DATAMORPHER: Automatic Data Transformation Using LLM-based Zero-Shot Code Generation

Ankita Sharma[a], Jaykumar Tandel[a], Xuanmao Li[a], Lanjun Wang[b], Anna Fariha[c], Liang Zhang[d], Syed Arsalan Ahmed Naqvi[e], Irbaz Bin Riaz[e], Lei Cao[d], Jia Zou[a]
Arizona State University[a], Tianjin University[b], University of Utah[c], University of Arizona[d], Mayo Clinic-Arizona[e]

***Abstract*—Data transformation is a critical challenge in modern data management systems, particularly when handling complex operations over multiple data sources. However, existing approaches rely on supervised learning, which requires tremendous data labeling and training overhead. To alleviate such overhead while improving accuracy, we demonstrate a novel system DATAMORPHER that leverages Large Language Models (LLMs) to generate code that transforms source datasets into a user-specified target format. To generate a high-quality and token-efficient prompt, we leverage data profiling to extract features from the source datasets and historical examples of the target data. We also select a subset of features to reduce noise and costs using a ranking algorithm. These selected features are finally translated into a declarative language, which is inspired by SQL's data definition language (DDL), before being added to the prompt. We will demonstrate the workflow and effectiveness of DATAMORPHER using real-world data transformation workflows from Microsoft's GitHub benchmark, smart building, and medical data integration. (A 5-min video of our demo is available at https://youtu.be/CuDm46K-_eA).**

I. INTRODUCTION

Integrating and transforming data collected from heterogeneous sources into a desired target format requires tremendous human domain expertise and manual efforts, estimated to account for more than 90% of the total costs associated with data science pipelines [1]. For example, in a movie recommendation pipeline, the `user rating` table needs to be joined with the `movie` table and the `user` table to extract features, which will be fed to a recommendation model to obtain recommendation decisions. A small change in the source datasets, e.g., renaming `movie_id` to `title_id`, will cause the data-integration code to fail, resulting in a crash of the recommendation pipeline. Repetition of human efforts can be required to locate and fix such issues. To automate the data integration process and alleviate human efforts, we focus on the following problem in this demonstration.

Problem Definition. Given (1) multiple source datasets, (2) a schema of the transformed data, termed *target schema*, and (3) examples that illustrate the patterns of the transformed data, termed *target examples*, which *do not* need to correspond to any tuples in the source dataset, we would like to provide a mechanism that automatically transforms the source datasets to the *target data* that conforms to the *target schema* and follows the patterns of the *target examples*. It is known as the *Transform-By-Target* (TBT) problem [2].

State-of-the-art approaches to the TBT problems [2], [3] often struggle with achieving high accuracy in complex scenarios that involve large, heterogeneous source files, and numerous transformation operators. They also heavily use supervised learning models and suffer from tremendous training data labeling and training overheads.

Recent developments in large language models (LLMs) have shown promise in automating complex transformations via zero-shot learning, eliminating the need for curated training data. At a high level, there are two approaches to using LLMs: (1) *A data generation approach*, which packs all the source datasets, the target schema, and the target examples into a prompt and instructs an LLM to output the target dataset. and (2) *A code generation approach*, which instructs an LLM to generate code that will transform the source datasets to the target. In this work, we adopt the code generation approach since the data generation approach does not scale to large-scale datasets, considering the token limit and the expensive costs associated with large LLM prompts and responses. For instance, OpenAI's ChatGPT-4-turbo API (used in this work) charges $10 per million of prompt tokens and $30 per million of sampled tokens. ChatGPT-4o is even more expensive.

Since the focus of our work is to reduce human efforts, we do not consider straightforward human-centric code generation, where a user iteratively fine-tunes prompts using natural language to guide an LLM to resolve each data transformation problem. Instead, we argue for an automated approach to generate and finetune the prompt to interact with the LLM for the target problems to satisfy the following challenging objectives. *(1) Prompt Informativeness*: The automatically generated prompt should be sufficiently informative for the LLM to generate the correct data transformation code. *(2) Prompt Cost-Effectiveness*: While informativeness is guaranteed, it is important to minimize the size of the prompt to reduce the monetary costs. *(3) Prompt Interpretability*: The prompt should be easily understandable to facilitate auditing and debugging to reduce the development and deployment costs. *(4) Prompt Robustness*: The generated prompt should be iteratively and automatically improved to fix errors observed in executing the LLM-generated code.

