    query ← LLM_embedding(code)
    docs ← VecDB_retrieve(query, top-n)
    prompt* ← combine(prompt, docs)
return prompt*
```

- What is the cost efficiency of Lazy-RAG compared with Eager-RAG? (Exp-4)
- How is the overall performance in terms of the number of coding attempts? (Exp-5)

6.1 Experiment Setup

Benchmark Datasets. Initially, our experiments utilize two challenging datasets from the Transform-Data-by-Example (TDE) benchmark[1]: StackOverflow and Bing-QueryLogs. Besides, we carefully curated three new datasets from real-world applications, which are ETL, CommonSense, and Science, which serve as the testing set for our proposed approach. Below is a detailed description of each dataset with the corresponding notation.

(1) StackOverflow (SO): it consists of 49 carefully sampled questions from "StackOverflow.com" that specifically focus on data transformation tasks (*e.g.,* "how to get the domain name from URL").
(2) Bing-QueryLogs (BQ): it is constructed from 50 frequently occurring "convert A to B" pattern queries in Bing's search logs. Non-data transformation queries such as "convert jpg to png" are filtered out.
(3) ETL: it is refactored from the DTT paper [12], which contains 46 data transformation tasks related to ETL processes.
(4) CommonSense (COM): it is constructed based on the common data transformation patterns, with a total of 43 data transformation tasks.
(5) Science (SCI): it is constructed based on scientific topics, such as physics and chemistry, which contains 23 data transformation tasks.

The above five datasets contain more than 200 transformation tasks that cover diverse domains and representative challenges. We follow the same data processing pipeline as baselines [32, 50, 55]. For the example tests in each task, we split them into two groups: the first 3 tests are used as the training set, which will be combined as part of the prompt input. The remaining tests will become test cases during the iterative code generation process. Such an evaluation setting employs the unseen samples for the final code solution, which avoids the overfitting issue. Meanwhile, we ensure that all methods are tested on the same test cases.

Evaluation Metrics. Following the same evaluation metric, we measure accuracy as the proportion of successfully completed transformation tasks. For LLM-based methods, we additionally track the

[1]https://github.com/Yeye-He/Transform-Data-by-Example/tree/master/Benchmark

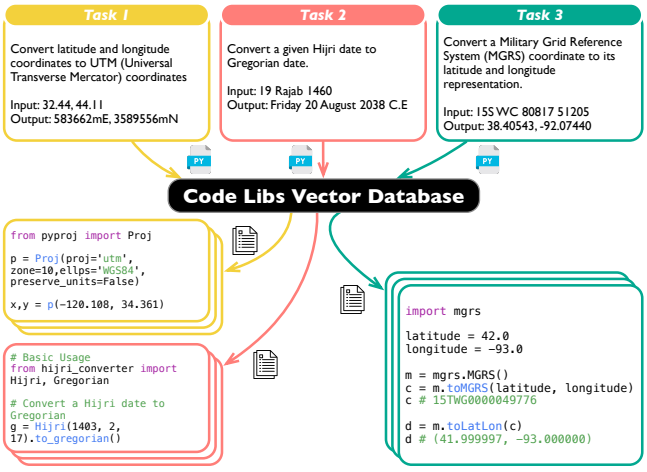

Figure 8: Real-world examples from testing benchmarks. Documents relevant to the code are retrieved.

inference cost for each transformation task, including execution time, token consumption, and the frequency of calling LLM API.

Besides, we utilize the Pass@K metric [8], which evaluates the ability of data transformation solutions to produce at least one correct code solution for current task within K attempts, where K can take values of 1, 3, or 5 (see Exp-5, Figure 9). This metric serves as an additional measure for assessing the performance of LLMs in example-based code generation. Note, we report accuracy at Pass@3 for main experiments, where $K = 3$.

Baselines. We compare our approach against the below baselines:

(1) TDE [32]: it represents a search-based approach for data transformation from the original benchmark.
(2) Foundation Model (FM) [50]: it prompts naive LLM with in-context learning to generate the transformation output directly.
(3) DTT [12]: it proposes a pre-trained language model trained on 2,000 synthetic training groups of transformation. The model can directly generate the transformed results based on provided examples.
(4) UniDM [55]: it is a unified framework for LLM-based data manipulation, which represents the current LLM-based SoTA method.
(5) Code LLM [3]: this code generation approach with LLM can be treated as the "backbone" of our proposed approach. The prompt guidance is simple and only contains general instructions (*e.g.,* "Write a function to transform the input data into the desired output") and examples.

Implementation Details. The **MegaTran** is built entirely in Python, with all experiments conducted on a computing server equipped with 8 RTX 4090 GPUs. For all LLM-related reasoning jobs, we set the model temperature as 0.2, which makes the output more deterministic and focused on the most probable tokens.

