CleanAgent: Automating Data Standardization with LLM-based Agents

Danrui Qi
Simon Fraser University
dqi@sfu.ca

Zhengjie Miao
Simon Fraser University
zhengjie@sfu.ca

Jiannan Wang
Simon Fraser University
jnwang@sfu.ca

ABSTRACT

Data standardization is a crucial part of the data science life cycle. While tools like Pandas offer robust functionalities, their complexity and the manual effort required for customizing code to diverse column types pose significant challenges. Although large language models (LLMs) like ChatGPT have shown promise in automating this process through natural language understanding and code generation, it still demands expert-level programming knowledge and continuous interaction for prompt refinement. To solve these challenges, our key idea is to propose a Python library with declarative, unified APIs for standardizing different column types, simplifying the LLM's code generation with concise API calls. We first propose `Dataprep.Clean`, a component of the Dataprep Python Library, significantly reduces the coding complexity by enabling the standardization of specific column types with a single line of code. Then, we introduce the CleanAgent framework integrating `Dataprep.Clean` and LLM-based agents to automate the data standardization process. With CleanAgent , data scientists only need to provide their requirements once, allowing for a hands-free process.

VLDB Workshop Reference Format:
Danrui Qi, Zhengjie Miao, and Jiannan Wang. CleanAgent: Automating Data Standardization with LLM-based Agents. VLDB 2025 Workshop: Data Driven AI.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sfu-db/CleanAgent.

1 INTRODUCTION

Data standardization, which is pivotal in the realm of data science, aims to transform heterogeneous data formats within a single column into a unified data format. This crucial data preprocessing step is essential for enabling effective data integration, data analysis, and decision-making.

Example 1. *We illustrate the data standardization task in Figure 1. Given the input table T, it is obvious that data in the "Admission Date" column and the "Address" column are in different formats, and the data in the cells of the "Admission Date" column includes two different date formats. The goal of data standardization is to unify the data format in each column in T, to get the standardized table T' satisfying the data scientist's requirements. In Figure 1, the data scientist inputs their requirement to standardize "Admission Date" with the "MM/DD/YYYY HH:MM:SS" format. In the resulting T', data in the cells of the "Admission Date" column follows only one date format, i.e., the "MM/DD/YYYY HH:MM:SS" format.*

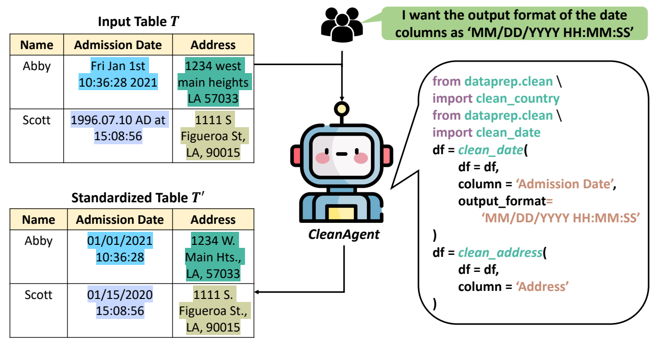

Figure 1: An example of automatic data standardization process with CleanAgent.

Previously, data scientists relied on libraries like Pandas [3] for data standardization, often writing hundreds of lines of code. Standardizing a single column requires identifying its type, validating each cell (e.g., with regex), and converting values to the desired format. With multiple columns of different types, this process demands custom code for each one.

Example 2. *Still considering the data standardization task in Figure 1. For standardizing "Admission Date" and "Address", data scientists need to write the datetime standardization code for "Admission Date" and address standardization code for "Address" using regex. An example standardization code for "Address" is shown as follows.*

```python
def standardize_address(addr):
    # Extract street number and street name
    street = pd.Series(addr).str.extract(r'(\d+ [^,]+)').
     squeeze()
    # Extract state name
    state = "LA"
    # Extract zipcode
    zipcode = pd.Series(addr).str.extract(r'(\d{5})').
     squeeze()
    # Output standardized address
    return f"{street}, {state}, {zipcode}"
```

If the input table T has other column types such as email and IP addresses, data scientists also need to write standardization code tailored for the new types, which is time-consuming.

Recently, LLMs have shown promise for automating this process by generating standardization code from conversational prompts. However, they still require detailed prompt engineering and often multi-turn dialogues [1] for each column type, which limits their efficiency and practicality.

To overcome these limitations, our key idea is to introduce a Python library involving declarative and unified APIs specifically

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment. ISSN 2150-8097.

designed for standardizing different column types. This idea lowers the burden of the LLM, as it now only needs to convert natural language (NL) instructions into succinct, declarative API calls instead of lengthy, procedural code. Such an approach simplifies the LLM's code generation process for data standardization, requiring just a few lines of code.

