Can an LLM find its way around a Spreadsheet?

Cho-Ting Lee
Virginia Tech
Arlington, VA, USA
choting@vt.edu

Andrew Neeser
Virginia Tech
Blacksburg, VA, USA
aneeser24@vt.edu

Shengzhe Xu
Virginia Tech
Arlington, VA, USA
shengzx@vt.edu

Jay Katyan
Virginia Tech
Blacksburg, VA, USA
jkatyan@vt.edu

Patrick Cross
Virginia Tech
Blacksburg, VA, USA
patrickcross7@vt.edu

Sharanya Pathakota
Virginia Tech
Blacksburg, VA, USA
sharanyap21@vt.edu

Marigold Norman
World Forest ID
Washington DC, USA
marigold.norman@worldforestid.org

John Simeone
Simeone Consulting
Littleton, NH, USA
simeoneconsulting@gmail.com

Jaganmohan Chandrasekaran
Virginia Tech
Arlington, VA, USA
jagan@vt.edu

Naren Ramakrishnan
Virginia Tech
Arlington, VA, USA
naren@cs.vt.edu

***Abstract*—Spreadsheets are routinely used in business and scientific contexts, and one of the most vexing challenges is performing data cleaning prior to analysis and evaluation. The ad-hoc and arbitrary nature of data cleaning problems, such as typos, inconsistent formatting, missing values, and a lack of standardization, often creates the need for highly specialized pipelines. We ask whether an LLM can find its way around a spreadsheet and how to support end-users in taking their free-form data processing requests to fruition. Just like RAG retrieves context to answer users' queries, we demonstrate how we can retrieve elements from a code library to compose data preprocessing pipelines. Through comprehensive experiments, we demonstrate the quality of our system and how it is able to continuously augment its vocabulary by saving new codes and pipelines back to the code library for future retrieval.**

***Index Terms*—LLMs, code generation, data cleaning, end-user programming**

I. Introduction

Pre-trained large language models (LLMs) have demonstrated significant proficiency in generating code from natural language prompts, heralding a new era in software development [1, 2, 3, 4]. In addition to generating code, modern integrated development environments (IDEs) also incorporate LLMs to assist with error correction, code refactoring, and multilingual programming knowledge. By engaging in English conversations, LLMs can function as coding assistants, enabling users with limited proficiency to produce accurate and executable code to accomplish their tasks [5, 6].

Nevertheless, LLMs have not yet attained the maturity to address all or most challenges programmers face in software development and machine learning [7, 8]. One of the vexing aspects for programmers involves processing tabular data, e.g., in the form of spreadsheets [9, 10]. Such datasets (Fig. 1 for example) frequently suffer from issues like typographical errors, inconsistent formatting, missing values, and lack of

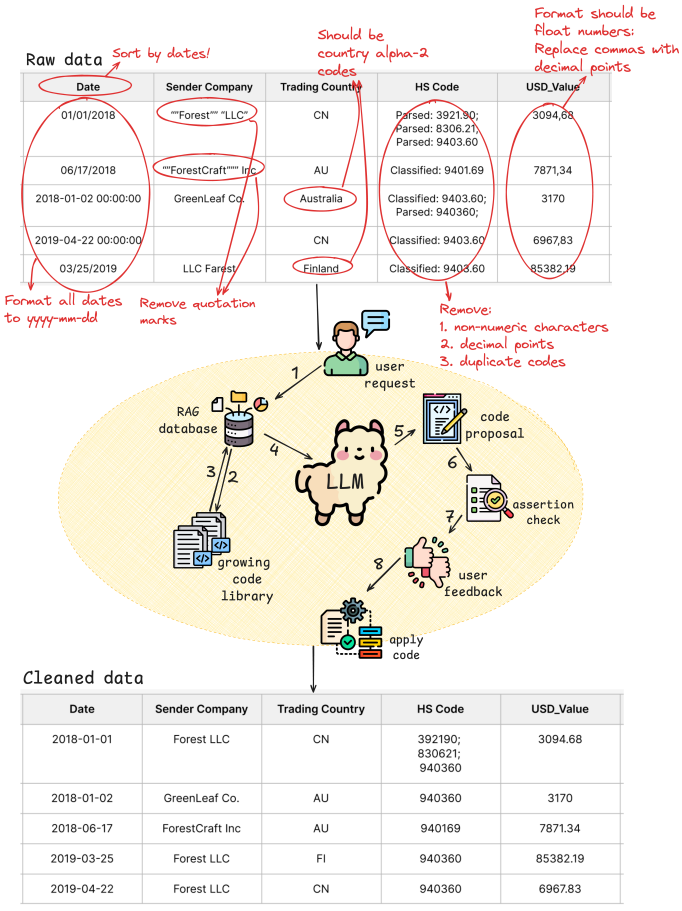

Fig. 1. Missing values, inconsistent formatting, misspellings, and other similar issues are frequently encountered in large spreadsheets. Our approach, TradeSweep, proposes the use of an LLM to systematize data cleaning and transformations.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00101

standardization, necessitating the creation of highly specialized pipeline [11, 12].

Data cleaning and data preprocessing is not merely a matter of resolving inconsistencies; it can significantly impact downstream results [13, 14]. For example, correcting typographical errors in a dataset may inadvertently lead to over-clustering or under-clustering of the relevant column, while inaccurately resolving entities can distort the true distribution of data points. Therefore, these processes must be approached with care and deliberation, with careful attention to sequential transformations. The need for automation support is widely acknowledged [15, 16, 17], particularly for a "human-in-the-loop" pipeline to guide the cleaning process and prevent spurious results.

We present TradeSweep (named for its focus on tabular trade datasets), an approach to use LLMs to systematically transduce spreadsheets. TradeSweep interprets English requests for data preprocessing and generates code proposals that can be composed and applied to targeted datasets, achieving high performance. Similar to how Retrieval-Augmented Generation (RAG) retrieves context to answer users' queries, we demonstrate how elements from a code library can be stored and retrieved to compose complex pipelines.

The contributions of this paper are as follows:

- TradeSweep utilizes English conversations to understand and respond to users' requests with Python code encompassing preprocessing functions, supporting three main capabilities:
  1) It produces new code when requested for a completely new task.
  2) It can precisely modify its proposed code based on users' feedback in English conversations.
  3) The proposed code is automatically tailored to the target data, including accurate column names and suitable algorithms.
- TradeSweep develops and continuously expands a library of fundamental data preprocessing codes by storing executable functions that have been successfully deployed previously. The augmentation of the code library supports the composition and creation of elaborate data pipelines.
- To incorporate feedback from users who lack programming expertise, TradeSweep offers both code proposals and execution results on example data. This allows users to determine whether to accept TradeSweep 's proposal or to make modifications based on output data visualizations, rather than focusing on code specifics.
- We perform extensive experiments on three trade datasets. Results show that TradeSweep is capable of generating executable and efficient code for data preprocessing, significantly reducing time and effort for data analysts.

II. Related Work

A. Generating Formulas and SQL queries

Formulas and SQL queries are the lingua franca of spreadsheets. "Formula Language Model for Excel" (FLAME) [18] is a T5-based model trained exclusively on Excel formulas and used for code generation tasks such as formula repair, formula completion, and similarity-based formula retrieval. (TradeSweep is designed to address various formats of tabular data, including Excel and CSV.)