Weak2StrongPrompt. For the offline training, we employ the supervised fine-tuning technique to fine-tune a weak LLM. In our

experiments, the base model is `Llama3-8B-Instruct` [18]. We utilize the LoRA technique [33] to speed up the training efficiency. With the support of LlaMa-Factory [87], we build an easy and efficient fine-tuning pipeline and it takes less than 30 minutes on our curated fine-tuning dataset with 8 GPUs. The online inference is supported by vLLM library [40] with a single GPU to ingest the lightweight fine-tuned model.

`Prompt2Code`. For code generation, we employ the OpenAI's `GPT-4o` and `GPT-4o-mini` [34] as our strong LLM. After parsing the relevant code block from the model completion, we utilize a temporary Python file that stores the generated code, then calls a sub-process to execute it in auto-mode via `importlib`. The Sanity-check Reflection is implemented during the code execution. The relevant running result and errors will be captured based on the pre-defined checklist. In terms of the Lazy-RAG module, we construct the vector database offline by leveraging Meta FAISS [17] for efficient vector similarity search. Based on the missing package logs, we manually collect relevant online documents, such as web pages, Github repositories, or the library homepage, to build the code libs vector database. During the online retrieval, we utilize the OpenAI's embedding model (`text-embedding-3-small`[2]) to create embedding based on the code snippet. Then, we perform a similarity search on the vector database to return top-3 documents as the retrieval results. Overall, we adapt LangChain library for Lazy-RAG.

LLM API cost. We report the module-level cost in terms of API calls and token consumption in Exp-2, Table 3. The main expense comes from three parts: (1) *Embedding Generation.* The online and offline stages in Lazy-RAG both require vector embedding for the knowledge retrieval in code libs vector database and the construction of the database itself via unified embedding, respectively; (2) *Fix Suggestion.* The Sanity-check Reflection module is to provide the fix suggestion for the generated code. It is implemented by calling the LLM API; (3) *Code Generation.* The cost of code generation is dominated by the LLM API calls. To control the monetary cost, we limit the process by setting a maximum number, *i.e., K* attempts.

6.2 Results and Analysis

Exp-1: End2End Comparison. Table 2 shows the overall result of different approaches. We evaluate the accuracy of each approach on five benchmark datasets. As shown in this table, our approach achieves the highest accuracy on all datasets, outperforming the LLM-based SoTA method UniDM and the search-based SoTA method TDE. Specifically for experiments backend by `GPT-4o`, our approach obtains 77.6%, 78.0%, 67.4%, 74.4% and 65.2% accuracy on StackOverflow, Bing-QueryLogs, ETL,CommonSense and Science, with huge improvements of 12.0%, 14.0%, 2.2%, 9.3% and 26.1% respectively relative to the current SoTA method.

Note, however, both UniDM and FM approaches are using GPT-3-175B parameter model (`text-davinci-002`). This particular model is not publicly available at the time we conduct the experiments. Also, the TDE and UniDM projects are not open-sourced. As the FM provides its open-source codes, we reproduce its implementation using OpenAI latest models to conduct a fair comparison.

[2]https://platform.openai.com/docs/guides/embeddings/

Table 2: Accuracy (%) comparison. * represents the result from the original paper. We underline the current SoTA method.

Method	SO	BQ	ETL	COM	SCI
#-Tasks	49	50	46	43	23
TDE [32]	63.0*	32.0*	-	-	-
FM [50]	65.3*	54.0*	-	-	-
UniDM [55]	67.4*	56.0*	-	-	-
DTT [12]	4.1	1.1	26.1	9.3	4.4
By GPT-4o					
FM	75.6	64.0	56.5	65.1	30.4
Code LLM	67.3	56.0	65.2	55.8	39.1
`MegaTran`	**77.6**	**78.0**	**67.4**	**74.4**	**65.2**
By GPT-4o-mini					
FM	67.3	54.0	56.5	53.5	21.7
Code LLM	61.2	56.0	58.7	44.2	34.8
`MegaTran`	**75.5**	**74.0**	**60.9**	**69.8**	**52.2**

Moreover, we reproduced the DTT implementation [12] using its original pre-trained model and adopted it as a baseline. We feed the model three examples along with the data to be transformed following our setting. However, DTT performs poorly compared to our approach, likely due to two factors: (1) its synthesized data relies on only five basic transformation units (*e.g.,* substring, split), significantly limiting its applicability to broader scenarios; (2) DTT is designed for joinability, where the transformed output is then matched with the most similar candidate [27]. Thus, transformation accuracy is not the primary objective of it.