The pursuit of simplicity, however, introduces two primary challenges. The first challenge (***C1***) is the design of the declarative and unified APIs for data standardization, ensuring it can effectively reduce the intricacies involved in standardizing specific column types (ideally one line of code per column type). The second challenge (***C2***) centers on optimizing the interaction between data scientists and LLMs. Our goal is to minimize human involvement, ideally allowing data scientists to input their standardization requirements in one instance, thereby enabling an autonomous and hands-off data standardization process.

To solve C1, we propose the type-specific `Clean` module in the Dataprep Library, named `Dataprep.Clean`. By observing and summarizing the common steps of data standardization for specific column types, we design unified APIs `clean_type(df, column_name, target_format)`, where the `type` represents the desired standardization type, such as date, address, and phone, etc. These unified APIs offer enhanced expressiveness compared to raw Pandas code, reducing the complexity of standardizing specific column types and allowing one to standardize a column with only one line of code.

To address C2, we propose the CleanAgent framework, which automates data standardization by combining `Dataprep.Clean` with LLM-based agents [5, 6]. Once users specify their goals, the agents autonomously plan, generate, and execute the necessary steps. Data scientists simply provide the table and requirements; CleanAgent then annotates column types, generates concise Python standardization code, and runs it automatically.

Example 3. *Continuing with Example 1. Given an input table T which needs to be standardized and the data scientists' requirements, the CleanAgent first recognizes that the "Admission Date" column belongs to the date type, and the "Address" column belongs to the address type. According to the column-type annotation results, the CleanAgent generates and executes Python code for standardization by calling the "clean_date" and "clean_address" functions, then returns the standardized table T'.*

In summary, our contributions are: (1) `Dataprep.Clean`, an open-source library that simplifies data format standardization with type-specific functions; (2) CleanAgent , which automates standardization by combining `Dataprep.Clean` with LLM-based agents; (3) a user-friendly web application and interface for CleanAgent , allowing users to select data and standardize it; and (4) open-source code on GitHub and a demonstration video on YouTube."

2 TYPE-SPECIFIC STANDARDIZATION API DESIGN

In this section, we first describe the common steps of data standardization. Then, we introduce the type-specific API design of `Dataprep.Clean`.

Common Steps of Data Standardization. Inspired by the steps of how human users standardize data cells, we identify three common steps of data standardization. We take the `datetime` column type as an example to illustrate these steps.

Assume a data scientist is dealing with an `datetime` column including two records *"Thu Sep 25 10:36:28 2003"* and *"1996.07.10 AD at 15:08:56"*. The data scientist wants to unify the messy column into a target format *"YYYY-MM-DD hh:mm:ss"*.

(1) Split. In the beginning, the data scientist needs to split the datetime string into several single parts, which include one kind of specific information. In our example, the data scientist can get several tokens {`'Thu'`,`'Sep'`,`'25'`,`'10'`,`'36'`,`'28'`,`'2003'`} from the first record by using space and colon as separators. Each different type has its splitting strategy, which may not always be splitting the string into tokens. For example, the data scientist will split the email string into the `username` part and the `domain` part.

(2) Validate. Standardization can only be performed on valid inputs. Thus, the second step should be validation. For example, if the string "little cat" is an instance of the `datetime` column, this string is invalid, and the data scientist will transform it to a default value like `NaN`. Intuitively, a valid string indicates that each part of this string after splitting is valid. Usually, the data scientist will recognize and validate each part by their domain knowledge, some corpus or some rules. If every split part is valid, the string is also valid. For instance, the token `'Sep'` can be recognized as a valid representation of a month, and `'2003'` can be recognized as a valid year.

(3) Transform. The last step of standardization is to transform each split part and combine them into the target format. In our example, because the target format is *"YYYY-MM-DD hh:mm:ss"*, the month `Sep` is transformed into number `09` and recombined with other parts to the target `"2003-09-25 10:36:28"`.

The Design of Unified APIs. The goal of our API design is to enable data scientists to complete all the common steps of standardizing one column with a single function call. Simplicity and consistency are considered the principles of API design. The observation of the common steps of data standardization brings the type-specific API design idea. More specifically, we design the API to be in the following form:

clean_*type*(`df, column_name, target_format`)

where `clean_type` is the function name, `type` represents the type of the current column. The first argument `df` represents the input DataFrame, the second argument `column_name` is the column being standardized, and the third argument `target_format` is the target standardization format users specified. Our API design is flexible and extensible, which makes it convenient for users to add their standardization functions for new data types. Currently, we have 142 standardization functions in `Dataprep.Clean`, each handles one data type. These functions serve to demonstrate the value of a more declarative approach, illustrating that building declarative data standardization tools for LLMs is not only feasible but essential, motivating the community to develop even more advanced tools.