A significant amount of research focuses on converting natural language questions into executable SQL queries (often referred to as Text-to-SQL) [19, 20, 21, 22]. Most existing benchmarks primarily focus on small databases, which do not accurately reflect the challenges of working with large databases in real-world situations. BIRD (Big Bench for Large-scale Databases) [23] is a Text-to-SQL benchmark designed to narrow the gap between experimental and practical scenarios.

To generate high-level programs, Polozov et al. [24] developed "FlashMeta" to streamline the development of program synthesis tools across various domains by offering a reusable, domain-agnostic set of components, making it adaptable to tasks such as data extraction and UI programming. Similarly, Gulwani et al. [25] employed a programming-by-example approach to help users perform transformations by providing example input-output pairs, which the system generalizes into reusable transformation rules. These contributions underscore the growing accessibility of data manipulation and transformation functions for both spreadsheet and database users.

B. LLMs with Information Retrieval

While LLMs are highly effective when fine-tuned for particular NLP tasks, their ability to access and manipulate knowledge is intrinsically constrained [26, 27, 28, 29]. RAG [30] has achieved great success by integrating retrieval and generation techniques, leading to the development of various related approaches and variations. This line of work has facilitated the convergence between information retrieval and information generation.

Language models have shown substantial improvements in code completion tasks by learning from "internal" source code contexts [31, 32, 33]. Several retrieval-augmented frameworks have been proposed, e.g., Retrieval-Augmented Code Completion framework (ReACC) [34], RedCoder [35], and RepoCoder [36]. These systems however are intended to support developers rather than end-users.

C. LLMs with Data Interaction

Tian et al. proposed SpreadsheetLLM [37], a framework designed to enable large language models (LLMs) to interpret and analyze spreadsheet data, focusing on creating an effective encoding approach that leverages LLMs' powerful comprehension and reasoning capabilities. Expanding the application of LLMs for spreadsheet reasoning and manipulation, Yibin et al. developed SheetAgent [38], which enables users to perform a broad range of tasks through natural language interactions.

D. LLMs and Code Generation

Li et al. developed SkCoder [39], a sketch-based code generation strategy designed to mimic the code reuse behaviors of software engineers. To avoid hallucination [40, 41],

works such as Jigsaw [42], ALGO [43], SWE-agent [44], CodeAgent [45], and CleanAgent [46] implement guardrails to structure code generation outputs. In general, while LLMs have strong capabilities in generating code based on user prompts, they still encounter difficulties in handling algorithmic complexities and often require human verification [47, 48] to ensure correct and/or efficient code. For example, a minor typo in column names can cause the model to generate code that applies to the wrong column, and failing to specify an algorithm in the user's request may result in functions with low efficiency. For instance, this could involve generating code that uses a brute-force approach to sorting numbers instead of a more efficient algorithm like quicksort.

Jacob et al. [49] conducted a thorough research exploring the limitations of current large language models for program synthesis. While LLMs are powerful in generating code, they can struggle with complex logical reasoning and require effective prompting skills and user feedback to maximize performance—a technique we adapted in our experiments.

III. Approach

We present TradeSweep , an LLM-based tool that leverages the LLM's code-writing abilities to produce executable programs for these tasks. Users are only required to provide a dataset in either CSV or Excel format and submit a preprocessing request. As illustrated in Fig. 2, TradeSweep comprises three primary components:

- **Prompt augmentation**: This component employs information retrieval techniques to select the top-k relevant codes from the code library. (Section III-B)
- **Code generation**: The LLM generates a code proposal and visualized samples of execution results, awaiting user feedback. (Section III-C)
- **Code library**: We build a reference document that includes classic Python scripts for data preprocessing tasks. (Section III-D)

A. Problem Definition

Let $\mathcal{D}$ represent a table with n rows and m columns, with column names $c_1, c_2, ..., c_m$ denoted as $\mathcal{C}$. Each element x_{ij}, where $i \in \{1, ..., n\}$ and $j \in \{1, ..., m\}$, represents the value of the j-th feature of the i-th data record. In practical applications, it is often necessary to perform data cleaning (e.g., handling inconsistent formats, outliers, missing values) and preprocessing (e.g., normalization, label encoding) on the dataset $\mathcal{D}$ before training a machine learning model.

Let $\mathcal{A}$ be a subset of $\{c_1, c_2, ..., c_m\}$, representing the collection of columns that require processing. TradeSweep $(\mathcal{M})$ is designed to receive a preliminary description r of the user's data preprocessing requirements in English and to automatically generate Python code $\hat{f}$ and a processed dataset $\hat{\mathcal{D}}$. To achieve this goal, TradeSweep $(\mathcal{M})$ begins by constructing a prompting curriculum denoted as $\mathcal{P} = [r, \mathcal{C}]$. This curriculum is then inputted into an LLM as a prompt to develop Python code that meets the specified requirements r. Using the code generated by TradeSweep (code proposal $\mathcal{G}$), the system executes and evaluates $\mathcal{G}$ on a subset of $\mathcal{D}$ to obtain an execution outcome $\mathcal{O}$ that can be presented to the user. It is important to note that, TradeSweep operates without requiring programming expertise from the user, as the specific columns $\mathcal{A}$ necessitating modification are automatically determined by the model, and the entire process from input description to execution outcome is managed by the system.

One of the commonly reported challenges with using LLMs is its vulnerability to hallucinations. In the context of code generation, this can result in code that includes non-existent functions or incorrect syntax, among other issues. To address hallucination challenges, we expanded the existing curriculum $\mathcal{P} = [r, \mathcal{C}]$ into $\mathcal{P} = [r, \mathcal{C}, \mathcal{F}]$. Here, $\mathcal{F}$ denotes a finite set of closely interconnected fundamental functions that serve as a reference for the LLM. This approach offers a dual advantage:

1) **Decreased Occurrence of Hallucinations:** By incorporating a set of fundamental functions, the likelihood of hallucinations is reduced.
2) **Enhanced Code Generation:** LLMs can develop new code in addition to leveraging the fundamental reference functions.

To implement the expanded curriculum $\mathcal{P} = [r, \mathcal{C}, \mathcal{F}]$, TradeSweep employs an adaptable code library $\mathcal{L}$. First, the request r is transformed into an embedding representation emb_r. A set of k relevant fundamental functions $\mathcal{F} = \{f_1, ..., f_k\}$ is then extracted from the code database $\langle \text{emb}_i, \text{function}_i \rangle$ based on the minimum cosine similarity $\min_{i \in L} \text{cosine}(\text{emb}_r, \text{emb}_i)$. Once a new code proposal $\mathcal{G}$ is successfully developed to meet complex requirements r, typically through interaction with users, it is saved in the library $\mathcal{L}$. This allows the library $\mathcal{L}$ to continually learn and provide more accurate and advanced reference functions in the future.

Furthermore, incorporating human feedback in the code generation process ensures that TradeSweep produces code meeting user expectations. Specifically, we present the code proposal $\mathcal{G}$ along with an execution result $\mathcal{O}$. This allows users to inspect and determine if the outcome meets their expectations. If satisfied, the code is then executed on the entire dataset, enabling users to verify if the processed data aligns with their domain knowledge rather than examining the code in detail. This makes the tool accessible for non-expert programmers. If the intended outcome is not achieved, TradeSweep will initiate a continuous conversation $\mathcal{P} = [r', \mathcal{C}, \mathcal{F}]$, where r' denotes a revision request. In the Results section (Section V), we demonstrate how TradeSweep achieves a high probability of meeting users' requests on the LLM's first attempt and adapts more rapidly than baselines when revisions are requested.