We observed that the overall accuracy of FM is even worse after upgrading to a more powerful LLM model. This may be due to the generation preference of the GPT-4o family, which is more likely to chat with users, instead of generating a simple output. Consequently, it makes trouble for us to evaluate the performance of the FM approach. We also observed that the naive code LLM approach performs poorly on the data transformation task. Recall the example in Figure 3, the weak prompt-guided LLM is yet unable to generate a correct code solution.

> Finding 1: **`MegaTran`** significantly outperforms existing SoTA methods on all benchmark datasets. The performance gain is consistent across different LLM backends (GPT-4o and GPT-4o-mini) and testing on different datasets.

Exp-2: Cost Efficiency. This experiment is to explore the contribution of each module in **`MegaTran`**. Recalled that the **`Prompt2Code`** relies on Sanity-check Reflection and Lazy-RAG modules, thus we monitor the API-based cost for fix suggestion and query embedding respectively. Specifically, we report the inference cost including the execution time, token consumption per task (prompt tokens, completion tokens), and the number of API calls. Due to Code LLM relying on the code generation module only, we report its cost followed by the code generation phase of **`MegaTran`**.

As shown in Table 3, our approach requires fewer API calls for code generation compared to Code LLM on all datasets, except for Science, while still achieving superior accuracy. Also, our approach is more efficient in terms of time cost when evaluating on

Table 3: API-based LLM usage for each inference phase. GPT-4o is used as the strong LLM. Be noted that the query embedding does not have any cost on completion tokens.

Inference phase Measure Unit	API #-Calls	Prompt #-Tokens	Completion #-Tokens
SO: 5m32s			
Code LLM (6m20s)	98	184	106
Code Generation	68	250	138
Weak2StrongPrompt	49	77	13
Fix Suggestion	19	177	234
Query Embedding	-	-	-
BQ: 8m51s			
Code LLM (13m52s)	94	253	215
Code Generation	76	382	190
Weak2StrongPrompt	50	76	15
Fix Suggestion	26	256	266
Query Embedding	7	185	0
ETL: 8m23s			
Code LLM (8m54s)	89	247	172
Code Generation	73	259	60
Weak2StrongPrompt	46	100	14
Fix Suggestion	27	154	178
Query Embedding	-	-	-
COM: 11m19s			
Code LLM (6m22s)	80	222	149
Code Generation	66	384	223
Weak2StrongPrompt	43	92	15
Fix Suggestion	23	307	306
Query Embedding	3	158	0
SCI: 7m46s			
Code LLM (6m26s)	50	254	153
Code Generation	52	416	171
Weak2StrongPrompt	23	100	15
Fix Suggestion	29	261	266
Query Embedding	-	-	-

StackOverflow, Bing-QueryLogs, ETL. This demonstrates the efficiency of our framework while maintaining high performance. The utilization patterns across all modules indicate their essential contributions to the overall system performance.

Note that the Lazy-RAG online stage is only triggered when the generated code contains statements and references that were previously logged from the Sanity-check Reflection module. In StackOverflow, CommonSense and Science, the Lazy-RAG module remained unused (zero cost of query embedding) since none of the generated codes required external knowledge.

Finding 2: **MegaTran** achieves a competitive inference cost compared to the SoTA methods. Our framework demonstrates efficient resource utilization through its modular design. The varying costs between datasets reflect the adaptive nature of our system to different task complexities.

Exp-3: Ablation study. This experiment is to justify the effectiveness of each module in **MegaTran**. We compare the accuracy decrease by removing each module out of the whole system. As

Table 4: Ablation study of our approach on each module, evaluated on accuracy (%).

	SO	BQ	ETL	COM	SCI
GPT-4o	77.6	78.0	67.4	74.4	65.2
- w/o Weak2StrongPrompt	-8.2	-14.0	-2.2	-13.9	-13.0
- w/o Sanity-check Reflection	-2.1	-4.0	0	-9.3	-30.4
- w/o Lazy-RAG	-	-2.0	-	-2.3	-
GPT-4o-mini	75.5	74.0	60.9	69.8	52.2
- w/o Weak2StrongPrompt	-12.2	-20.0	-6.6	-18.6	-17.4
- w/o Sanity-check Reflection	-4.1	-8.0	-3.1	-16.3	-21.8
- w/o Lazy-RAG	-	-4.0	-	-2.4	-

Table 5: Explore the cost efficiency of Lazy-RAG vs Eager-RAG. GPT-4o-mini is used as the strong LLM.