DATAMORPHER. To achieve these objectives, we present and demonstrate DATAMORPHER, a novel LLM-based zero-shot data transformation approach, consisting of four critical components as illustrated in Fig. 1: *(1) Feature Extraction based on Data Sampling and Profiling:* We found that a prompt mostly consisting of samples of data tuples from the

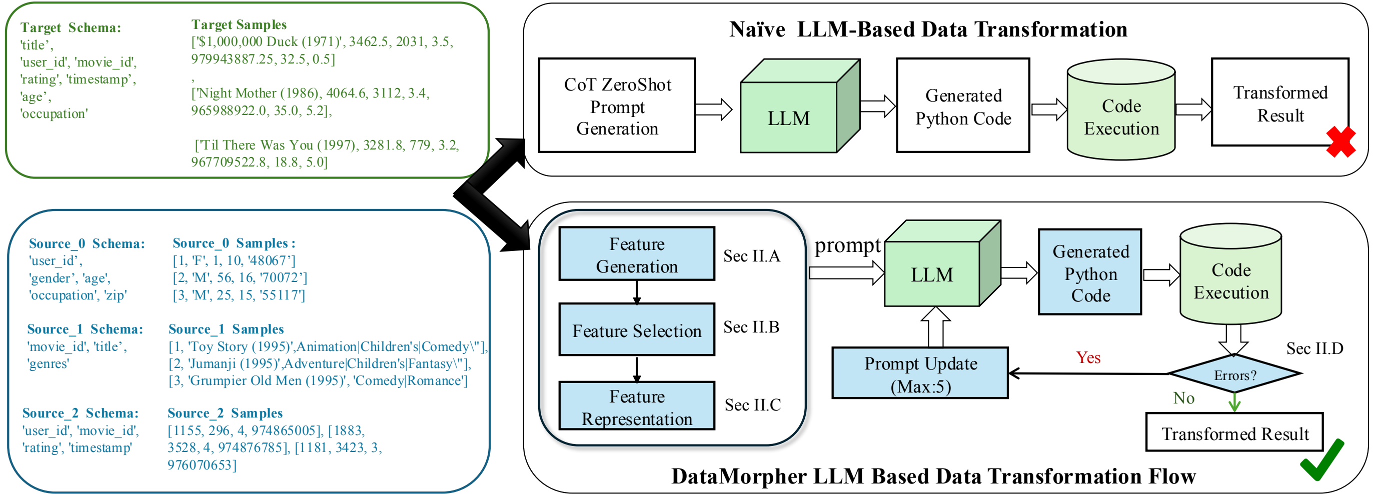

Fig. 1: System Overview and Comparison to the Naive LLM-based Data Transformation using Chain-of-Thoughts in zeroshot style (CoT-Zeroshot)

source datasets and the target examples is not informative enough to generate correct code for complex data transformation cases, and it is important to profile table-level, column-level, cross-column statistics of the datasets as detailed in Sec. II-A. Each statistic value is indexed by the metric name and context, composing a feature in the format of a key-value pair. *(2) Feature Selection based on Ranking of Relevance:* We also found that data profiles for complex transformation problems that involve thousands of columns from hundreds of source datasets could exceed the token limit due to the exponentially increased number of attribute pairs for computing cross-attribute statistics. Therefore, we select features by grouping the features based on the similarity of key, ranking the values sharing similar keys, and selecting the most relevant values, as detailed in Sec. II-B. *(3) Feature Representation using a SQL DDL-like language:* Each selected feature will be translated into a statement following a declarative language that is similar to SQL's Data Definition Language (DDL) for users (and LLMs) to easily understand the features, described in Sec. II-C. *(4) Code Generation and Data Transformation:* The LLM-generated code is run in a sandbox to transform the source datasets into the target dataset. The execution errors will be captured and appended to the prompt for repeated execution. See Sec. II-D for more details.

The key contributions of this work include:

- We develop DATAMORPHER, a novel tool that leverages LLM to automatically generate code to transform user-provided source datasets to conform to the target data format specified by the user, using zero-shot learning.
- DATAMORPHER automatically generates a cost-effective and interpretable prompt through our unique feature extraction, feature selection, and feature representation techniques.
- We deploy DATAMORPHER as a web application with an interactive and easy-to-use interface. We will demonstrate its effectiveness using a diverse set of data transformation scenarios from the GitHub benchmark [2] developed by Microsoft, which consists of 700 cases crawled from public GitHub repositories, 105 data integration cases for smart building energy usage prediction, and 11 medical data analysis cases.