B. Prompt Augmentation

Following the submission of the user's request for data preprocessing, TradeSweep examines all functions contained within the code library to learn from and utilize them as references. The library comprises code functions commonly applied in various preprocessing tasks. Along with these

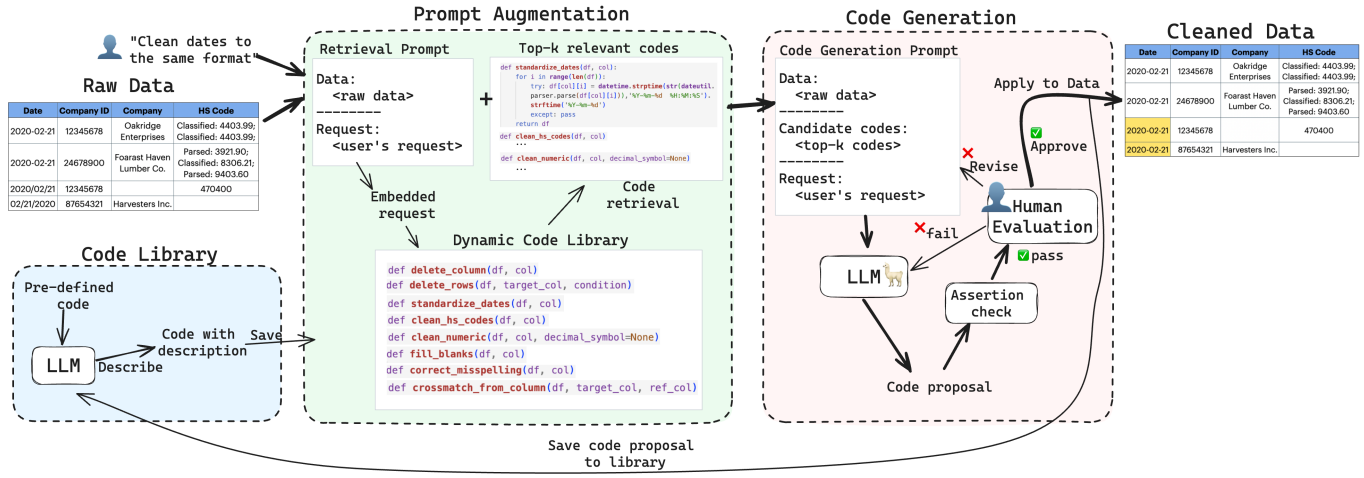

Fig. 2. An overview of TradeSweep with three key components: code retrieval for prompt augmentation, LLM-based code generation followed by human evaluation, and continuous updates to the code library.

functions, data information is also provided to the LLM to aid its understanding of the dataset structure.

However, inputting both code functions and data information can result in a lengthy prompt containing redundant information. This not only reduces the LLM's performance in accurately identifying the most relevant function but also significantly delays the LLM's response time. To address this issue, we aim to shorten the input context. Instead of presenting all code functions in the prompt, we utilize information retrieval (IR) techniques to perform a more precise selection on code functions, providing the LLM with the most relevant and useful codes.

We implemented this approach by representing our code library as a vector database. Each vector includes an embedded description of a code function, with its corresponding function code serving as the payload of the vector. During each round of code generation, we query the vector database to retrieve k code functions deemed most relevant to the user's request. These selected functions are then included in the LLM prompt. This process not only effectively shortens the prompt length and reduces the LLM's response time but also helps the LLM focus on functions most pertinent to fulfilling the user's request (see Fig. 3).

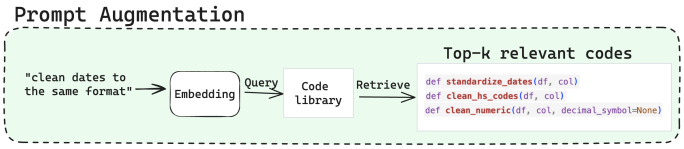

Fig. 3. Identifying relevant code functions for prompt augmentation.

While many state-of-the-art tools allow users to provide <input, output> examples to help the LLM understand the expected result dataset, this approach becomes impractical for real-world datasets with massive record volumes. It is generally impractical for users to define precise pairs of examples for complex data preprocessing tasks. Additionally, providing only a small subset of examples could lead to an overfitting risk, where the LLM might generate code that handles specific cases accurately but fails to generalize across the dataset. Therefore, our approach emphasizes a flexible, prompt-based design, allowing TradeSweep to dynamically interpret natural language requests and generate adaptable code. This design choice focuses on key prompt components, enhancing both the adaptability of code to varied user needs and overall user-friendliness.

C. Code Generation

Once the top-k relevant codes have been retrieved as candidate codes, a prompt for the LLM is created by combining the candidate codes with data information and the user's request. The LLM examines the provided information and produces a code proposal designed to accomplish the requested task. If the LLM does not find any candidate code that aligns with the user's request, it generates a novel code function.

The process of code generation includes iterative enhancements under the following scenarios after the initial code proposal is generated:

1) **Incorrect Code Proposal Format**: We provide the LLM with a response template specifying that the proposal format should include a Python code function that reads a dataframe and returns it at the end. The proposal should also include a function call for applying the code to the data. If either of these components is missing in the generated code, the LLM automatically modifies the proposal to meet the required format.
2) **Execution Error**: Once a code proposal is generated in the expected format, it is tested on sample data. If execution fails due to bugs, syntax errors, exceptions, or other issues, both the code and its corresponding error message are returned to the LLM for revision.

3) **User Feedback**: Users are provided with a visualized execution result on sample data, showing examples of input values and their corresponding output values when the code is applied. Based on the user's review of the code proposal and execution outcomes, they can request code revisions by providing feedback describing necessary fixes to the LLM.

Finally, once the user confirms that both the code proposal and execution examples are correct, the code is applied to the target dataset, as shown in Fig. 4. For novel code or when users request saving multiple functions into a single pipeline, the newly generated code or series of codes are added to our code library.

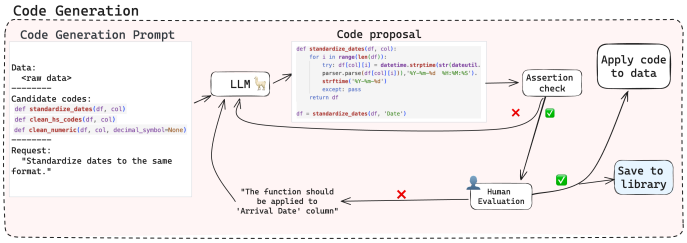

Fig. 4. Code generation and enhancement based on execution results and user feedback.

D. Code Library

Due to the complexity of data preprocessing tasks - such as those requiring specific algorithms or involving multiple datasets - a code library is beneficial as a reference document for the LLM to learn from and follow when generating code proposals. In TradeSweep's code library, each function is designed to handle a particular data preprocessing task and includes comments describing its usage. To enhance the efficacy of TradeSweep, we have developed a code library capable of supporting the addition of new functions as interactions progress:

1) **Novel Code**: When the LLM does not identify any function that corresponds to the user's request after analyzing the retrieved codes, it generates a novel code proposal. Since this newly generated code is not initially included in the library, we incorporate it back into the library to improve efficiency and accuracy for future code generations.
2) **Pipeline Creation Request**: After a series of code functions have been generated, the user may request that the entire procedure be stored as a pipeline. During this process, the several functions previously applied to the dataset are combined into a single code function, which is then added to our code library.