	SO	BQ	ETL	COM	SCI
Lazy-RAG					
Acc. (%)	75.5	74.0	60.9	69.8	52.2
#-Queries	-	6	-	3	-
#-Tokens	-	199	-	216	-
Time cost	4m13s	6m24s	5m9s	7m52s	4m33s
Eager-RAG					
Acc. (%)	75.5	72.0	60.9	67.4	47.8
#-Queries	25	30	35	31	29
#-Tokens	147	157	64	366	155
Time cost	4m57s	7m32s	8m31s	8m21s	6m1s

shown in Table 4, all components in our framework contribute positively to the overall performance. The **Weak2StrongPrompt** module proves to be the most critical component, where its removal leads to significant performance drops across all datasets. Specifically, the accuracy decreases by 14.0% and 13.9% on Bing-QueryLogs and CommonSense respectively when using GPT-4o. The impact is even more pronounced with GPT-4o-mini, showing drops of 20.0% and 18.6% respectively. This substantial degradation demonstrates that the quality of the strong prompt is crucial for code generation.

The Sanity-check Reflection module also presents meaningful contribution, with its removal causing accuracy drops varying from 9.3% to 16.3% on CommonSense and surprisingly from 21.8% to 30.4% on Science across backend LLMs. The Lazy-RAG component, while not activated for all datasets, contributes to a 2-4% accuracy improvement on Bing-QueryLogs and CommonSense, suggesting its utility for more complex transformation tasks.

Finding 3: **Weak2StrongPrompt** module is the most critical component of our framework, with its removal causing the largest performance degradation (up to 20% drop in accuracy). Both the Sanity-check Reflection and Lazy-RAG optimizations contribute meaningfully to the overall system performance.

Exp-4: Lazy-RAG vs Eager-RAG. This experiment discussed the cost efficiency of Lazy-RAG vs Eager-RAG. We report the accuracy, number of queries, average token cost per task, and time cost for both RAG strategies. Normally, the Lazy-RAG retrieval will take place only if triggered by code content, while Eager-RAG requires processing all retrieved documents for every query. As presented in Table 5, it makes our Lazy-RAG particularly cost-efficient compared to Eager-RAG. The latter one results in substantial but redundant

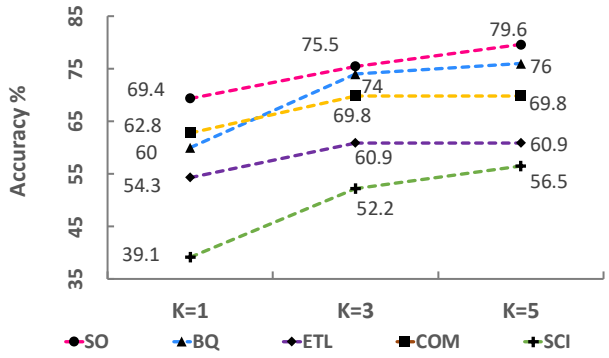

Figure 9: Pass@K ($K = 1, 3, 5$) accuracy, which measures our framework to generate at least one correct code within K attempt. GPT-4o-mini is used as the powerful LLM.

query, *i.e.,* approximately 5x higher on Bing-QueryLogs and CommonSense. The results also reveal GPT-4o-mini's strong capability in handling potentially noisy RAG results. Despite receiving irrelevant documents, GPT-4o-mini can effectively filter them out, maintaining fairly good performance compared to Lazy-RAG.

> Finding 4: Our Lazy-RAG strategy achieves favorable cost-efficiency by triggering context-aware retrieval behaviour, resulting in 5x lower query frequency compared to Eager-RAG.

Exp-5: Pass@K. This experiment is to explore the performance of our framework when scaling up the number of attempts. We evaluate our framework's ability to generate correct code within multiple attempts using Pass@K metric, where $K = 1, 3, 5$.

As shown in Figure 9, the framework demonstrates strong performance improvement as K increases. In StackOverflow, the task accuracy increases from 69.4% ($K = 1$) to 75. 5% ($K = 3$) and reaches 79. 6% ($K = 5$), showing a consistent upward trend. The substantial jump in accuracy between $K = 1$ and $K = 3$, particularly for 14% and 13.1% improvement for Bing-QueryLogs and Science respectively, suggests that allowing multiple generation attempts effectively mitigates the inherent randomness in LLM output.

We also notice that the overall accuracy remains unchanged from $K = 3$ to $K = 5$ on ETL and CommonSense, which indicates that these two datasets are relatively harder to handle.

> Finding 5: In **`Prompt2Code`**, increasing the number of generation attempts significantly improves code transformation accuracy.

7 RELATED WORK

Human-based. Commercial tools like Trifacta [72] and Tableau [67] provide domain-specific language (DSL) for data transformation tasks. There are also open-sourced tools [49] integrated with data transformation, such as Potter's wheel [57], Wrangler [38], Falx [75], and many others [11, 26, 52, 81, 86]. Moreover, domain-specific packages and platforms, such as Tidyverse [28] and GitHub, offer plug-in code solutions. While these approaches achieve high accuracy, they incur significant learning curves *w.r.t.* expertise requirements.

