CodeGenWrangler: Data Wrangling task automation using Code-Generating Models

Ashlesha Akella
IBM Research, India
ashlesha.akella@ibm.com

Abhijit Manatkar
IBM Research, India
abhijitmanatkar@ibm.com

Krishnasuri Narayanam
IBM Research, India
knaraya3@in.ibm.com

Sameep Mehta
IBM Research, India
sameepmehta@in.ibm.com

Abstract

Assuring the data quality of tabular datasets is essential for the efficiency of the diverse tabular downstream tasks (like summarization and fact-checking). Data-wrangling tasks effectively address the challenges associated with structured data processing to improve the quality of tabular data. Traditional statistical methods handle numeric data efficiently but often fail to understand the semantic context of the textual data in tables. Deep learning approaches are resource-intensive, requiring task and dataset-specific training. Addressing these shortcomings, we present an automated system that leverages LLMs to generate executable code for data-wrangling tasks like missing value imputation, error detection, and error correction. Our system aims to identify inherent patterns in the data while leveraging external knowledge, effectively addressing both memory-independent and memory-dependent tasks.

1 Introduction

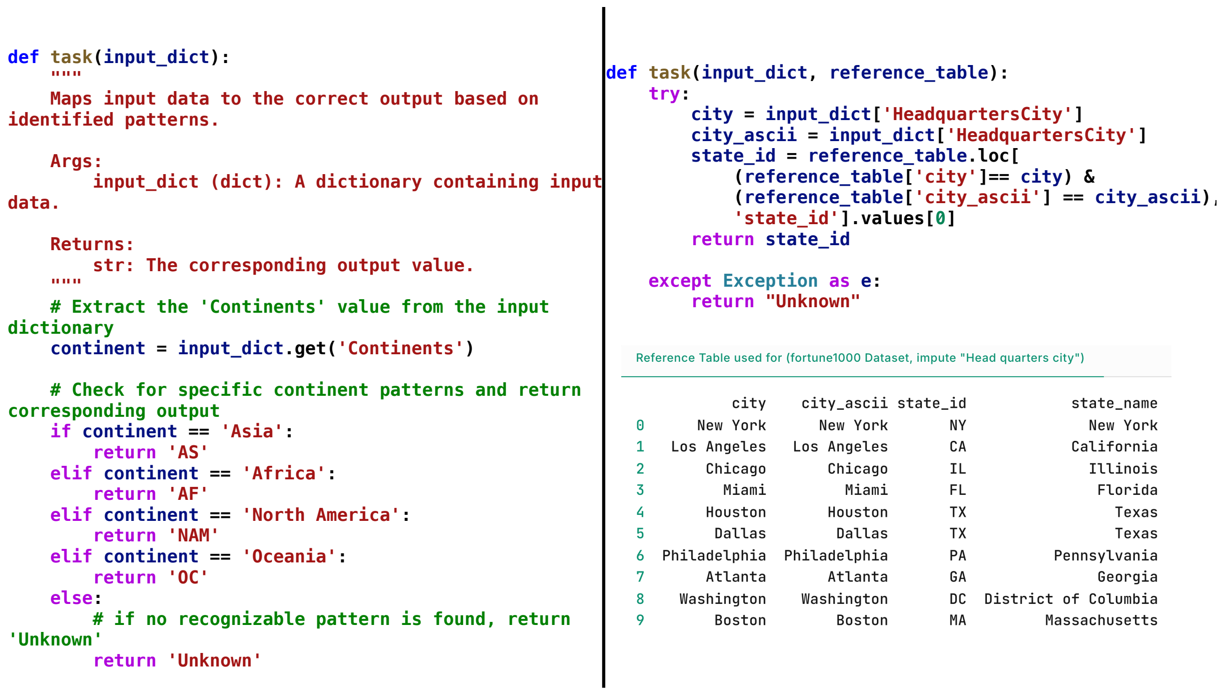
Tabular datasets in industrial settings frequently encompass extensive data with numerous rows and columns. Given the pivotal role of this data in informed business decision-making (via exercising diverse tabular downstream tasks), maintaining high data quality has become increasingly crucial. Data wrangling tasks (like imputing missing values or correcting errors) are vital in enhancing the quality of tabular datasets. Such tasks require both statistical insights and domain-specific semantic understanding. Statistical methods (Van Buuren, 2018; Gong et al., 2021; Thomas and Rajabi, 2021) cannot often incorporate semantics or external context (e.g., imputing city from zip code), limiting their effectiveness in complex industrial datasets. Deep learning approaches (Lin et al., 2022; Samad et al., 2022; Huang et al., 2024) can capture intricate patterns but require dataset-specific training, which is computationally expensive for large datasets.

Large language models (LLMs) offer new potential for data wrangling (Iida et al., 2021; Narayan et al., 2022; Huh et al., 2023; Jaimovitch-López et al., 2023; Liu et al., 2023b, 2024; Ashlesha et al., 2024; Li and Döhmen, 2024) tasks by leveraging broad contextual knowledge. Trained on extensive datasets, these models hold vast knowledge that enables contextual insights and supports semantically informed data wrangling. However, the need to invoke LLM inference calls independently for each row (Narayan et al., 2022) incurs high computational costs and adds latency, making it difficult to scale for large datasets.

To address these challenges, we introduce `CodeGenWrangler`, which leverages code-generating LLMs for efficient data wrangling. Tabular datasets often contain inherent patterns with dependencies between specific columns. Our system identifies such data patterns, represents them as concisely formulated rules, and translates them into executable code for data wrangling tasks to enhance scalability by eliminating the need for row-level LLM inference calls.

While existing study (Li and Döhmen, 2024) has demonstrated the efficacy of code-generating LLMs in translating data patterns into executable code for data-wrangling tasks, their system is constrained by the language model's outdated knowledge and lack of the ability to incorporate external or domain-specific enterprise data. This can be effectively addressed using Retrieval-Augmented Generation (RAG) (Lewis et al., 2020; Liu et al., 2023a) by enhancing the model's capacity to retrieve context-specific knowledge to improve accuracy and relevance.