II. SYSTEM ARCHITECTURE AND EVALUATION

As illustrated in Fig. 1, DATAMORPHER consists of the following key components (features are also termed as hints):

A. Feature Extraction

For large-scale source datasets and target examples, it is important to extract a small amount of critical information to reduce the prompt size and improve the accuracy of LLM inferences. To this end, we generate a data summary that includes information useful for inferring the data transformation operators and their properties (e.g., predicates, functions, attributes) [4] by profiling the source datasets and target examples, including but not limited to the following:

- **Table-Level Features** such as *number of tuples* in a table and *functional dependencies (FD) analysis* in the target examples to identify the FD constraints that must be preserved during the data transformation process.
- **Column-Level Features** including (1) *column index* of each column in its table, which is important for predicting key columns, since key attributes are often placed on the leftmost side of a table; (2) *column distribution statistics*, such as max, min, medium, average, and quantiles in a numerical column, and peak frequency of distinct values in a categorical column; (3) *Column sortness*, e.g., whether the values in a column are sorted. It can be used to identify key, since tables are often sorted by their primary key attributes; (4) *Number of missing values* in a column; (5) *Cardinality*, i.e., the number of distinct values in a specified column. Based on the cardinality and the number of tuples, we can derive the distinct value ratio. If the distinct value ratio is one, it indicates the column follows the Uniqueness constraint.
- **Cross-Table Column-Pair Features** such as *inclusion dependency between two columns* by checking whether two columns share a lot of common values using the Jaccard similarity of two discrete columns. For columns that have

continuous values, we use we use Kolmogorov–Smirnov test to determine whether two columns have similar distributions.

B. Feature Selection

The total number of features increases with the number of tables and the total number of columns. Particularly, the number of column-pair features increases exponentially with the total number of columns. For large-scale data transformation problems, it is important to prune the features to reduce noise and keep the prompt size small to reduce costs and inference latency. To achieve the goal, we conduct feature selection according to their relevance with potential data transformation operators, with the following examples. (1) For `join`, we rank column pairs based on the probability that they will match with each other and may serve as attributes in a predicate of an `equi-join`. Since most of `equi-joins` are along the foreign key relationships, we first prune column pairs where both columns' indexes are greater than k_1, both columns are unsorted, of categorical type, with the distinct value ratio smaller than 1, and the missing value ratio greater than 0, which indicates that both columns have small probabilities to be a key attribute in its table. In the rest of the column pairs, we only keep those that have the highest k_2 percentage of Jaccard similarity to show their column mapping features. (2) For `group-by`, we rank columns based on the column's type, missing value ratio, distinct value ratio, and peak frequency. These discrete columns with low missing value ratio, low distinct value ratio, and high peak frequency are more likely to be a `group-by` attribute. We select the top-k_4 percentage of columns ranked accordingly to show the column-level features. In addition to the aforementioned features, we also sample k_5 percentage of examples from the source datasets and the target examples using a uniform sampling approach. We further finetune the value of hyperparameters (such as k_1, k_2, k_3, k_4, and k_5) through Bayesian optimization.

C. Feature Representation

To effectively represent the profiled statistics for users to better understand the selected features, we designed a declarative language by augmenting the SQL DDL. We chose SQL DDL as the base language because (1) Compared to other existing structured languages, SQL is the most relevant to data transformation tasks, which mostly involve relational operators and data. In addition, SQL DDL is the most suitable for describing data profiling results since it is designed to describe/define the types and integrity constraints of data, which are meant to be discovered by the data profiling operators. Compared to designing a new language, LLM is more familiar with SQL DDL, given its popularity and representativeness in publicly available code bases.

We leveraged the column unique/non-null/check statement, table primary key and foreign key definitions, etc., from SQL DDL. For example, a column having high *inclusion dependency* with the `key` of a remote table, is a strong indicator of a foreign key relationship. Therefore, we leverage the foreign key constraints in SQL DDL to represent such dependencies, e.g., `CONSTRAINT FOREIGN KEY ASU_Person (ID) REFERENCES Persons(PersonID)`. Due to the limited set of commands available in SQL DDL, we also extend it to represent more profiling results. For example, to represent the `functional dependency` relationship, we added keywords such as `DETERMINES`, e.g.,`Target(ItemID) DETERMINES Count, Price`, representing that in the target table, if ItemID is given, the values of Count and Price are determined.