Algorithm-based. Several algorithmic for automated data transformation [76] are through programming-by-example (PBE) [21, 24, 43, 64, 65]. FlashFill [30] generates transformation rules from spreadsheet examples. Following this paradigm, Foofar [36], DataXformer [1], Trinity [47] and Transform-Data-by-Examples (TDE) approaches [32] generate transformation rules or scripts based on user-provided input-output pairs. Particularly, enumeration-based PBE frameworks [4, 15] can be embedded with high-level skills and information. However, they often suffer from scalability issues due to the exponential growth of the search space. More advanced approaches have been explored, such as Auto-Transform [37] and Explain-Data-by-Visual [63]. While these algorithmic solutions offer medium cost and high accuracy, they suffer from limited transformation type support and require separate algorithms for different transformation scenarios.

LLM-based. Recent work has explored using Large Language Models (LLMs) for data transformation in two main ways. Implicit transformation approaches [39, 50, 55, 84, 88] directly utilize foundation models to transform data values. Although such a method is straightforward, it lacks explainability and makes it difficult for users to verify or modify the transformation process. More explainable approaches leverage LLMs to generate a particular script [8, 35] or DSL [61] to perform the data transformation. However, Code LLMs often struggle to generate non-executable or non-scalable code (*e.g.,* over-fitting to specific examples), which limits the code's applicability in real-world scenarios. These limitations can lead to increased manual intervention and debugging efforts, ultimately reducing the efficiency and increasing the monetary cost.

8 CONCLUSION AND FUTURE WORK

We have presented **`MegaTran`**, a novel two-stage framework that addresses the critical challenges of data transformation through cost-effective and accurate code generation. Our system combines a fine-tuned lightweight LLM for converting weak prompts into strong, structured instructions, with a powerful LLM for generating high-quality transformation code. We conduct extensive experiments to validate the effectiveness of our approach.

For future work, we plan to explore multivariate transformations where multiple attributes may influence the transformation process. Furthermore, we plan to extend our framework to support complex and nested transformations, including those involving relational joins and cross-table enhancements. We are also considering the automatic generation of error checklists using LLMs to improve adaptability across different programming languages. Another promising future direction is the integration of LLMs with traditional program synthesis and satisfiability theory for data transformation tasks.

ACKNOWLEDGMENTS

This work is partly supported by NSF of China (62402409, 62436010, and 62441230), Guangdong Basic and Applied Basic Research Foundation (2023A1515110545), Guangzhou-HKUST (GZ) Joint Funding Program (2025A03J3714), Guangzhou Basic and Applied Basic Research Foundation (2025A04J3935), Guangzhou Municipality Big Data Intelligence Key Lab (2023A03J0012), and Guangdong provincial project 2023CX10X008.

REFERENCES

[1] Ziawasch Abedjan, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2016. Dataxformer: A robust transformation discovery system. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 1134–1145.

[2] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection. *ArXiv* abs/2310.11511 (2023).

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[4] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

[5] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[6] Chengliang Chai, Nan Tang, Ju Fan, and Yuyu Luo. 2023. Demystifying Artificial Intelligence for Data Preparation. In *SIGMOD*.

[7] Chengliang Chai, Jiayi Wang, Yuyu Luo, Zeping Niu, and Guoliang Li. 2023. Data Management for Machine Learning: A Survey. *IEEE Trans. Knowl. Data Eng.* 35, 5 (2023), 4646–4667.

[8] 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 preprint arXiv:2107.03374* (2021).

[9] Pierre-Olivier Côté, Amin Nikanjam, Nafisa Ahmed, Dmytro Humeniuk, and Foutse Khomh. 2023. Data Cleaning and Machine Learning: A Systematic Literature Review. https://doi.org/10.48550/arXiv.2310.01765 arXiv:2310.01765 [cs]

[10] Fan Cui, Chenyang Yin, Kexing Zhou, You lin Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Demin Song, Dahua Lin, Xingcheng Zhang, and Yun Liang. 2024. OriGen:Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection. *ArXiv* abs/2407.16237 (2024).

[11] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. 2013. NADEEF: a commodity data cleaning system. In *SIGMOD*. ACM, 541–552.

[12] Arash Dargahi Nobari and Davood Rafiei. 2024. DTT: An Example-Driven Tabular Transformer for Joinability by Leveraging Large Language Models. *Proc. ACM Manag. Data* 2, 1, Article 24 (March 2024), 24 pages. https://doi.org/10.1145/3639279

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

[14] Dong Deng, Wenbo Tao, Ziawasch Abedjan, Ahmed K. Elmagarmid, Ihab F. Ilyas, Guoliang Li, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Unsupervised String Transformation Learning for Entity Consolidation. In *ICDE*. IEEE, 196–207.

[15] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*. PMLR, 990–998.