As illustrated in Fig. 5, the process of integrating the novel code function into the code library involves a sequence of steps. First, we use an LLM to generate a description of the novel code, which is then added as comments. In order to save a new function, TradeSweep verifies if it meets three criteria: it must be a valid Python function, include a function description (or generate one if missing), and can read a dataset for manipulation. Subsequently, both the newly generated code function and its associated description are then added into our vector database code library. A new query vector is constructed using the function description, and its corresponding payload is generated by the code.

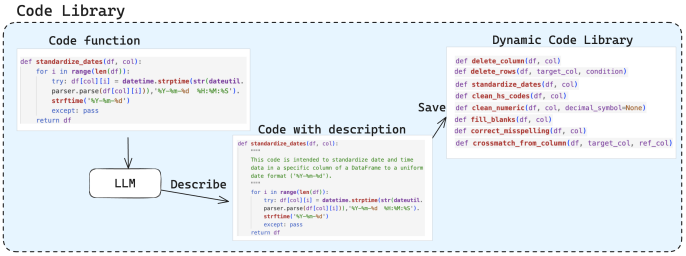

Fig. 5. Code library dynamically updated with each novel LLM-generated code or user pipeline request.

IV. Experiments

In this section, we present the design of our experiments, including the choice of datasets, large language model, discussion about the baselines, and evaluation methodology.

A. Experimental Setup

Data: We use shipment-level bill of lading data [50] that captures business-to-business international trade and highlights the complexity of supply chains. The effectiveness of identifying shipments potentially circumventing economic sanctions, high tariffs, or engaging in suspicious activities largely depends on the quality of data initially cleaned and preprocessed. Specifically, we use three trade datasets involving imports and exports across multiple nations for commodities that have been subject to recent import prohibitions, high tariff rates, and sanctions, and all may contain risks associated with origin fraud [51, 52]. The datasets are as follows:

1) **Teak**: This dataset includes 69,134 teakwood transactions exporting from 116 countries to the United States, spanning from July 1, 2007, to August 10, 2023 (5,885 days in total). The data source is Panjiva[1].
2) **Grain**: This dataset comprises 145,217 grain transactions from Russia to 118 global destinations, covering the period from May 20, 2021, to November 30, 2022 (560 days in total). The data sources are ExportGenius[2] and ImportGenius[3].
3) **Timber**: This dataset contains 3,087,822 timber exports from Russia to 173 countries, starting from October 20, 2021, to March 31, 2023 (528 days in total). The data sources are ExportGenius and ImportGenius.

These datasets span significant periods and were manually entered, making them susceptible to errors such as typos,

[1]https://panjiva.com/
[2]https://www.exportgenius.in/
[3]https://www.importgenius.com/

inconsistent formatting, and missing information. Table I presents detailed statistics for each dataset. Consequently, data analysis becomes a laborious and time-consuming task for analysts, highlighting the urgency and importance of effective data preprocessing.

TABLE I
DISTRIBUTION FOR REAL-WORLD DATASETS

	Teak	Grain	Timber
# of columns	127	56	29
# of rows	69,134	145,217	>3M
# of unused columns	101	0	6
Missing values (NaN) (%)	48.77	66.94	3.19
Unformatted numbers (%)	19.99	2.84	87.93
Misspelled strings (%)	9.29	18.06	93.19
# of data preprocessing tasks required	12	18	24

Vector Database: We use Qdrant[4] to store and retrieve code functions organized as vector embeddings.

Large Language Model: For this study, we employed CodeLlama-13b-Instruct[5] to accommodate both our hardware constraints and the need for data confidentiality. This model is used to learn from retrieved codes, generate executable Python functions, and modify codes in response to user feedback. Unlike API-based LLMs, CodeLlama-Instruct allows for local execution, offering greater flexibility and control.

Code Library: Our goal is to demonstrate TradeSweep 's effectiveness in handling practical data preprocessing tasks frequently encountered by analysts in industry. Given the diverse range of preprocessing tasks applicable to spreadsheet data, we selected the 12 most relevant operations based on our dataset characteristics—which contained common issues such as missing values, spelling errors, and inconsistent formatting—to include in the initial code library. These selected functions cover a range of tasks, including standardizing date formats, removing punctuation marks, correcting misspellings, and filling in missing values based on other columns. Table II provides an overview of selected functions from the initial code library that TradeSweep is expected to generate.

Hardware Environment: The experiments were conducted using a Tesla P40 GPU with 8 cores, 38 GB of RAM, and 500 GB of disk memory. This hardware setup ensures efficient processing and management of computational tasks associated with code generation and evaluation.

B. Baselines

Existing state-of-the-art tools (SkCoder, CleanAgent, etc) are widely recognized for their code generation capabilities. However, these tools are specialized for distinct tasks, making direct comparisons challenging. For instance, SkCoder focuses on task-specific code synthesis, while CleanAgent emphasizes interactive data cleaning processes. To ensure a meaningful evaluation of TradeSweep in generating effective and relevant code, we designed three baselines that simulate key characteristics of these approaches while maintaining compatibility with our experimental setup.

[4]https://qdrant.tech/

[5]https://github.com/meta-llama/codellama

TABLE II
PREPROCESSING FUNCTIONS FROM THE INITIAL CODE LIBRARY

Task	Explanation
Remove columns	Delete a list of unwanted column(s).
Filter rows	Only keep rows that satisfy a certain condition and delete the rest.
Clean numbers	Remove non-numeric symbols and convert the value to numbers.
Clean strings	Remove punctuation marks, quotation marks, and any extra spaces.
Fill in blanks	Replace NaN values with a string defined by user.
Standardize dates	Standardize all date values to a YYYY-mm-dd format.
Correct misspellings	Apply word-embedding and clustering to a column to cluster similar values. Then, in each cluster group, find the most frequent value and correct others to that.
Compare columns and clean	Between a to-clean column and a reference column, group the two columns. Then, for all to-clean values that have the same reference value, find the most frequent to-clean value and update others on this.
Lookup document	Given a to-clean column in the dataset and an external CSV/Excel document, map the to-clean values to a reference column in the document, then create a new column with the mapped values.

1) **Baseline 1 (B1)**: *Code Generation Using Only an LLM prompted with User's Request*. For approaches relying purely on LLMs without additional resources (GPT, Jigsaw, etc), we implemented Baseline 1, excluding the access to the code library. In this baseline, the LLM must independently generate code based on the user's request without external references. This setup allows us to evaluate the LLM's capability to produce relevant code purely from the textual description provided by the user, which may involve significant effort and may result in less effective data cleaning outcomes.
2) **Baseline 2 (B2)**: *LLM Prompted with Candidate Codes and User's Request*. Most SOTA approaches require external APIs for accessing the LLM (ex. CleanAgent and CodeAgent). Due to data confidentiality, we introduced Baseline 2, which excludes data information to evaluate the role of contextual input. In this baseline, we provided the LLM with a set of top-k candidate codes retrieved from the code library, along with the user's request. In TradeSweep , we included data information like column names to aid the LLM in understanding which columns to manipulate without users needing to specify exact column names; In contrast, in B2, such data details are excluded from the prompt. This design assesses the impact of not providing data context on the LLM's performance. The LLM must generate

code based solely on the provided candidate codes and user request, potentially leading to less accurate code generation due to the lack of data-specific guidance.