Proposed `CodeGenWrangler` system employs a tailored prompt design and two pipeline



```python
def task(input_dict):
    """
    Maps input data to the correct output based on
identified patterns.

    Args:
        input_dict (dict): A dictionary containing input
data.

    Returns:
        str: The corresponding output value.
    """
    # Extract the 'Continents' value from the input
dictionary
    continent = input_dict.get('Continents')

    # Check for specific continent patterns and return
corresponding output
    if continent == 'Asia':
        return 'AS'
    elif continent == 'Africa':
        return 'AF'
    elif continent == 'North America':
        return 'NAM'
    elif continent == 'Oceania':
        return 'OC'
    else:
        # if no recognizable pattern is found, return
'Unknown'
        return 'Unknown'
```

```python
def task(input_dict, reference_table):
    try:
        city = input_dict['HeadquartersCity']
        city_ascii = input_dict['HeadquartersCity']
        state_id = reference_table.loc[
            (reference_table['city']== city) &
            (reference_table['city_ascii'] == city_ascii),
            'state_id'].values[0]
        return state_id

    except Exception as e:
        return "Unknown"
```

Reference Table used for (fortune1000 Dataset, impute "Head quarters city")

```
          city   city_ascii state_id            state_name
0     New York     New York       NY              New York
1  Los Angeles  Los Angeles       CA            California
2      Chicago      Chicago       IL              Illinois
3        Miami        Miami       FL               Florida
4      Houston      Houston       TX                 Texas
5       Dallas       Dallas       TX                 Texas
6 Philadelphia Philadelphia       PA          Pennsylvania
7      Atlanta      Atlanta       GA               Georgia
8   Washington   Washington       DC  District of Columbia
9       Boston       Boston       MA         Massachusetts
```

Figure 1: Illustrative examples of code snippets generated by the CodeGenWrangler system, demonstrating its ability to handle data wrangling for Memory Independent (Left) and Memory Dependent (Right) tasks. A few more code snippets are shown in Appendix B

routes—one external memory-dependent (to integrate relevant external knowledge), the other memory-independent. An iterative refinement process further optimizes the generated code, addressing challenges such as efficiently selecting sample data for prompts. Later sections describe the full technical details of our proposed system (and an overview of our system demonstration is available at (Ashlesha and Narayanam, 2025)).

2 Background

Recent studies (Wang and Chen, 2023; Zan et al., 2023; Jiang et al., 2024) have shown that LLMs are capable of functioning as code generation models, which can generate code by interpreting natural language instructions (Jiang et al., 2022; Wang et al., 2023; Dong et al., 2024), complete partially written code (Barke et al., 2023; Guo et al., 2023), and fix buggy code (Fan et al., 2023; Joshi et al., 2023; Zhang et al., 2024) due to their extensive training on vast source code data. However, we sought to investigate if these models could also recognize logical patterns in the data without requiring explicit descriptions to determine their potential for handling data-wrangling tasks. These models when prompted with sample data and instructions, we observed that their generated code aligned with the inherent patterns in the sample data (Figure 1). However, leveraging code-generating LLMs to automate data wrangling presents several challenges: (i) addressing tasks that depend on external or enterprise-specific knowledge beyond the dataset for accuracy (ii) correctly handling complex patterns in the data that go beyond simple one-to-one mappings requires coherent integration of different control flows in the code (iii) providing optimal data samples in prompts to ensure comprehensive coverage of data patterns (iv) determining which columns of the given dataset should be presented to the LLM for effective performance on specific wrangling tasks. Section 3 explains how our system addresses these challenges.

3 Method

The CodeGenWrangler system (shown in Figure 2) takes as input a dataset $D = [c_1, \ldots, c_n]$, where each c_i is an attribute (column) of the dataset, a target column c_T, and a data wrangling task, such as data imputation (DI), error detection (ED), or error correction (EC).

For DI, the task is to predict the missing values of the dataset column $D[c_T]$. For ED, the task is to identify the erroneous entries in $D[c_T]$, and for EC, the task is to detect erroneous entries in $D[c_T]$ and impute them.

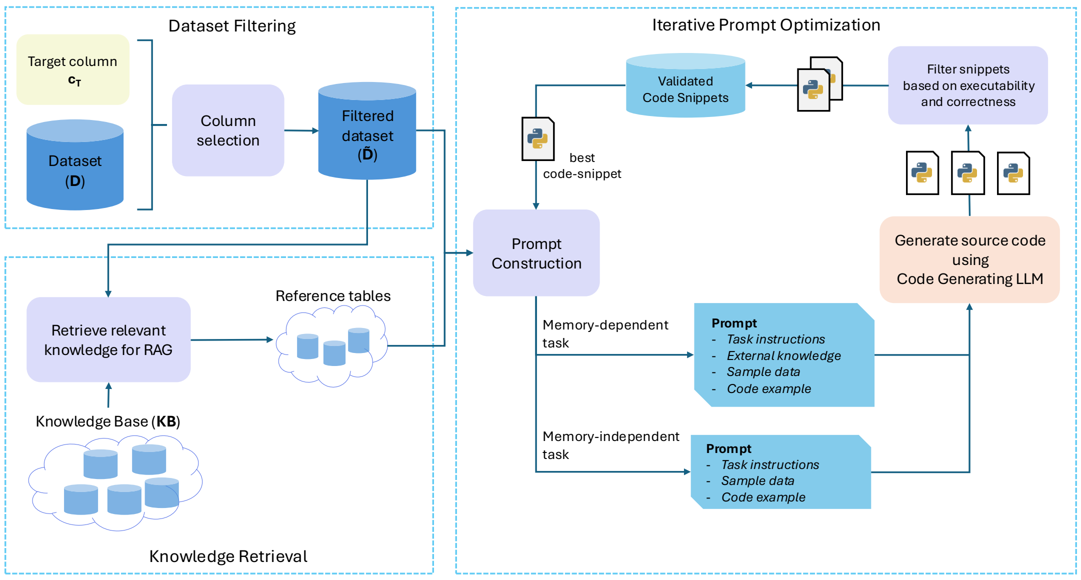

Dataset Filtering
Target column c_T
Dataset (D)
Column selection
Filtered dataset ($\tilde{D}$)
Iterative Prompt Optimization
Validated Code Snippets
Filter snippets based on executability and correctness
best code-snippet
Prompt Construction
Generate source code using Code Generating LLM
Memory-dependent task
Prompt - Task instructions - External knowledge - Sample data - Code example
Memory-independent task
Prompt - Task instructions - Sample data - Code example
Retrieve relevant knowledge for RAG
Reference tables
Knowledge Base (KB)
Knowledge Retrieval

Figure 2: Dataset Filtering and Knowledge Retrieval of `CodeGenWrangler` system extract relevant information before it automatically generates a few code snippets in iterations that collectively capture the data wrangling task.

3.1 Datasets

We used datasets from (Narayan et al., 2022; Ashlesha et al., 2024), which were collected from various sources like Kaggle[1] and OpenML[2]. These datasets span across multiple domains and contain numerous columns and rows. Each dataset is split (as in (Ashlesha et al., 2024)) into three sets: a *train* set for iteratively constructing and improving the prompt for obtaining the optimal code snippets, a *validation* set for validating the performance of intermediate code snippets, and a final *test* set where we evaluate the performance of our system.