[16] Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang, Zheng Yuan, Chang Zhou, and Jingren Zhou. 2023. How Abilities in Large Language Models are Affected by Supervised Fine-tuning Data Composition. *ArXiv* abs/2310.05492 (2023).

[17] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]

[18] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[19] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining Small Language Models and Large Language Models for Zero-Shot NL2SQL. *Proc. VLDB Endow.* 17, 11 (2024), 2750–2763. https://doi.org/10.14778/3681954.3681960

[20] Meihao Fan, Xiaoyue Han, Ju Fan, Chengliang Chai, Nan Tang, Guoliang Li, and Xiaoyong Du. 2024. Cost-Effective In-Context Learning for Entity Resolution: A Design Space Exploration. In *IEEE*. 3696–3709.

[21] Yingjie Fu, Bozhou Li, Linyi Li, Wentao Zhang, and Tao Xie. 2024. The First Prompt Counts the Most! An Evaluation of Large Language Models on Iterative Example-based Code Generation. *arXiv preprint arXiv:2411.06774* (2024).

[22] Khouloud Gaaloul, Claudio Menghi, Shiva Nejati, Lionel Claude Briand, and Yago Isasi Parache. 2021. Combining Genetic Programming and Model Checking to Generate Environment Assumptions. *IEEE Transactions on Software Engineering* 48 (2021), 3664–3685.

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

[24] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[25] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[26] Yuxiang Guo, Lu Chen, Zhengjie Zhou, Baihua Zheng, Ziquan Fang, Zhikun Zhang, Yuren Mao, and Yunjun Gao. 2023. Camper: An effective framework for privacy-aware deep entity resolution. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 626–637.

[27] Yuxiang Guo, Yuren Mao, Zhonghao Hu, Lu Chen, and Yunjun Gao. 2025. Snoopy: Effective and Efficient Semantic Join Discovery via Proxy Columns. *IEEE Transactions on Knowledge and Data Engineering* 37, 5 (2025), 2971–2985.

[28] Hadley Wickham and the RStudio Team. 2024. Tidyverse: Easily Install and Load the Tidyverse Packages. https://www.tidyverse.org/. Accessed: 2024-12-01.

[29] Mazhar Hameed and Felix Naumann. 2020. Data Preparation: A Survey of Commercial Tools. *SIGMOD Rec.* 49, 3 (Dec. 2020), 18–29. https://doi.org/10.1145/3444831.3444835

[30] William R Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. *ACM SIGPLAN Notices* 46, 6 (2011), 317–328.

[31] Michael Hausenblas, Boris Villazón-Terrazas, and Richard Cyganiak. 2012. Data Shapes and Data Transformations. *ArXiv* abs/1211.1565 (2012).

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

[33] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[34] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[35] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.

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

[37] Zhongjun Jin, Yeye He, and Surajit Chauduri. 2020. Auto-transform: learning-to-transform by patterns. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2368–2381.

[38] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the sigchi conference on human factors in computing systems*. 3363–3372.

[39] Moe Kayali, Anton Lykov, Ilias Fountalis, Nikolaos Vasiloglou, Dan Olteanu, and Dan Suciu. 2023. CHORUS: foundation models for unified data discovery and exploration. *arXiv preprint arXiv:2306.09610* (2023).

[40] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[41] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[42] Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025. Alpha-SQL: Zero-Shot Text-to-SQL using Monte Carlo Tree Search. In *Forty-Second International Conference on Machine Learning, ICML 2025, Vancouver, Canada, July 13-19, 2025*. OpenReview.net.

[43] Henry Lieberman. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.

[44] Bang Liu, Xinfeng Li, Jiayi Zhang, Jinlin Wang, Tanjin He, Sirui Hong, Hongzhang Liu, Shaokun Zhang, Kaitao Song, Kunlun Zhu, Yuheng Cheng, Suyuchen Wang, Xiaoqiang Wang, Yuyu Luo, Haibo Jin, Peiyan Zhang, Ollie Liu, Jiaqi Chen, Huan Zhang, Zhaoyang Yu, Haochen Shi, Boyan Li, Dekun Wu, Fengwei Teng, Xiaojun Jia, Jiawei Xu, Jinyu Xiang, Yizhang Lin, Tianming Liu, Tongliang Liu, Yu Su, Huan Sun, Glen Berseth, Jianyun Nie, Ian Foster, Logan Ward, Qingyun Wu, Yu Gu, Mingchen Zhuge, Xiangru Tang, Haohan Wang, Jiaxuan You, Chi Wang, Jian Pei, Qiang Yang, Xiaoliang Qi, and Chenglin Wu. 2025. Advances and Challenges in Foundation Agents: From Brain-Inspired Intelligence to Evolutionary, Collaborative, and Safe Systems. arXiv:2504.01990 [cs.AI] https://arxiv.org/abs/2504.01990

[45] Jiawei Liu, Chun Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *ArXiv* abs/2305.01210 (2023).