3) **Baseline 3 (B3)**: *Providing User's Request and the Entire Code Library Without Descriptions*. Many SOTA tools allow the LLM to access a provided repository and extract relevant documents for further tasks. (ex. SWE-Agent). Thus, we designed Baseline 3, where the LLM receives the entire code library, but without any accompanying descriptions or comments. The candidate codes are provided in their raw form, with no explanatory notes. This setup explores the effect of removing code descriptions on the LLM's ability to generate relevant code. Additionally, B3 does not utilize the vector database for prompt augmentation; instead, it provides the full code library as part of the prompt. This approach helps understand the influence of having complete access to code functions without context or descriptions on code generation performance.

C. Evaluation Methodology

For each dataset, we defined a set of data preprocessing tasks based on the dataset's specific content and characteristics, as presented in Table III. The tasks studied in our experiment were determined by data experts who identified the most relevant operations based on our real-world datasets. These datasets presented common challenges, including missing values, spelling errors, and inconsistent formatting, aiming to demonstrate TradeSweep 's effectiveness in tackling practical data preprocessing tasks.

TABLE III
NUMBER OF DATA PREPROCESSING TASKS USED IN OUR EXPERIMENTS

	Teak	**Grain**	**Timber**
Remove columns	1 (101 columns at once)	×	1 (6 at once)
Standardize dates	1	1	1
Clean numbers	1	1	4
Clean strings	×	4	3
Fill in blanks	4	1	2
Correct misspellings	2	×	2
Compare columns and clean	2	6	1
Lookup documents	×	5	5
Others (not in library)	1	×	5

Our objective with TradeSweep is to empower non-programmers to use the tool with ease. Thus, to simulate these users, we prompted the LLM with vague requests, refraining from providing specific instructions on code structure or algorithms.

The code for each task is generated using TradeSweep and the three baselines, and evaluated based on several key metrics. We record both the initial and final versions of the generated codes, noting any revisions made, and measure the time taken to generate the code. This "generation time" is measured by the time it takes for the LLM to retrieve code snippets and generate proposals; If the user modifies the generated code, the revision time is also included. After generating the code, we apply it to the initial, unprocessed dataset and compare the execution outputs to manually preprocessed data to assess the accuracy and effectiveness of each method.

When multiple columns are assigned to perform the same task within the same dataset, we generate separate codes for each column using the same algorithm but with slight modifications to tailor the code to the specific column requirements. This approach allows us to evaluate how well each method adapts to different columns and their unique attributes, ensuring a comprehensive comparison of code generation efficiency and performance.

V. RESULTS

This section outlines the results and presents a discussion on them. Fig. 6 illustrates the user interface of TradeSweep . First, users upload a spreadsheet (e.g., Timber.csv), which is displayed on the left half of the screen. They then enter their data preprocessing request, such as *Only show records where trading country is US*, into the text box. The interface also provides access to all previously completed data preprocessing tasks. Once the LLM generates a code proposal, it appears in the box below, where users can test it on sample data and request revisions if necessary.

We applied the codes generated by TradeSweep and the three baselines to the trade datasets for evaluation. In our analysis, we refer to the initial data as *Init* and the manually cleaned data as *GT* (Ground Truth) for simplicity. The Ground Truth data represents the fully preprocessed and cleaned version of the dataset, achieved through meticulous manual adjustments by data analysts. Our experiments are designed to answer the following questions:

1) Can TradeSweep preprocess data as accurate as preprocessing manually? (Section V-A and V-D)
2) Does TradeSweep generate high-quality data preprocessing functions? (Section V-B)
3) How effective can TradeSweep generate a valid code proposal? (i.e. generating code that passes assertion checks and performs the task correctly?) (Section V-B)
4) To what extent can TradeSweep independently generate valid code proposals, without requiring user feedback? (Section V-C)
5) How does the source code library enhance TradeSweep 's code generation capabilities? (Section V-D)
6) What role does RAG play in improving TradeSweep 's code generation? (Section V-D)

A. Preprocessed Dataset Correctness

After preprocessing the three raw datasets (Init) using the code generated by all three baselines (B1 to B3) and TradeSweep , we compared the resulting preprocessed datasets to the Ground Truth (GT), the manually cleaned dataset. As Table IV illustrates, TradeSweep 's execution results closely

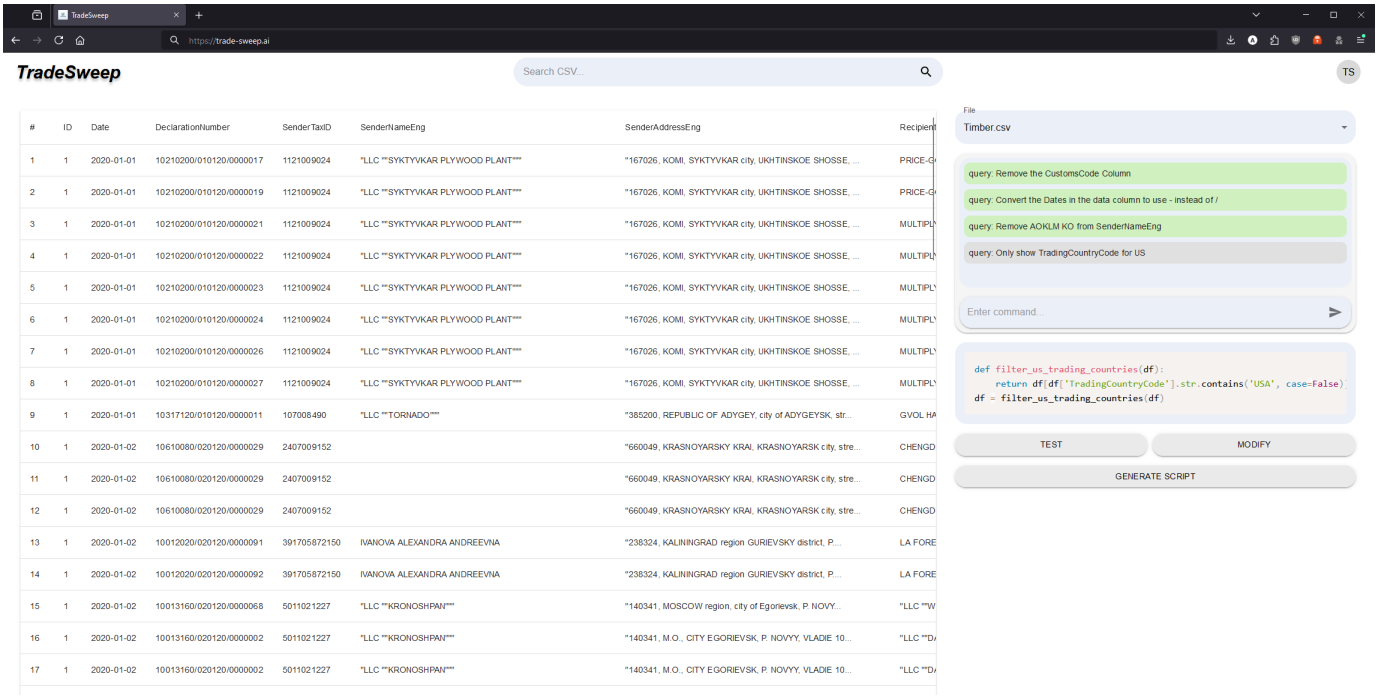

Fig. 6. TradeSweep in action - input data (on the left), user query and generated code (on the right).

align with GT, demonstrating its effectiveness in preprocessing across the three datasets. Baselines 2 and 3 also produced results similar to TradeSweep, as they benefited from referencing and learning from the code library during code generation. Conversely, the codes generated by Baseline 1 showed considerable deviation from Ground Truth. This discrepancy is due to Baseline 1's reliance on the LLM's independent code generation, which often led to the use of different algorithms and approaches compared to those in the code library.