D. Code Generation and Execution

Once the prompt is created, it is sent to the LLM to generate the data transformation code. When the response from LLM is received, it will be parsed and the code will be extracted and executed in a sandbox to avoid security risks. We will execute the code multiple times. If it meets an execution error, e.g., thrown by the Python script or the relational database (i.e., we used PostgreSQL), the error(s) will be appended to the prompt and sent to the LLM again in the hope that LLM will resolve the issue in the next response. The examples of errors that we met and are resolved by LLM through iterative execution include "name 'datetime' is not defined" for SQL and "module 'pandas' has no attribute 'read_corpus"' for Python. The iteration will continue until the execution is successful or the maximal number of executions has been reached (five times by default).

Evaluation. We evaluate DATAMORPHER using three types of workload, detailed as follows: *(1) Github Benchmark [2],* which includes 700 data transformation cases crawled from the Github public repositories by Microsoft. In this benchmark, we compared our approach with three baselines, the naive LLM-based data transformation using Chain-of-Thoughts (CoT) planning in a zero-shot style, called CoT-zeroshot, AutoPipeline [2], and AutoSuggest [3]. The results showed that DATAMORPHER can successfully transform 79% of 700 cases, while CoT-zeroshot achieved 40%, AutoPipeline achieved 77%, and AutoSuggest achieved 30%. Note that both AutoPipeline and AutoSuggest rely on multiple supervised learning models, while DATAMORPHER is zero-shot. (AutoPipeline and AutoSuggest are not publicly available, and their results are obtained from [2].) (2) *Smart Building Benchmark [5]*, which consists of 105 real-world data transformation cases collected from 20 utility companies in the United States. On this benchmark, DATAMORPHER achieved 77% success rate, outperforming CoT-zeroshot (48%) and SQL-Morpher [5] (28%). (3) *Synthetic Medical Data Transformation.* Our study employs 11 cases representing structured clinical data that were synthetically generated by two Mayo Clinic clinicians, mapped from electronic health records (EHR), and datasets for medical systematic reviews and meta-analysis. In this dataset, CoT-zeroshot only successfully transformed four cases, while DATAMORPHER's success rate is 91%.

III. DEMONSTRATION SCENARIOS

We developed a web interface for DATAMORPHER as illustrated in Fig. 2. We demonstrate it on an example from the