[46] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025. A Survey of NL2SQL with Large Language Models: Where are we, and where are we going? arXiv:2408.05109 [cs.DB]

https://arxiv.org/abs/2408.05109
[47] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An extensible synthesis framework for data science. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1914–1917.
[48] Nick Mecklenburg, Yiyou Lin, Xiaoxiao Li, Daniel Holstein, Leonardo Nunes, Sara Malvar, Bruno Leonardo Barros Silva, Ranveer Chandra, Vijay Aski, Pavan Kumar Reddy Yannam, Tolga Aktas, and Todd Hendry. 2024. Injecting New Knowledge into Large Language Models via Supervised Fine-Tuning. *ArXiv* abs/2404.00213 (2024).
[49] Mashaal Musleh, Mourad Ouzzani, Nan Tang, and AnHai Doan. 2020. CoClean: Collaborative Data Cleaning. In *SIGMOD*. ACM, 2757–2760.
[50] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911* (2022).
[51] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[52] Mourad Ouzzani, Nan Tang, and Raul Castro Fernandez. 2019. Data civilizer: end-to-end support for data discovery, integration, and cleaning. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, Michael L. Brodie (Ed.). ACM Books, Vol. 22. ACM / Morgan & Claypool, 291–300.
[53] Jinglin Peng, Weiyuan Wu, Brandon Lockhart, Song Bian, Jing Nathan Yan, Linghao Xu, Zhixuan Chi, Jeffrey M. Rzeszotarski, and Jiannan Wang. 2021. DataPrep.EDA: Task-Centric Exploratory Data Analysis for Statistical Modeling in Python. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China*.
[54] Fabio Petroni, Tim Rocktäschel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, Alexander H Miller, and Sebastian Riedel. 2019. Language models as knowledge bases? *arXiv preprint arXiv:1909.01066* (2019).
[55] Yichen Qian, Yongyi He, Rong Zhu, Jintao Huang, Zhijian Ma, Haibin Wang, Yaohua Wang, Xiuyu Sun, Defu Lian, Bolin Ding, et al. 2024. UniDM: A Unified Framework for Data Manipulation with Large Language Models. *Proceedings of Machine Learning and Systems* 6 (2024), 465–482.
[56] Xuedi Qin, Yuyu Luo, Nan Tang, and Guoliang Li. 2020. Making data visualization more efficient and effective: a survey. *VLDB J.* 29, 1 (2020), 93–117.
[57] Vijayshankar Raman and Joseph M Hellerstein. 2001. Potter's wheel: An interactive data cleaning system. In *VLDB*, Vol. 1. 381–390.
[58] Houxing Ren, Mingjie Zhan, Zhongyuan Wu, Aojun Zhou, Junting Pan, and Hongsheng Li. 2024. ReflectionCoder: Learning from Reflection Sequence for Enhanced One-off Code Generation. *ArXiv* abs/2405.17057 (2024).
[59] El Kindi Rezig, Lei Cao, Michael Stonebraker, Giovanni Simonini, Wenbo Tao, Samuel Madden, Mourad Ouzzani, Nan Tang, and Ahmed K. Elmagarmid. 2019. Data Civilizer 2.0: A Holistic Framework for Data Preparation and Analytics. *Proc. VLDB Endow.* 12, 12 (2019), 1954–1957.
[60] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, Online, 4902–4912. https://doi.org/10.18653/v1/2020.acl-main.442
[61] Yu-Zhe Shi, Haofei Hou, Zhangqian Bi, Fanxu Meng, Xiang Wei, Lecheng Ruan, and Qining Wang. 2024. AutoDSL: Automated domain-specific language design for structural representation of procedures with constraints. In *Annual Meeting of the Association for Computational Linguistics*.
[62] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Neural Information Processing Systems*.
[63] Roee Shraga and Renée J. Miller. 2023. Explaining Dataset Changes for Semantic Data Versioning with Explain-Da-V (Technical Report). *Proc. VLDB Endow.* 16 (2023), 1587–1600.
[64] Rishabh Singh. 2016. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment* 9, 10 (2016), 816–827.
[65] Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. 2017. Synthesizing Entity Matching Rules by Examples. *Proc. VLDB Endow.* 11, 2 (2017), 189–202.
[66] Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. ARKS: Active Retrieval in Knowledge Soup for Code Generation. *ArXiv* abs/2402.12317 (2024).
[67] Tableau Software. 2024. Tableau: Business Intelligence and Analytics. https://www.tableau.com/. Accessed: 2024-12-01.