B. Code Quality

To evaluate the code quality of the functions produced by TradeSweep and the baselines, we utilized Pylint[6] and Radon[7] as metrics. According to the documentations, Pylint analyzes source code for errors, warnings, and code smells, providing a quality score between 0 (inefficient) and 10 (efficient); Radon measures *Cyclomatic Complexity*, providing a rank from A (simplest) to F (most complex), with "A" indicating better readability and maintainability. For simple data preprocessing tasks, such as "Clean dates to yyyy-mm-dd format", TradeSweep and the three baselines produced code of comparable quality and complexity, regardless of whether they utilized a code library (see Table V).

However, for more complex tasks, such as comparing two columns and cleaning the target column using the most frequent value from the reference column, the use of a code library significantly improves code quality. TradeSweep obtained a higher Pylint score and generated a more complex code; Baseline 1, with a Radon complexity rank of A, produced code that was too simple to fulfill the task. This enhancement is evident in the results, where functions generated with the aid of a code library demonstrated higher quality scores and are more complex compared to Baseline 1, which relied on independent code generation without library references.

TABLE IV
EVALUATION OF PREPROCESSED OUTPUTS RELATIVE TO GROUND TRUTH (GT)

		Init	GT	B1	B2	B3	TS
Teak	# of cols	127	26	26			
	# of rows	69,134		69,134			
	NaNs (%)	48.77	7.57	10.94	7.09	7.09	**7.09**
	incorrect formats (%)	19.99	0	29.98	9.86	1.03	**1.03**
	typos (%)	9.29	0	25	3.92	3.92	**3.92**
Grain	# of cols	56	61	61			
	# of rows	145,217		145,217			
	NaNs (%)	66.94	62.10	3.76	**63.07**	66.22	63.57
	incorrect formats (%)	2.84	0	9.68	0.09	0.08	**0.08**
	typos (%)	18.06	0	14.43	**2.37**	6.65	3.21
Timber	# of cols	29	23	23			
	# of rows	3,087,822		3,087,822			
	NaNs (%)	3.19	4.55	8.68	4.75	8.24	**4.42**
	incorrect formats (%)	87.93	0	64.86	0.63	12.81	**0.63**
	typos (%)	93.19	0	74.69	4.04	6.83	**2.76**

[6]https://www.pylint.org/
[7]https://pypi.org/project/radon/

TABLE V
COMPARISON OF CODE QUALITY FOR A SIMPLE TASK (CLEAN DATES TO YYYY-MM-DD FORMAT) VS. A COMPLEX TASK (COMPARE TWO COLUMNS AND CLEAN)

		Teak		**Grain**		**Timber**	
		simple task	complex task	simple task	complex task	simple task	complex task
Baseline 1	# of lines (↓)	7	7	7	6	6	6
	Pylint (↑)	4.29	3.64	5.71	4.44	5.71	1.67
	Radon (↑)	A	**A**	A	**A**	A	**A**
Baseline 2	# of lines (↓)	8	6	9	6	8	6
	Pylint (↑)	4.17	5.65	5	5.65	**5.83**	5.65
	Radon (↑)	A	C	A	C	A	C
Baseline 3	# of lines (↓)	8	6	10	6	**6**	6
	Pylint (↑)	**5.83**	5.65	5.38	5.65	5.71	5.65
	Radon (↑)	A	C	A	C	A	C
TradeSweep	# of lines (↓)	**7**	**6**	**7**	**6**	8	**6**
	Pylint (↑)	5.45	**5.65**	**6**	**5.65**	5	**5.65**
	Radon (↑)	**A**	C	**A**	C	**A**	C

As part of TradeSweep , we integrated an assertion check feature to ensure that the generated functions execute correctly before they are presented to the user. If a function encounters execution failures due to exceptions, errors, or other issues, the incorrect function and the corresponding error message are sent back to the LLM for correction. This process is performed in an iterative manner.

Table VI shows the average number of execution failures for TradeSweep compared to the baselines. TradeSweep , by leveraging a code library and data information for the LLM, consistently achieved the lowest number of execution failures across various data preprocessing tasks. In contrast, Baseline 1, which generated functions independently without utilizing a code database, had a higher error rate due to numerous coding errors. Baseline 2 encountered challenges due to the lack of data information; for instance, if a user request was vague (e.g., "Correct misspellings in shipper names" without specifying to apply the function on the "Shipper" column), the LLM erroneously applied the function to a non-existent column, leading to an infinite loop of KeyErrors. Baseline 3 experienced difficulties because the LLM was tasked with interpreting all functions in the code library rather than focusing on the most relevant ones, resulting in a higher likelihood of errors in its generated functions.

TABLE VI
AVERAGE NUMBER OF EXECUTION FAILURES IN THE GENERATED CODE

	Baseline 1	Baseline 2	Baseline3	TradeSweep
Teak	1.92	∞	0.17	**0.08**
Grain	1.72	∞	0.72	**0.11**
Timber	1.17	∞	0.67	**0.08**

C. User Involvement

Table VII records the rate of codes approved by users on the LLM's first attempt. Each value represents the number of codes accepted on the first try, divided by the total number of accepted codes. TradeSweep achieved the highest rate of user-approved codes across all three datasets without needing revisions, exhibiting better performance in generating a function that meets the user's demand on the LLM's first attempt by leveraging knowledge from the code library. In contrast, Baseline 1 often produces unstable results initially, as it generates code independently without the benefit of the code library. This results in frequent misalignment with vague user requests (e.g., "Clean numbers in net weights") and requires more explicit instructions (e.g., "Clean numbers in net weights by removing commas and converting the values to float numbers"). The lowest acceptance rate was noted for Baseline 2 due to the absence of column name information provided to the LLM. Although Baseline 2 used the same algorithm as TradeSweep , the lack of column name information necessitates nearly all preprocessing tasks to undergo revision, highlighting the need for accurate column specification (e.g., "Clean numbers in the NetWeight column" rather than "Clean numbers in net weights"). Baseline 3, while also capable of generating correct codes initially, requires slightly more user involvement compared to TradeSweep . In this baseline, although the LLM often selects the correct code for reference, the absence of function descriptions leads to over- or under-modification of reference code, resulting in additional revision requests.

TABLE VII
ACCEPTANCE RATE OF INITIAL CODE PROPOSALS

	Baseline 1	Baseline 2	Baseline3	TradeSweep
Teak	5/12	1/12	9/12	**10/12**
Grain	5/18	5/18	12/18	**12/18**
Timber	15/24	5/24	17/24	**18/24**

D. Enhancing Code Generation Capabilities

To evaluate the impact of how having a code library can enhance the generation of data preprocessing functions, we compared the overall results of data preprocessed by TradeSweep with Baseline 1, as the primary distinction between the two lies in TradeSweep's use of a code library. After conducting preprocessing on the three datasets, Table VIII illustrates that TradeSweep, by leveraging example code functions from the library, consistently produces suitable and executable code that can preprocess data with much higher correct-value rates (comparable to the Ground Truth data) compared with Baseline 1.