3.2 Dataset Filtering for Relevant Columns

Given D, c_T and a task, the system identifies relevant columns by calculating permutation importances for each column in a learned Histogram-based Gradient Boosting Classification Tree (Guryanov, 2019) for predicting the target column. The relevant columns $\tilde{c} = [c_1^*, \dots c_k^*]$ with the highest permutation importances are selected to form a subset of the data, denoted as $\tilde{D} = D[\tilde{c}, c_T]$, which contains only the relevant and target columns. This helps reduce noise and ensures the code generation LLM focuses on essential data patterns within its limited context length.

[1]https://www.kaggle.com
[2]https://openml.org

3.3 Knowledge Retrieval

In addition to the LLM's parametric memory, the knowledge required for the LLM to generate source code can come from multiple other sources: it may be derived directly from the dataset itself (e.g., set the `24_hour_service` column in the Starbucks dataset (Alice, 2017) to 'True' if the values of both the columns `opening_time` and `closing_time` are midnight), or it may come from external or enterprise datasets (e.g., mapping cities to respective states or imputing job role based on job title).

To accommodate knowledge inclusion from varied sources, our system employs two parallel modules for generating code snippets: a **memory-independent module**, which relies solely on patterns derived directly from the dataset, and a **memory-dependent module**, which incorporates relevant contextual knowledge from the external knowledge base $KB = \{T_1, T_2, \dots, T_m\}$, where each T_i is a tabular data. To retrieve the relevant knowledge, we compute semantic similarity between the sample rows of dataset D and each table $T_i \in KB$.

Let the embedding of a row r of any dataset be denoted by $\mathbf{e}_r$, computed as:

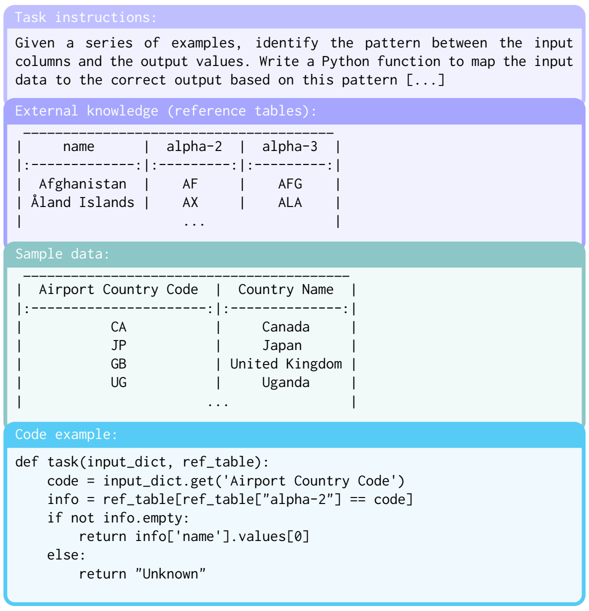
$$\mathbf{e}_r = \text{concat}(\mathbf{h}_r^1, \mathbf{h}_r^2, \dots, \mathbf{h}_r^n),$$

where $\mathbf{h}_r^j = \text{LLM}(r[c_j])$ is the hidden state computed by the encoder-only language model (`all-miniLM-L6-v2` in our setup) for the j^{th} column c_j of the row r. The similarity score $sim(T_i)$ for a table T_i is computed as:

$$sim(T_i) = \sum_{\substack{r_D \in D \\ r_{T_i} \in T_i}} \mathbf{e}_{r_D}^{\mathsf{T}} \mathbf{e}_{r_{Ti}},$$

where $\mathbf{e}_{r_D}$ and $\mathbf{e}_{r_{T_i}}$ are the embeddings of the rows r_D (sampled from D) and r_{T_i} (sampled from T_i) respectively. We select the top-k tables $\mathcal{T} = \{T_1^*, \ldots, T_k^*\}$ such that the similarity score $sim(T_i^*)$ exceeds a fixed threshold.

Further, the **memory-independent module** has two types of tasks: **row-level tasks**, which use only the data in the current row to generate code (e.g., imputing the `24_hour_service` column using `opening_time` and `closing_time` columns), and **exemplar-based tasks**, where patterns are inferred from a small set of examples in the prompt.



```
Task instructions:
Given a series of examples, identify the pattern between the input
columns and the output values. Write a Python function to map the input
data to the correct output based on this pattern [...]

External knowledge (reference tables):
 --------------------------------------
|     name      |  alpha-2  |  alpha-3  |
|:-------------:|:---------:|:---------:|
|  Afghanistan  |    AF     |    AFG    |
| Åland Islands |    AX     |    ALA    |
|                   ...                 |

Sample data:
 ----------------------------------------
|  Airport Country Code  |  Country Name  |
|:----------------------:|:--------------:|
|           CA           |     Canada     |
|           JP           |     Japan      |
|           GB           | United Kingdom |
|           UG           |     Uganda     |
|                       ...                |

Code example:
def task(input_dict, ref_table):
    code = input_dict.get('Airport Country Code')
    info = ref_table[ref_table["alpha-2"] == code]
    if not info.empty:
        return info['name'].values[0]
    else:
        return "Unknown"
```

Figure 3: Prompt template for code generation

3.4 Prompt Construction

For each module above, code is generated by prompting a code-generating LLM, requiring a narrowly tailored prompt structure. The prompt consists of the following components. **Task instructions**: contains a description of the task to instruct the LLM to detect patterns in the data and write a Python function corresponding to the task. **External knowledge** (reference tables): with *memory-dependent tasks*, a set of rows retrieved from relevant tables from the external knowledge base. **Sample data**: a small subset of rows sampled from the dataset. For *exemplar-based tasks*, a few additional rows from the dataset similar to each sampled row are also added alongside each of the sampled rows, enabling the LLM to infer context and patterns effectively. **Code example**: the latest and most effective code snippet generated. Figure 3 provides an example of the prompt structure.

3.5 Sample data for the Prompt

The system employs an unsupervised clustering approach to select diverse rows of the dataset for inclusion in the prompt. Given a training dataset split $\tilde{D}_{train}$ (containing only the relevant columns), the process involves the following steps.

For each row $r \in \tilde{D}_{train}$, an embedding $\mathbf{e}_r$ is computed as described in Section 3.3. The set of embeddings $\{\mathbf{e}_r\}$ is partitioned into k clusters using k-means clustering. Each cluster is represented by its centroid $\mathbf{c}_i$ ($i \in \{1, 2, \ldots, k\}$). For each cluster $\mathcal{C}_i$, the row embedding $\mathbf{e}_{r^*}$ closest to the centroid $\mathbf{c}_i$ is selected as the representative sample:

$$r^* = \arg \min_{\mathbf{e}_r \in \mathcal{C}_i} \|\mathbf{e}_r - \mathbf{c}_i\|^2.$$

The corresponding row r^* is then included in the prompt as the sample data. For exemplar-based tasks, along with each row r^*, a set of rows similar to r^* (based on semantic similarity of embeddings) from $\tilde{D}_{train}$ is included as additional examples in the prompt. The resulting set of representative rows and additional examples ensures semantic diversity and relevance for the sample data while effectively covering the training data.