[68] Aarne Talman, Marianna Apidianaki, Stergios Chatzikyriakidis, and Jörg Tiedemann. 2021. NLI Data Sanity Check: Assessing the Effect of Data Corruption on Model Performance. In *Nordic Conference of Computational Linguistics*.
[69] Nan Tang, Ju Fan, Fangyi Li, Jianhong Tu, Xiaoyong Du, Guoliang Li, Samuel Madden, and Mourad Ouzzani. 2021. RPT: Relational Pre-trained Transformer Is Almost All You Need towards Democratizing Data Preparation. *Proc. VLDB Endow.* 14, 8 (2021), 1254–1261.
[70] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model.
[71] Fengwei Teng, Zhaoyang Yu, Quan Shi, Jiayi Zhang, Chenglin Wu, and Yuyu Luo. 2025. Atom of Thoughts for Markov LLM Test-Time Scaling. *CoRR* abs/2502.12018 (2025).
[72] Trifacta, Inc. 2024. Trifacta: Data Wrangling for Machine Learning and Analytics. https://www.trifacta.com/. Accessed: 2024-12-01.
[73] Jianhong Tu, Ju Fan, Nan Tang, Peng Wang, Guoliang Li, Xiaoyong Du, Xiaofeng Jia, and Song Gao. 2023. Unicorn: A Unified Multi-tasking Model for Supporting Matching Tasks in Data Integration. *Proc. ACM Manag. Data* 1, 1 (2023), 84:1–84:26. https://doi.org/10.1145/3588938
[74] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).
[75] Chenglong Wang, Yu Feng, Rastislav Bodík, Işıl Dillig, Alvin Cheung, and Amy J. Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (2021).
[76] Jiannan Wang and Nan Tang. 2014. Towards dependable data repairing with fixing rules. In *SIGMOD*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 457–468.
[77] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024. CodeRAG-Bench: Can Retrieval Augment Code Generation? *ArXiv* abs/2406.14497 (2024).
[78] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, F. Xia, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *ArXiv* abs/2201.11903 (2022).
[79] Yeming Wen, Pengcheng Yin, Kensen Shi, Henryk Michalewski, Swarat Chaudhuri, and Alex Polozov. 2024. Grounding Data Science Code Generation with Input-Output Specifications. *arXiv preprint arXiv:2402.08073* (2024).
[80] Danny Weyns, Nelly Bencomo, Radu Calinescu, Javier Cámara, Carlo Ghezzi, Vincenzo Grassi, Lars Grunske, Paola Inverardi, Jean-Marc Jézéquel, Sam Malek, Raffaela Mirandola, Marco Mori, and Giordano Tamburrelli. 2019. Perpetual Assurances for Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems*.
[81] Yifan Wu, Lutao Yan, Leixian Shen, Yunhai Wang, Nan Tang, and Yuyu Luo. 2024. Chartinsights: Evaluating multimodal large language models for low-level chart question answering. *arXiv preprint arXiv:2405.07001* (2024).
[82] Jinyu Xiang, Jiayi Zhang, Zhaoyang Yu, Fengwei Teng, Jinhao Tu, Xinbing Liang, Sirui Hong, Chenglin Wu, and Yuyu Luo. 2025. Self-Supervised Prompt Optimization. *CoRR* abs/2502.06855 (2025).
[83] Xiao Yang, Kai Sun, Hao Xin, Yushi Sun, Nikita Bhalla, Xiangsen Chen, Sajal Choudhary, Rongze Daniel Gui, Ziran Will Jiang, Ziyu Jiang, Lingkun Kong, Brian Moran, Jiaqi Wang, Yifan Xu, An Yan, Chenyu Yang, Eting Yuan, Hanwen Zha, Nan Tang, Lei Chen, Nicolas Scheffer, Yue Liu, Nirav Shah, Rakesh Wanga, Anuj Kumar, Scott Yih, and Xin Dong. 2024. CRAG - Comprehensive RAG Benchmark. In *NeurIPS*.
[84] Haochen Zhang, Yuyang Dong, Chuan Xiao, and Masafumi Oyamada. 2023. Large language models as data preprocessors. *arXiv preprint arXiv:2308.16361* (2023).
[85] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2024. AFlow: Automating Agentic Workflow Generation. *CoRR* abs/2410.10762 (2024).
[86] Zhengxuan Zhang, Yin Wu, Yuyu Luo, and Nan Tang. 2024. MAR: Matching-Augmented Reasoning for Enhancing Visual-based Entity Question Answering. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 1520–1530. https://doi.org/10.18653/v1/2024.emnlp-main.91
[87] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. Association for Computational Linguistics, Bangkok, Thailand. http://arxiv.org/abs/2403.13372
[88] Yizhang Zhu, Shiyin Du, Boyan Li, Yuyu Luo, and Nan Tang. 2024. Are Large Language Models Good Statisticians?. In *NeurIPS*.