TABLE VIII
IMPACT OF CODE LIBRARY IN GENERATING RELEVANT CODE FOR DATA PREPROCESSING TASKS - BASELINE 1 VS. TRADESWEEP

		Teak	Grain	Timber
Init correct-rate		86.60%	85.94%	9.09%
GT correct-rate		100% (time-consuming)		
Baseline 1	correct-rate	73.09%	87.65%	29.58%
	first-attempt valid rate	41.66%	27.78%	62.5%
TradeSweep	correct-rate	**97.19%**	**97.61%**	**98.17%**
	first-attempt valid rate	83.33%	66.67%	75%

TABLE IX
IMPACT OF RAG ON CODE GENERATION - BASELINE 3 VS. TRADESWEEP

	Teak	Grain	Timber
Baseline 3	555.46	369.99	269.34
TradeSweep	167.55	117.40	94.08

The ability to maintain and expand the code library is crucial for handling more complex and repetitive user requests, as newly generated code functions can be added to the library for future use. The results presented in Table IX suggests that Baseline 3, which provides the entire code library to the LLM, requires significantly more time for code generation compared to TradeSweep. This is due to the increased prompt length, which slows down the process. TradeSweep's use of RAG to retrieve only the top-k most relevant codes from the library reduces prompt length and improves code generation efficiency. In our experiments, we determined that setting the value of k to 3 optimizes performance. Values of $k \geq 4$ result in longer code generation times, while values of $k \leq 2$ increase the likelihood of retrieving irrelevant functions. For instance, if a user requests to "clean the dates," setting k too low may lead to retrieving functions like "clean numbers" or "clean strings", which are less relevant compared to "standardize dates". Thus, $k = 3$ strikes a balance between retrieval accuracy and prompt length efficiency. In conclusion, Baseline 3, which excluded information retrieval and inputted a lengthy prompt to the LLM, required considerably longer time for generating codes compared to TradeSweep.

E. Case Study

To better understand the differences in code generation performance among various baselines and TradeSweep, we analyzed the initial code proposals generated by each method before any user feedback was provided. All baselines received identical user request inputs for each preprocessing task.

Using the example of cleaning the column "Shipper Country", Fig. 7 shows the code proposal generated by the three baselines and TradeSweep. In the scenario, the user's request was: "Compare values in shipper country with shippers, and clean country names with the same shipper to the most frequent value." The ideal approach would involve using the "compare_and_clean" function from the code library and apply it to the "Shipper Country" column while comparing with the "Shipper" column.

*1) **TradeSweep**:* In TradeSweep, the top three relevant functions were retrieved from the code library and inputted into the LLM. The generated code proposal demonstrated that the LLM effectively learned from the "compare_and_clean" function in the code library. By including data column names, TradeSweep accurately identified the columns intended for the task. The proposed code grouped the data by the "Shipper" column and, within each group, identified the most frequent "Shipper Country" value, modifying other values to match this frequent value.

*2) **Baseline 1**:* Baseline 1 generated code based on assumptions about the user's request. Although the proposed code successfully grouped the columns and found the most frequent shipper country, it made errors by converting all strings to lowercase and not handling potential NaN values properly. This led to a higher prevalence of NaNs and incorrectly formatted country names. Unlike TradeSweep, Baseline 1 failed to use the most frequent non-NaN value, resulting in a less accurate output.

*3) **Baseline 2**:* Baseline 2 was provided with candidate codes and the user's request but lacked data information. The code generated attempted to use the "compare_and_clean" function from the code library, but without knowing the exact column names, it made assumptions. The LLM used column names like "shippers" and "shipper_country" instead of the correct "Shipper" and "Shipper Country," leading to incorrect application of the function. In contrast, TradeSweep accurately applied the function to the intended columns by utilizing the provided data information.

*4) **Baseline 3**:* In Baseline 3, all functions in the code library were inputted into the LLM without their descriptions. Although the LLM could understand the usage of each function, the excessive length of the prompt, due to including all functions, led to a significant slowdown in code generation—taking 690 seconds. Despite successfully applying the "compare_and_clean" function, the approach of inputting the entire code library resulted in inefficiencies compared to TradeSweep's method of using only the top-k relevant functions, which streamlined the process and improved generation speed.

Fig. 7. Case Study: A snapshot of code proposal generated by the three baselines (B1 - B3) and TradeSweep.

F. Limitations

Despite TradeSweep demonstrating strong results, certain failure cases remain where the system does not generate the correct code proposal on the first attempt. This happens due to misinterpretations of user requests, e.g., "clean dates" might lead to retrieval of incorrect functions for "clean numbers" or "clean strings".

Another limitation is that the execution outputs of sample data may not provide users with sufficient information to assess the code's performance for certain tasks. For instance, correcting misspellings in company names might result in over- or under-correction, and by observing a limited number of correction examples, users may be unable to confirm whether the code successfully captures all corner cases. In such scenarios, it is helpful to perform an apply-test on the full target data and generate a comparison of names before and after correction. However, applying codes to the full dataset takes much more time than only applying it to sample data.

To address these limitations in the future, we plan to expand the code library with a broader range of well-described functions. We also plan to implement more advanced sampling techniques for execution outputs to provide users with a more representative subset of results. Additionally, optimizing the apply-test procedure to efficiently handle larger datasets can expedite the process, enabling quicker feedback and validation for users.

VI. Conclusion

The rise of automation and programming support through LLMs has significantly reduced the turnaround time for processing large spreadsheets. Our method, TradeSweep, functions as an LLM-driven data preprocessing agent, retrieving suitable functions from a code library and adapting them to fulfill specific data preprocessing tasks. The results demonstrate TradeSweep's effectiveness in practical data transformation scenarios. Future work is aimed at improving the expressiveness and enhancing the range of applicability of TradeSweep.

Acknowledgments

This paper is based upon work supported by the NSF under Grant No. CMMI-2240402. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF. The authors would also like to express their sincere gratitude to Dr. Na Meng for her feedback on an earlier draft of the paper, which helped enhance the presentation and clarity.

REFERENCES

[1] M. Heller, "Large language models and the rise of the ai code generators," *InfoWorld*, May 2023. [Online]. Available: https://www.infoworld.com/article/3696970/llms-and-the-rise-of-the-ai-code-generators.html

[2] P. Ingle, "Top artificial intelligence (ai) tools that can generate code to help programmers," *MarkTechPost*, Mar 2024. [Online]. Available: https://www.marktechpost.com/2024/03/14/top-artificial-intelligence-ai-tools-that-can-generate-code-to-help-programmers/

[3] B. Jury, A. Lorusso, J. Leinonen, P. Denny, and A. Luxton-Reilly, "Evaluating llm-generated worked examples in an introductory programming course." New York, NY, USA: Association for Computing Machinery, 2024.

[4] T. Coignion, C. Quinton, and R. Rouvoy, "A performance study of llm-generated code on leetcode," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE 2024. ACM, Jun. 2024, p. 79–89.

[5] B. D. Ramel, "Top 10 ai 'copilot' tools for visual studio code," *Visual Studio Magazine*, Jun 2023, accessed: Jul. 22, 2024. [Online]. Available: https://visualstudiomagazine.com/articles/2023/06/30/vs-code-copilots.aspx

[6] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," 2024.