3.6 Iterative Prompt Optimization

The system employs an iterative approach (Wang et al., 2022) for both memory-dependent and memory-independent modules. Algorithm 1 outlines the process of optimizing prompts iteratively, incorporating the best-performing code snippets from previous iterations. At each iteration, prompts are built using different chunks of the train set, and multiple outputs are sampled, filtering out non-executable and low-accuracy code.

This approach tackles two challenges: first, the iterative process incorporates diverse data samples, generating a set of code snippets that collectively capture various patterns. This eases the need for a single perfect snippet. Second, by including the best-performing snippet from previous iterations, the prompt is incrementally refined, improving code quality and task alignment.

```
def task(input_dict):
    product_name = input_dict['product_name']
    if 'Women Wedges' in product_name:
        return ["Footwear >> Women's Footwear >> Wedges"]
    elif 'Ring' in product_name:
        return ["Jewellery >> Rings"]
    elif 'iPad' in product_name:
        return ["Mobiles & Accessories >> Tablet Accessories >> Cases & Covers >>
DailyObjects Cases & Covers"]
    elif 'Bangles' in product_name:
        return ["JeweLllery >> Bangles, Bracelets & Armlets >> Bracelets"]
    elif 'Mug' in product_name:
        return ["Kitchen & Dining >> Coffee Mugs >> Rockmantra Coffee Mugs" ]
   elif 'Towel' in product_name:
        return ["Home Furnishing >> Bath Linen >> Towels"]
    elif 'Apple iPad Air' in product_name:
        return ["Mobiles & Accessories >> Tablet Accessories >> Cases & Covers >> Cases &
Covers"]
    elif 'Bra' in product_name:
        return ["Clothing >> Women's Clothing >> Lingerie, Sleep & Swimwear >> Bras >> Q-
rious Bras" ]
    elif 'Router' in product_name:
        return ["Computers >> Network Components >> Routers >> Aeoss Routers"]
    else:
        return "Unknown"
```

Figure 4: Example of code generated for complex data pattern, for imputing product_category_tree

3.7 Utilizing multiple code snippets

The system generates multiple code snippets, each independently applied to the dataset. The outputs from these snippets are evaluated for each row, and a majority voting approach is employed to determine the final output value for that row. This approach enhances our solution's robustness by bringing consensus among generated code snippets, thereby mitigating the risk of individual code snippets producing erroneous outputs and improving the overall reliability and accuracy of the data-wrangling process.

4 Experiments

We evaluated the `CodeGenWrangler` system through controlled experiments, comparing it to two baselines. The first baseline used a row-wise LLM approach for missing value imputation, error detection, and correction, following the method described in (Ashlesha et al., 2024). This approach involves a row-wise application of LLMs. The second baseline replicated the (Li and Döhmen, 2024) system without external memory (*memory-independent module*), as outlined in (Li and Döhmen, 2024), which operates without leveraging an external knowledge base, distinguishing it from our proposed system.

To ensure a rigorous comparison, we employed three distinct LLM models across the experimental setups. The *row-wise* LLM baseline leveraged results derived from `flan-t5-xxl` and `mixtral-8x7b` models, in alignment with the results reported in (Ashlesha et al., 2024). In contrast, the `CodeGenWrangler` system, both with and without external memory, utilized state-of-the-art code models, `codellama-34b-instruct` (Roziere et al., 2023) and `deepseek-coder-33b-instruct` (Guo et al., 2024), selected for their relevance in handling code generation tasks. Crucially, to guarantee the validity and fairness of the evaluation, all setups incorporated `llama-3.1-70b-instruct` as a common baseline model, controlling for architectural and computational differences across the experimental conditions.

For imputation and error detection, we used datasets from (Ashlesha et al., 2024). For error correction, 50% of the target column values were swapped with entries from other rows to simulate realistic errors.

5 Results and Analysis

We compared performance between `CodeGenWrangler` and baselines on various datasets across DI, ED and EC tasks. In Table 1, we report results on datasets which reveal some key insights. Complete results for all datasets can be found in Appendix A.1. Broadly, we make the following observations:

Effectively incorporating external data improves performance on knowledge-dependent tasks: Utilizing external memory to improve performance and consistency by providing a reliable

Task	Dataset	Target Column	Row-level			CGW with memory			CGW w/o memory		
			flan-t5-xxl	mixtral-8x7b	llama-3.1-70b	codellama-34b	deepseek-coder-33b	llama-3.1-70b	codellama-34b	deepseek-coder-33b	llama-3.1-70b
DI	Airline	Country Name	0.97	**0.99**	0.46	0.98	0.98	**0.99**	0.67	0.66	0.67
	Airline	Airport Continent	**1.00**	**1.00**	0.81	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	Airline	Airport Country Code	0.90	**1.00**	0.62	0.98	0.99	0.99	0.70	0.67	0.77
	fortune1000_2023	Gained_in_Rank	0.93	0.93	0.67	0.97	**0.98**	**0.98**	0.92	0.92	**0.98**
	fortune1000_2023	Dropped_in_Rank	0.94	0.91	0.77	0.98	**0.97**	0.94	0.94	0.94	0.95
	flipkart_com-ecommerce_sample	product_category_tree	0.48	0.31	0.06	**0.59**	0.30	0.49	**0.57**	0.30	0.49
	starbucks_in_california	24_hour_service	0.76	0.79	0.00	0.92	0.50	**1.00**	0.92	0.65	0.96
	finance_sentiment_analysis	Sentiment	0.51	**0.70**	0.69	0.41	0.40	0.57	0.39	0.40	0.57
ED	fortune1000_2023	Industry	0.77	**0.96**	0.90	0.63	0.63	0.62	0.62	0.62	0.63
	fortune1000_2023	Sector	0.39	**0.99**	0.85	0.54	0.54	0.55	0.53	0.51	0.55
	shopping_trends	Season	**0.95**	0.96	0.85	0.55	0.55	0.54	0.55	0.55	0.55
	starbucks_in_california	24_hour_service	0.93	0.99	0.93	0.94	0.56	**1.00**	0.99	0.99	**1.00**
	Airline	Airport Country Code	0.91	**0.99**	**0.99**	**0.99**	0.98	**0.99**	0.89	0.88	0.77
EC	Airline	Airport Continent	**1.00**	**1.00**	0.80	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	Airline	Airport Country Code	0.89	**1.00**	0.65	0.99	0.97	0.97	0.99	0.66	0.90
	Airline	Country Name	0.97	**0.99**	0.49	0.98	0.98	0.98	0.98	0.66	0.63
	flipkart_com-ecommerce_sample	product_category_tree	0.04	0.04	0.06	0.22	**0.55**	0.50	0.24	0.53	0.54
	fortune1000_2023	Dropped_in_Rank	0.92	0.49	0.75	**0.99**	0.98	**0.99**	0.98	0.92	**0.99**
	fortune1000_2023	Gained_in_Rank	0.94	0.91	0.63	0.98	0.94	**0.99**	0.97	0.94	**0.99**

Table 1: Comparison between `CodeGenWrangler` (CGW with memory) and baselines on Missing Data Imputation (DI), Error Detection (ED) and Error Correction (EC). For DI and EC, accuracy is reported. For ED, F1-macro is reported.