3 CLEANAGENT WORKFLOW

In this section, we first introduce the basic structure of LLM-based agents. Then, we describe the CleanAgent workflow constructed by four agents. The automatic data standardization process can be completed by the cooperation of the four agents in CleanAgent .

Basic Structure of LLM-based Agents. According to the previous surveys on LLM-based Agent [6], an LLM-based agent includes four

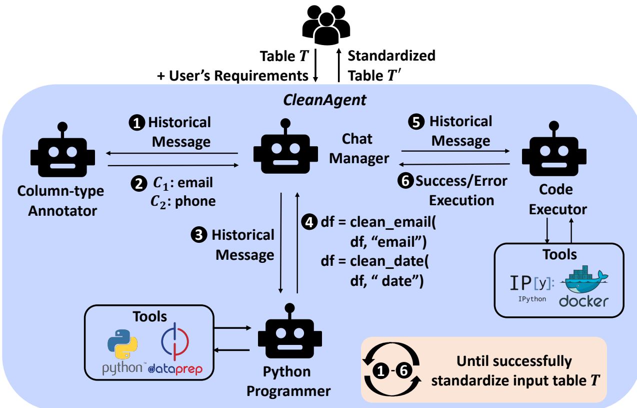

Figure 2: The Workflow of CleanAgent.

main components: (1) a backbone LLM used to generate replies for input prompts, (2) a memory used to store historical conversation messages, (3) a system message defining the role of the agent, and (4) a set of external tools which can be called by the LLM-based agent to complete specific tasks, such as web searching, code execution, etc.

Detailed Workflow. The detailed workflow of CleanAgent is shown in Figure 2. The CleanAgent is composed of four agents, including a *Chat Manager*, a *Column-type Annotator*, a *Python Programmer*, and a *Code Executor*. They can communicate with each other and automatically complete the data standardization process by cooperation. Each agent has its own memory to store the historical conversational messages between it and other agents. Note that the memory of the *Chat Manager* is uniquely comprehensive, encompassing the entire historical conversational messages from all agents within the CleanAgent system. This extensive memory enables every agent in the CleanAgent to generate responses that are informed by the complete historical messages.

The input of CleanAgent includes a table T that needs to be standardized. Data scientists can also input extra requirements such as "the format of the date type column should be MM/DD/YYYY". By receiving the input table and data scientists' extra requirements, CleanAgent stores this information into the *Chat Manager's* memory and then completes the data standardization process. The *Chat Manager* delivers messages in its memory to the *Column-type Annotator*(① in Figure 2). Then, *The Column-type Annotator* receives the table information and leverages an LLM to annotate the type of each column in the input table. If the *The Column-type Annotator* cannot figure out the specific type of one column, the *Column-type Annotator* outputs "I do not know". The annotation result is returned to the *Chat Manager* and stored in the *Chat Manager*'s memory (② in Figure 2).

Thirdly, the *Python Programmer* receives historical messages from the *Chat Manager* including the column-type annotation results (③ in Figure 2), picks up the corresponding clean functions, and generates Python code for the data standardization process. The generated Python code is also returned to the *Chat Manager* and stored in the *Chat Manager*'s memory (④ in Figure 2). Finally, the *Code Executor* receives historical messages from the *Chat Manager* including the column-type annotation results and the generated Python code (⑤ in Figure 2), then executes the generated Python code. If the generated code executes without errors, the standardized table T' is returned; otherwise, the error message is returned to the *Chat Manager* and stored in its memory (⑥ in Figure 2). Then, CleanAgent will retry the whole workflow until it can complete the data standardization process successfully.

4 EXPERIMENTS

Dataset. We use the ***Flights*** dataset from [4], which contains highly inconsistent datetime formats across four attributes. Examples include `"2011-12-08 3:50:00 PM"`, `"2:30pDec 27"`, and `"06:45 AM Sun 25-Dec-2011"`, making it well-suited for evaluating standardization capabilities.

Baselines. We compare CleanAgent with the following two baselines: *(1) GPT-4o + Prompting*. Data standardization code can be directly generated by prompting powerful chat models such as GPT-4o. (2) *Cocoon* [8]. Cocoon is a one-shot data cleaning system that breaks down complex tasks into manageable steps within a workflow, leveraging large language models. It supports tasks like missing value imputation, outlier detection, and functional dependency violation. In this paper, however, we focus on evaluating Cocoon's ability for data standardization.

Note that there are other LLM-based data cleaning approaches, such as *RetClean [2]*. However, *RetClean* primarily adopts a retrieval-based strategy such as RAG to enhance the ability of LLMs for data cleaning, which supplements the LLM with user-provided data sources. This paradigm is not suitable for our scenario.