[7] K. biblioteka Beograd, "11 things large language models (llms) can never do." *Medium*, Jun 2024. [Online]. Available: https://medium.com/@kombib/11-things-large-language-models-llms-can-never-do-854304724a15

[8] R. Krishnan, "What can llms never do?" *Strange Loop Canon*, Apr 2024. [Online]. Available: https://www.strangeloopcanon.com/p/what-can-llms-never-do

[9] M. Chen, "10 data analytics challenges and solutions," *Oracle Cloud Infrastructure*, Jun 2024. [Online]. Available: https://www.oracle.com/business-analytics/data-analytics-challenges/

[10] Pathstream, "10 common data analysis challenges facing businesses," *Pathstream*, May 2022. [Online]. Available: https://pathstream.com/data-analysis-challenges

[11] S. Anunaya, "Data preprocessing in data mining: a hands-on guide," *Analytics Vidhya*, Aug 2021, accessed: Jul. 22, 2024. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/08/data-preprocessing-in-data-mining-a-hands-on-guide

[12] A. A. Kandilli, "How is dirty data handled in data analytics?" *Medium*, Jul 2022, accessed: Jul. 22, 2024. [Online]. Available: https://medium.com/@sweephy/how-is-dirty-data-handled-in-data-analytics-1767fb998e37

[13] G. Markus, "Why data preprocessing is necessary in data science," *Medium*, Feb 2024. [Online]. Available: https://medium.com/@gideonmarkus/why-data-preprocessing-is-necessary-in-data-science-546235345fdb

[14] Skillfloor, "https://skillfloor.medium.com/the-role-of-data-preprocessing-in-machine-learning-d9ad9db54d49," *Medium*, Aug 2023. [Online]. Available: https://skillfloor.medium.com/the-role-of-data-preprocessing-in-machine-learning-d9ad9db54d49

[15] P. Li, Z. Chen, X. Chu, and K. Rong, "Diffprep: Differentiable data preprocessing pipeline search for learning over tabular data," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, p. 1–26, Jun. 2023.

[16] R. Nouaji, S. Bitchebe, and O. Balmau, "Speedyloader: Efficient pipelining of data preprocessing and machine learning training." New York, NY, USA: Association for Computing Machinery, 2024.

[17] T. Kim, C. Park, M. Mukimbekov, H. Hong, M. Kim, Z. Jin, C. Kim, J.-Y. Shin, and M. Jeon, "Fusionflow: Accelerating data preprocessing for machine learning with cpu-gpu cooperation," vol. 17, no. 4, 2024.

[18] H. Joshi, A. Ebenezer, J. Cambronero, S. Gulwani, A. Kanade, V. Le, I. Radiček, and G. Verbruggen, "Flame: A small language model for spreadsheet formulas," 2023.

[19] Y. Song, S. Ezzini, X. Tang, C. Lothritz, J. Klein, T. Bissyande, A. Boytsov, U. Ble, and A. Goujon, "Enhancing text-to-sql translation for financial system design," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '24. ACM, Apr. 2024, p. 252–262.

[20] X. Xu, C. Liu, and D. Song, "Sqlnet: Generating structured queries from natural language without reinforcement learning," 2017.

[21] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "Sqlizer: Query synthesis from natural language," *Proceedings of the ACM on Programming Languages*, vol. 1, pp. 1 – 26, 2017.

[22] B. Wang, C. Ren, J. Yang, X. Liang, J. Bai, L. Chai, Z. Yan, Q.-W. Zhang, D. Yin, X. Sun, and Z. Li, "Mac-sql: A multi-agent collaborative framework for text-to-sql," 2023.

[23] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Cao, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K. C. C. Chang, F. Huang, R. Cheng, and Y. Li, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," 2023.

[24] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis," *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[25] S. Gulwani, W. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Communications of The ACM - CACM*, vol. 55, 08 2012.

[26] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A survey

on hallucination in large language models: Principles, taxonomy, challenges, and open questions," 2023.

[27] J. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why johnny can't prompt: How non-ai experts try (and fail) to design llm prompts." New York, NY, USA: Association for Computing Machinery, 2023.

[28] Z. Xu, S. Jain, and M. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models," 2024.

[29] A. Mittal, R. Murthy, V. Kumar, and R. Bhat, "Towards understanding and mitigating the hallucinations in nlp and speech," *Proceedings of the 7th Joint International Conference on Data Science & Management of Data (11th ACM IKDD CODS and 29th COMAD)*, 2024.

[30] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2020.

[31] M. Izadi, J. Katzy, T. van Dam, M. Otten, R. M. Popescu, and A. van Deursen, "Language models for code completion: A practical evaluation," 2024.

[32] A. de Moor, A. van Deursen, and M. Izadi, "A transformer-based approach for smart invocation of automatic code completion," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, ser. AIware '24. ACM, Jul. 2024, p. 28–37.

[33] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. ACM, May 2022.

[34] S. Lu, N. Duan, H. Han, D. Guo, S. won Hwang, and A. Svyatkovskiy, "Reacc: A retrieval-augmented code completion framework," 2022.

[35] M. R. Parvez, W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," in *Findings of the Association for Computational Linguistics: EMNLP 2021*. Association for Computational Linguistics, 2021.

[36] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," 2023.

[37] Y. Tian, J. Zhao, H. Dong, J. Xiong, S. Xia, M. Zhou, Y. Lin, J. Cambronero, Y. He, S. Han, and D. Zhang, "Spreadsheetllm: Encoding spreadsheets for large language models," 2024.

[38] Y. Chen, Y. Yuan, Z. Zhang, Y. Zheng, J. Liu, F. Ni, and J. Hao, "Sheetagent: Towards a generalist agent for spreadsheet reasoning and manipulation via large language models," 2024.

[39] J. Li, Y. Li, G. Li, Z. Jin, Y. Hao, and X. Hu, "Skcoder: A sketch-based approach for automatic code generation," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, May 2023, p. 2124–2135.

[40] B. A. Halperin and S. M. Lukin, "Artificial dreams: Surreal visual storytelling as inquiry into ai 'hallucination'." New York, NY, USA: Association for Computing Machinery, 2024.

[41] S. Roychowdhury, "Journey of hallucination-minimized generative ai solutions for financial decision makers," 2023.

[42] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. ACM, May 2022, p. 1219–1231.

[43] K. Zhang, D. Wang, J. Xia, W. Y. Wang, and L. Li, "Algo: Synthesizing algorithmic programs with llm-generated oracle verifiers," 2023.

[44] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," 2024.

[45] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin, "Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges," 2024.

[46] D. Qi and J. Wang, "Cleanagent: Automating data standardization with llm-based agents," 2024.

[47] A. Khurana, H. Subramonyam, and P. K. Chilana, "Why and when llm-based assistants can go wrong: Investigating the effectiveness of prompt-based interactions for software help-seeking," in *Proceedings of the 29th International Conference on Intelligent User Interfaces*, ser. IUI '24. ACM, Mar. 2024, p. 288–303.

[48] W. Wang, H. Ning, G. Zhang, L. Liu, and Y. Wang, "Rocks coding, not development–a human-centric, experimental evaluation of llm-supported se tasks," 2024.

[49] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021.

[50] A. Flaaen, F. Haberkorn, L. Lewis, A. Monken, J. Pierce, R. Rhodes, and M. Yi, "Bill of lading data in international trade research with an application to the covid-19 pandemic," *Finance and Economics Discussion Series*, vol. 2021, pp. 1–40, 10 2021.

[51] L. Aratani, "Us imports of 'blood teak' from myanmar continue despite sanctions," *The Guardian*, May 2023. [Online]. Available: https://www.theguardian.com/world/2023/may/16/myanmar-teak-wood-import-sanctions

[52] AP NEWS, "Russia smuggling ukrainian grain to help pay for putin's war," Oct 2022. [Online]. Available: https://apnews.com/article/russia-ukraine-putin-business-lebanon-syria-87c3b6fea3f4c326003123b21aa78099