Algorithm 1 Iterative Prompt Optimization

```
Require: $\tilde{D}$: Dataset with relevant columns, $c_T$: Target column, $task \in \{\text{DI}, \text{ED}, \text{EC}\}$, $\mathcal{T}$: External relevant tables, LLM, $s$: Number of samples, $v$: Validation interval
Ensure: Optimized set of source code snippets $code\_snippets$ to perform $task$
 1: $\tilde{D}_{train}, \tilde{D}_{val} \leftarrow$ split($\tilde{D}$)
 2: $code\_snippets \leftarrow \{\}$
 3: $best\_accuracy \leftarrow 0$, $best\_snippet \leftarrow None$
 4: for $i = 1$ to $num\_chunks$ do
 5:     $\tilde{D}^i_{train} \leftarrow$ Obtain chunk of $\tilde{D}_{train}$
 6:     $prompt \leftarrow \langle \tilde{D}^i_{train}, c_T, task, \mathcal{T}, best\_snippet \rangle$ ▷ (as per Section 3.4)
 7:     $snippets \leftarrow$ Execute LLM($prompt$) $s$ times
 8:     Filter $snippets$ for executable functions
 9:     $outputs \leftarrow$ Apply $snippets$ to $\tilde{D}^i_{train}$
10:     $accuracies \leftarrow$ Compare $outputs$ with $\tilde{D}^i_{train}[c_T]$
11:     $valid\_snippets \leftarrow snippets$ with $accuracies > 0$
12:     Update $best\_snippet$, $best\_accuracy$
13:     Append $valid\_snippets$ to $code\_snippets$
14:     if $i$ mod $v = 0$ then                    ▷ Periodic validation
15:         $val\_outputs \leftarrow$ Apply $code\_snippets$ to $\tilde{D}_{val}$
16:         $voted\_outputs \leftarrow$ Majority vote of $val\_outputs$
17:         $val\_accuracies \leftarrow$ Compare $voted\_outputs$ with $\tilde{D}_{val}[c_T]$
18:         if $val\_accuracies > 0.9$ then
19:             return $code\_snippets$
20:         end if
21:     end if
22: end for
23: return $code\_snippets$
```

and up-to-date knowledge base, which is particularly evident for tasks like DI and EC in the `Airline` dataset. `CodeGenWrangler` efficiently uses a reference table of country and continent codes, outperforming row-level baselines and variants relying solely on LLMs' internal knowledge, which are prone to errors from hallucinations.

Code generation is an effective strategy when the data pattern can be expressed in exact logical terms: The generated code outperforms the row-level baseline by applying precise logic, such as comparing `opening_time` and `closing_time` for `24_hours_service` in the Starbucks dataset or using the `Change_in_rank` sign to impute `Gained_in_rank` and `Dropped_in_rank` in the `fortune1000_2023` dataset. In these cases, the row-level baseline underperforms as it lacks the ability to apply precise logical decision-making and must rely on the LLM's ability to generalize from a limited number of in-context examples.

Generating code based on diverse data samples effectively captures complex patterns: For the row-level baseline, models rely on a small set of in-context examples, which may not be sufficient when the data pattern is complex (e.g., determining the product category taxonomy from the name alone in the `product_category_tree` column of the `flipkart_ecommerce` dataset as shown in Figure 4). By generating multiple code snippets over a diverse set of samples, the code captures information across the dataset and distills it into concise heuristics that better represent the pattern. Although these heuristics may not guarantee perfect accuracy, this approach significantly outperforms the row-level baseline.

Code generation is less effective on Error Detection tasks: `CodeGenWrangler` competes well in DI and EC but struggles with ED due to the variety of errors, like syntactic anomalies or semantic mismatches. Such errors are difficult to capture using concise code snippets. It performs poorly on tasks like the `Industry` column in `fortune1000_2023` but excels when errors can be captured via logical rules (e.g., `24_hours_service` in `starbucks`) or verified with external knowledge (e.g., `Airport Continent / Country Code` in `Airlines`). We observe that the code generation is not very effective for datasets that need deep semantic understanding or probabilistic reasoning or those which do not follow clear logical patterns.

The number of LLM calls required by *row-level* method is proportional to the number of dataset rows. In contrast, our system reduces the number of LLM calls by a factor of 10 approximately compared to *row-level* method (see Figure 5).

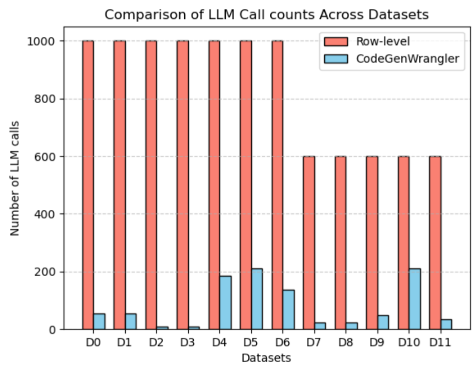


Figure 5: Number of LLM calls (`Llama-3.1-70b-instruct`) required for DI task across 12 datasets: D0-D3 (Airline), D4-D6 (Customer Support), and D7-D11 (Fortune 1000).

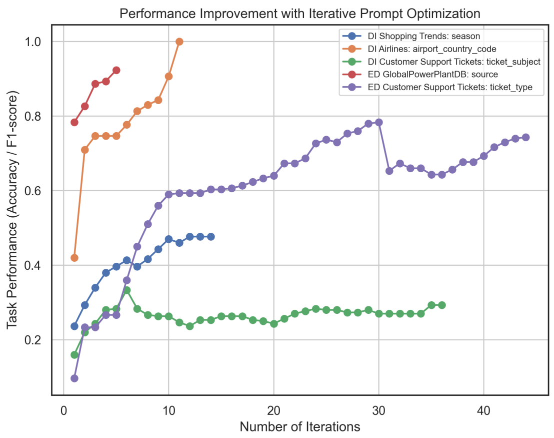


Figure 6: Gain with *Iterative Prompt Optimization*. (Legend format <Task Dataset: Target-col>)

Figure 6 demonstrates the effectiveness of Iterative Prompt Optimization, where refining prompts with the best-performing code improves alignment. High semantic complexity datasets like `Ticket Type` and `Ticket Subject` required up to 40 iterations, while simpler datasets like `Airport Country Code` converged in fewer than 10.