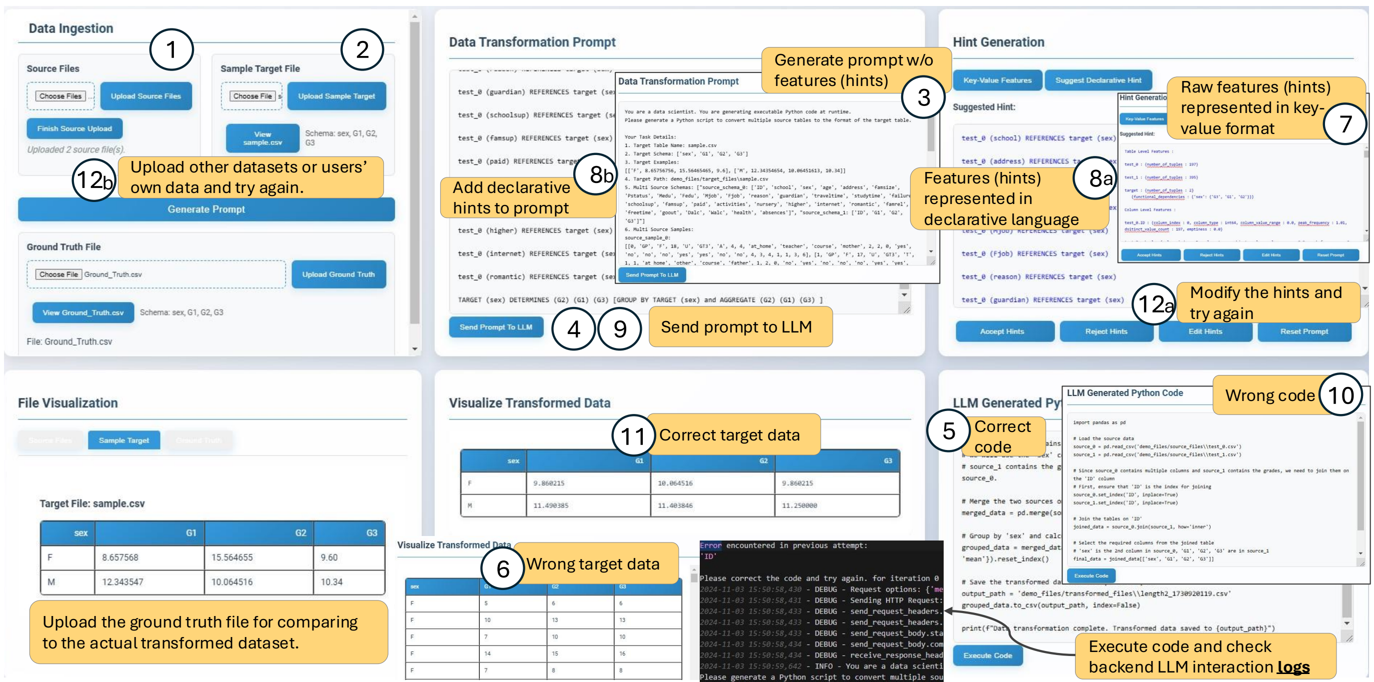

Fig. 2: DATAMORPHER GUI and Demonstration Outline.

Github benchmark that integrates two source datasets (Test0 and Test1) into a target dataset using `join`, `groupby`, and `aggregate`. We will also encourage users to try their own datasets and/or other data transformation cases that we used for evaluation. A demonstration consists of the following steps:

Step (1): Upload source datasets, such as test0.csv and test1.csv, of which the schemas are extracted and shown, and their data can be reviewed using File Visualization.

Step (2): Upload target examples, e.g., target.csv, which can be a target dataset resulting from a previous integration of earlier versions of the source datasets, of which the schema and data can also be reviewed by the user.

Step (3): Generate the naive prompt without any features (hints). The prompt includes a few samples of test0.csv, test1.csv, and target.csv, a description of the data transformation task, and an instruction that asks LLM to reason and plan the steps of the data transformation process.

Step (4): The naive prompt is sent to the LLM to generate a Python script. Users can review the script and experienced users will notice that the generated code missed the `groupBy`.

Step (5): The user will run the script to transform the source into the target in a sandbox. The user can check the background logs to monitor how the tool interacts with LLM iteratively to fix execution errors.

Step (6): Users can visualize the target dataset and will find it does not follow the expected target schemas, which indicates a failure of the transformation process.

Step (7): Trigger the data profiling process to generate features (hints) represented in naive key-value format. Users can scroll down to review table-level, column-level, and column-pair-level raw features, such as *target: functional_dependencies: {'sex': {'G3', 'G2', 'G1'}}*.

Step (8): Select features represent them in our declarative language to reduce the number of features (hints) and improve their interpretability, e.g., *Target ('sex') Determines ({'G3', 'G2', 'G1'})* (8a), and add hints to the prompt (8b).

Steps (9) and (10): Similar to steps (4) and (5).

Step (11): Similar to step (6). The user will find that the target data exactly meets the requirement and further confirm the consistency by uploading the ground truth file for comparison.

Step (12): Users can edit the selected hints,e.g., removing or adding a hint, modifying the representation of a hint, to check how the selection and representation of the hints will influence the quality of the generated code and transformed datasets (12a). They can also choose different transformation cases or upload their own data for more interactions (12b).

REFERENCES

[1] D. Abadi, A. Ailamaki, D. Andersen, P. Bailis, M. Balazinska, P. Bernstein, P. Boncz, S. Chaudhuri, A. Cheung, A. Doan *et al.*, "The seattle report on database research," *ACM Sigmod Record*, vol. 48, no. 4, pp. 44–53, 2020.

[2] J. Yang, Y. He, and S. Chaudhuri, "Auto-pipeline: synthesizing complex data pipelines by-target using reinforcement learning and search," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2563–2575, 2021.

[3] C. Yan and Y. He, "Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1539–1554.

[4] Z. Jin, Y. He, and S. Chauduri, "Auto-transform: learning-to-transform by patterns," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2368–2381, 2020.

[5] A. Sharma, X. Li, H. Guan, G. Sun, L. Zhang, L. Wang, K. Wu, L. Cao, E. Zhu, A. Sim *et al.*, "Automatic data transformation using large language model an experimental study on building energy data," *arXiv preprint arXiv:2309.01957*, 2023.