6 Conclusion and Future Work

We proposed a system to perform data wrangling on tabular datasets using code-generating LLMs. Our system generates source code by encoding rules that capture the logical patterns in the datasets. It generates multiple task-specific code snippets for each data pattern and chooses the best code snippet via a majority vote for higher reliability. The generated code snippets are executed to carry out data-wrangling tasks to replace the expensive row-wise LLM inference calls by the state-of-the-art approaches for scaling to large datasets. Our system can also handle memory-dependent tasks that require task-specific additional context provided as external domain knowledge. It adopts an iterative prompt refinement strategy to optimize the generated code for accuracy and efficiency. We plan to extend our approach for its applicability to other downstream tasks and different language models. We plan to evaluate the performance of the system on more realistic noisy and incomplete datasets.

References

Alice. 2017. Starbucks Dataset. https://data.world/alice-c/starbucks/workspace/file?filename=Starbucks+in+California.csv. Accessed on 3-Oct-2024.

Akella Ashlesha, Abhijit Manatkar, Brij Chavda, and Hima Patel. 2024. An automatic prompt generation system for tabular data tasks. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL): Human Language Technologies (Industry Track)*, pages 191–200.

Akella Ashlesha and Krishnasuri Narayanam. 2025. Data Wrangling task automation using Code-Generating Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI).*

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 7(1):85–111.

Cricsheet. 2023. IPL Matches. https://www.kaggle.com/datasets/patrickb1912/ipl-complete-dataset-20082020?select=matches.csv. Accessed on 3-Oct-2024.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 33(7):1–38.

Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481.

Forbes. 2019. Fortune 1000. https://www.kaggle.com/datasets/agailloty/fortune1000. Accessed on 3-Oct-2024.

Yongshun Gong, Zhibin Li, Jian Zhang, Wei Liu, Yilong Yin, and Yu Zheng. 2021. Missing value imputation for multi-view urban statistical data via spatial correlation learning. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 35(1):686–698.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian Mcauley. 2023. LongCoder: A long-range pretrained language model for code completion. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, pages 12098–12107.

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*.

Aleksei Guryanov. 2019. Histogram-based algorithm for building gradient boosting ensembles of piecewise linear decision trees. In *Proceedings of the 8th International Conference on Analysis of Images, Social Networks and Texts (AIST)*, pages 39–50.

Buliao Huang, Yunhui Zhu, Muhammad Usman, and Huanhuan Chen. 2024. Semi-supervised learning with missing values imputation. *Knowledge-Based Systems (KBS)*, 284:111171.

Joon Suk Huh, Changho Shin, and Elina Choi. 2023. Pool-search-demonstrate: Improving data-wrangling llms via better in-context examples. In *NeurIPS 2023 Second Table Representation Learning Workshop*.

Hiroshi Iida, Dung Thai, Varun Manjunatha, and Mohit Iyyer. 2021. TABBIE: Pretrained representations of tabular data. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL): Human Language Technologies*, pages 3446–3456.

Gonzalo Jaimovitch-López, Cèsar Ferri, José Hernández-Orallo, Fernando Martínez-Plumed, and María José Ramírez-Quintana. 2023. Can language models automate data wrangling? *Machine Learning (ML)*, 112(6):2053–2082.

Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI)*, pages 1–19.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 37, pages 5131–5140.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 9459–9474.

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

Wei-Chao Lin, Chih-Fong Tsai, and Jia Rong Zhong. 2022. Deep learning for missing value imputation of continuous data and the effect of data discretization. *Knowledge-Based Systems (KBS)*, 239:108079.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023a. Pretrain, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35.

Shang-Ching Liu, ShengKun Wang, Tsungyao Chang, Wenqi Lin, Chung-Wei Hsiung, Yi-Chen Hsieh, Yu-Ping Cheng, Sian-Hong Luo, and Jianwei Zhang. 2023b. Jarvix: A llm no code platform for tabular data analysis and optimization. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP): Industry Track*, pages 622–630.

Yilun Liu, Shimin Tao, Xiaofeng Zhao, Ming Zhu, Wenbing Ma, Junhao Zhu, Chang Su, Yutai Hou, Miao Zhang, Min Zhang, et al. 2024. Coachlm: Automatic instruction revisions improve the data quality in llm instruction tuning. In *Proceedings of the IEEE 40th International Conference on Data Engineering (ICDE)*, pages 5184–5197.

Yinan Mei, Shaoxu Song, Chenguang Fang, Haifeng Yang, Jingyun Fang, and Jiang Long. 2021. Capturing semantics for imputation with pre-trained language models. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 61–72.

Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? *Proceedings of the VLDB Endowment (PVLDB)*, 16(4):738–746.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Manar D Samad, Sakib Abrar, and Norou Diawara. 2022. Missing value estimation using clustering and deep learning within multiple imputation framework. *Knowledge-Based Systems (KBS)*, 249:108968.

Tressy Thomas and Enayat Rajabi. 2021. A systematic review of machine learning-based missing value imputation techniques. *Data Technologies and Applications (DTA)*, 55(4):558–585.

Stef Van Buuren. 2018. *Flexible imputation of missing data*. CRC press.

Boshi Wang, Xiang Deng, and Huan Sun. 2022. Iteratively prompt pre-trained language models for chain of thought. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2714–2730.

Jianxun Wang and Yixiang Chen. 2023. A review on code generation with llms: Application and evaluation. In *Proceedings of the 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289. IEEE.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1069–1088.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large language models meet nl2code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 7443–7464.

Jialu Zhang, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2024. Pydex: Repairing bugs in introductory python assignments using llms. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 8(1):1100–1124.

A Appendix

A.1 Comprehensive Evaluation

The full results in Table 2 present the performance of the proposed system on Error Detection and Missing Value Imputation tasks across 21 datasets each. These datasets, sourced from (Ashlesha et al., 2024), originate from publicly available repositories such as Kaggle and OpenML, ensuring a diverse range of real-world data patterns. The results highlight the system's effectiveness in handling various data complexities, demonstrating consistent performance across multiple datasets and validating its adaptability to different data quality tasks.

B Example Code snippets

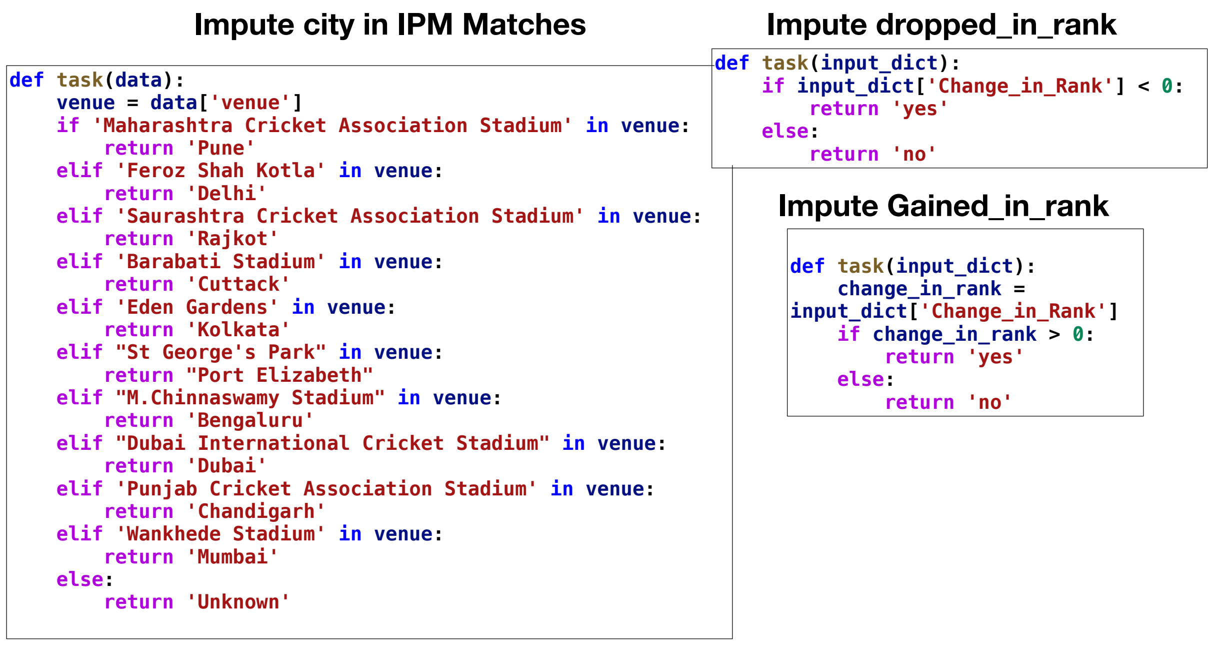
This section presents the example code snippets, which illustrate different task-specific codes generated for different datasets. These snippets highlight the adaptability of our approach in capturing diverse data patterns effectively. For example, Figure 7 shows the code generated by our system on two different datasets, imputing the city column in the IPM Matches dataset (Cricsheet, 2023), and imputing the Fortune 1000 dataset (Forbes, 2019). The code generated by the system is used for two tasks: (i) imputing the '24_hour_service' column in the 'Starbucks' dataset (Alice, 2017), and (ii) imputing the 'city' column in the 'Restaurant' dataset (Mei et al., 2021). These are shown in Figure 8.

C Example Prompts for Diverse Datasets

This section presents example prompts which are automatically constructed by the system for memory dependent tasks 9 and memory independent task 10. These prompts are designed to incorporate relevant data patterns, external knowledge (when applicable), and iterative refinements to enhance the quality of generated code snippets.

Task	Dataset (# columns)	Target column	Row-level (flan-t5-xxl)	Row-level (mixtral-8x7b)	Row-level (llama)	cgw (codellama)	cgw (deepseek)	cgw (llama)
DI	Restaurant	City	0.82	0.97	0.75	0.63	0.85	0.92
DI	Airline	Continents	1.00	1.00	0.85	1.00	1.00	1.00
DI	customer support tickets	Ticket Type	0.21	0.20	0.16	0.19	0.20	0.18
DI	customer support tickets	Ticket Priority	0.27	0.25	0.00	0.23	0.58	0.24
DI	customer support tickets	Ticket Subject	0.06	0.05	0.01	0.06	0.07	0.08
DI	fortune1000_2023	HeadquartersState	0.88	0.97	0.96	0.93	0.94	0.91
DI	fortune1000_2023	Sector	0.89	0.87	0.53	0.79	0.63	0.79
DI	fortune1000_2023	Industry	0.23	0.34	0.17	0.32	0.21	0.31
DI	flipkart_com-ecommerce_sample	brand	0.58	0.20	0.63	0.52	0.40	0.38
DI	starbucks_in_california	state	1.00	1.00	0.92	0.99	0.76	0.99
DI	starbucks_in_california	county	1.00	0.99	0.99	1.00	0.69	0.93
DI	starbucks_in_california	city	0.44	0.86	0.73	0.43	0.22	0.45
DI	starbucks_in_california	state	1.00	1.00	0.92	0.99	0.76	0.99
DI	starbucks_in_california	county	1.00	0.99	0.99	1.00	0.69	0.93
DI	starbucks_in_california	city	0.44	0.86	0.73	0.43	0.22	0.45
DI	shopping_trends	Category	1.00	0.99	0.69	0.66	0.93	0.96
DI	shopping_trends	Season	0.28	0.26	0.10	0.23	0.15	0.45
DI	AMTRAK	City	0.98	0.81	0.83	0.92	0.91	0.92
DI	IPM_Matches	city	0.85	0.94	0.94	0.80	0.62	0.73
DI	BigBasketProducts	category	0.92	0.92	0.89	0.73	0.88	0.91
DI	SpeedDating	race	0.61	0.64	0.48	0.45	0.57	0.50
ED	Airline	Country Name	0.96	0.96	0.99	0.99	0.99	0.99
ED	Airline	Airport Continent	0.76	0.99	0.97	1.00	1.00	1.00
ED	Airline	Continents	0.91	0.91	0.99	1.00	1.00	1.00
ED	customer_support_tickets	Ticket Priority	0.93	0.99	0.96	0.54	0.53	0.54
ED	customer_support_tickets	Ticket Subject	0.68	0.98	0.88	0.44	0.43	0.43
ED	customer_support_tickets	Ticket Type	0.81	0.98	0.92	0.53	0.49	0.53
ED	fortune1000_2023	Dropped_in_Rank	0.86	1.00	0.98	0.98	0.97	0.99
ED	fortune1000_2023	Gained_in_Rank	0.81	0.99	0.99	0.97	0.98	0.98
ED	BigbasketProducts	category	0.87	0.95	0.98	0.87	0.88	0.92
ED	BigbasketProducts	sub_category	0.48	0.45	0.86	0.44	0.34	0.43
ED	BigbasketProducts	type	0.86	0.88	0.84	0.50	0.56	0.52
ED	finance_sentiment_analysis	Sentiment	0.37	0.97	0.88	0.71	0.73	0.72
ED	flipkart_com-ecommerce_sample	brand	0.84	0.87	0.96	0.54	0.45	0.46
ED	flipkart_com-ecommerce_sample	product_category_tree	0.79	0.70	0.86	0.63	0.49	0.52
ED	GlobalPowerPlantDB	country_long	0.87	0.98	1.00	1.00	0.97	1.00
ED	IPM_Matches	city	0.88	0.91	0.87	0.91	0.76	0.79
ED	SpeedDating	race	0.69	0.53	0.99	0.70	0.69	0.78
ED	shopping_trends	Category	0.87	0.95	1.00	0.98	0.91	0.98
ED	starbucks_in_california	city	0.79	0.95	0.97	0.93	0.93	0.94
ED	starbucks_in_california	county	0.53	0.98	0.94	0.99	0.99	1.00
ED	starbucks_in_california	state	0.89	1.00	0.97	0.91	0.91	1.00

Table 2: Results for extended datasets on Data Imputation and Error Detection tasks



Impute city in IPM Matches

```
def task(data):
    venue = data['venue']
    if 'Maharashtra Cricket Association Stadium' in venue:
        return 'Pune'
    elif 'Feroz Shah Kotla' in venue:
        return 'Delhi'
    elif 'Saurashtra Cricket Association Stadium' in venue:
        return 'Rajkot'
    elif 'Barabati Stadium' in venue:
        return 'Cuttack'
    elif 'Eden Gardens' in venue:
        return 'Kolkata'
    elif "St George's Park" in venue:
        return "Port Elizabeth"
    elif "M.Chinnaswamy Stadium" in venue:
        return 'Bengaluru'
    elif "Dubai International Cricket Stadium" in venue:
        return 'Dubai'
    elif 'Punjab Cricket Association Stadium' in venue:
        return 'Chandigarh'
    elif 'Wankhede Stadium' in venue:
        return 'Mumbai'
    else:
        return 'Unknown'
```

Impute dropped_in_rank

```
def task(input_dict):
    if input_dict['Change_in_Rank'] < 0:
        return 'yes'
    else:
        return 'no'
```

Impute Gained_in_rank

```
def task(input_dict):
    change_in_rank =
input_dict['Change_in_Rank']
    if change_in_rank > 0:
        return 'yes'
    else:
        return 'no'
```

Figure 7: Code Generated by our system 1. to impute ‘city’ column in ‘IPM Matches’ dataset (Cricsheet, 2023). 2. to impute ‘Gained in rank’ and ‘Dropped in Rank’ columns in ‘Fortune 1000’ dataset (Forbes, 2019)

Starbucks dataset impute 24_hour_service

```python
def task(inputs):
    regular_hours = inputs.get('regular
                hours')
    saturday_opening_times =
        inputs.get('saturday opening times')
    sunday_opening_times =
        inputs.get('sunday opening times')

    if regular_hours == 'nan' or
        saturday_opening_times == 'nan' or
        sunday_opening_times == 'nan':
         return False

    if regular_hours == '12:00 AM to 12:00
AM'
        and saturday_opening_times == '12:00
        AM to 12:00 AM'
        and sunday_opening_times == '12:00 AM
        to 12:00 AM':
         return True
    return False
```

Restaurant dataset impute city

```python
def task(data):
    if data['phone'].startswith('415'):
        return 'san francisco'
    elif data['phone'].startswith('404'):
        return 'atlanta'
    elif data['phone'].startswith('213'):
        return 'los angeles'
    elif data['phone'].startswith('212'):
        return 'new york'
    elif data['phone'].startswith('718'):
        return 'queens'
    else:
        return 'Unknown'
```

Figure 8: Code Generated by the system: (i) To impute '24_hour_service' column in 'Starbucks' dataset (Alice, 2017) (ii) To impute 'city' column in 'Restaurant' dataset (Mei et al., 2021)

Task instructions:

```
Given the 2 Tables, Table 1 and Reference Table write a python code to answer the following question.
1. Table1 where the question is to be answered.
2. Reference Table is the reference table which helps to answer the question.
3. Write a python code which takes row dictionary of Table 1 and Table 2 as pandas dataframe input.
4. Make sure to use try and exception for exception handling.
5. In case of excpetion return "Unknown"
6. example function
def task(input_dict, reference_table):
# input_dict is a dictionary
# reference table is a pandas dataframe
7. User the correct column names from Table 1 and Reference Table
8. Check for string matches, use string functions such as śtartswithór éndswithór ińfor better pattern match.
```

Table 1:

```
|  HeadquartersCity  |  HeadquartersState  |
|:------------------:|:-------------------:|
|    Redwood City    |         CA          |
|      Chicago       |         IL          |
|     Arlington      |         VA          |
|      Houston       |         TX          |
```

External Knowledge (reference table):

```
 |           name            |  alpha-2  |  alpha-3  |  region  |         sub-region         |
|:-------------------------:|:---------:|:---------:|:--------:|:-------------------------:|
|         Hong Kong         |    HK     |    HKG    |   Asia   |       Eastern Asia        |
|  Cocos (Keeling) Islands  |    CC     |    CCK    | Oceania  | Australia and New Zealand |
|          Czechia          |    CZ     |    CZE    |  Europe  |      Eastern Europe       |
| Saint Pierre and Miquelon |    PM     |    SPM    | Americas |     Northern America      |
|         Viet Nam          |    VN     |    VNM    |   Asia   |    South-eastern Asia     |
|         Holy See          |    VA     |    VAT    |  Europe  |      Southern Europe      |
|         Hong Kong         |    HK     |    HKG    |   Asia   |       Eastern Asia        |
```

Task instructions:

```
#input_dict contains HeadquarterCity , HeadquarterState
# reference table contains name,alpha2,alpha3,region,subregion
# the code uses input_dict and table2 to output the HeadquartersState

What is the value of HeadquartersState

CODE:
```

Figure 9: Example prompt for Memory Dependent task.

Task instructions:

Instructions:

Given a series of examples, your task is to identify the pattern between the input columns and their corresponding output values. Write a Python function named t̀askt̀hat accurately maps the input data to the correct output based on this identified pattern.

Key Guidelines:

1. **Analyze Patterns:**
Observe the logical relationships and string patterns within the input data. Focus on identifying consistent connections between input columns and their corresponding outputs.
2. **Optimize the Logic:**
- Use **regular expressions** when necessary to match specific conditions or patterns efficiently.
- Employ methods like s̀tartswithànd èndswithìnstead of generic comparisons for precise string matching.
3. **Comprehensive Coverage:**
- Ensure your code considers **all possible patterns and conditions** given in the test examples.
- Write as many ifs̀tatements as required to handle each identified pattern thoroughly.
4. **Relevant Features Only:**
Utilize only the columns that show a consistent relationship to the output. Avoid introducing unnecessary complexity by including irrelevant columns.
5. **Default Behavior:**
If no recognizable pattern is found, the function should return ̋Unknown":

Table 1:

name	county	city
Pacific & Yokuts - Stockton	San Joaquin County	Stockton - San Joaquin County
Washington & Culver	Los Angeles County	Culver City - Los Angeles County
Albertsons - Temecula #6734	Riverside County	Murrieta - Riverside County
Bouquet Canyon & Newhall Ranch, San	Los Angeles County	Santa Clarita - Los Angeles County

Task instructions:

```
#input_dict contains name, county
What is the value of city

CODE:
```

Figure 10: Example prompt for Memory Independent task.