Ground Truth Generation. We find that GPT-4o can reliably convert individual datetime strings into a target format (e.g., YYYY-MM-DD HH:MM:SS). Thus, we use it to generate cell-level ground truth values and compile them into a complete table.

Metrics. We use the average cell-level matching rate across all columns as our evaluation metric. For a given table T, it is computed as: $d(T_{\text{clean}}, T_{\text{gt}}) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} \mathbf{1}(T_{clean_{ij}} = T_{gt_{ij}})}{m}$, where $\mathbf{1}[\cdot]$ is the indicator function, and T_{clean} and T_{gt} denote the standardized and ground truth tables, respectively.

Implementation. CleanAgent is implemented in Python 3.10.6. Cocoon is run using its official Colab notebook [1] from the GitHub repo [2]. All methods use the `gpt-4o-2024-08-06` model. Experiments are conducted on a MacBook Pro with an M1 chip, 16GB RAM, running macOS Sequoia 15.5.

Table 1: Data standardization performance by comparing different systems.

System	Cell-Level Matching Rate(%)	Latency (s)
GPT-4o	22.0	19.76
Cocoon	21.5	636.62
CleanAgent	**42.5**	29.57

Results. Table 1 presents the comparison of different systems in terms of cell-level matching rate and latency. CleanAgent achieves a 42.5% cell-level matching rate, approximately **2**× higher than that of GPT-4o and Cocoon. These results demonstrate that CleanAgent 's type-specific standardization API enhances the LLM's ability to generate more precise and concise standardization code.

[1] https://colab.research.google.com/github/Cocoon-Data-Transformation/cocoon/blob/main/demo/Cocoon_Stage_Demo.ipynb
[2] https://cocoon-data-transformation.github.io/page/clean

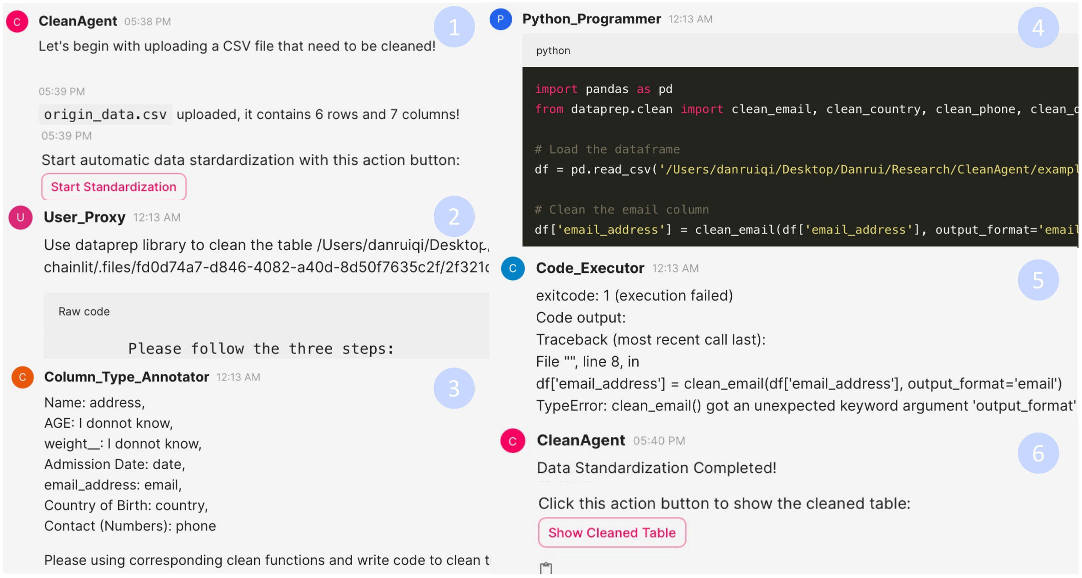

Figure 3: User interface of CleanAgent.

In addition to higher accuracy, CleanAgent also exhibits lower latency compared to Cocoon. This is because Cocoon generates a one-shot SQL query for all columns without the ability to target specific ones, leading to unnecessary overhead.

5 USER INTERFACE OF CLEANAGENT

We developed a web-based user interface for CleanAgent , allowing users to simply upload their tables without performing any operations. The system then automatically returns the standardized results of their data.

Figure 3 shows the user interface of CleanAgent . As area ① shows, users must first upload a CSV file that needs to be cleaned. Then CleanAgent shows the basic information of the uploaded file (number of rows and number of columns). If the users can click the "Start Standardization" button to start the data standardization process by want CleanAgent .

After clicking the "Start Standardization" button, as area ② shows, the `User_Proxy` generates three detailed steps to complete the data standardization task. Firstly, the `Column-type Annotator` receives messages from the `Chat Manager`, annotates and outputs the type of each column, as area ③ shows. Then, the `Python Programmer` picks up standardization functions from Dataprep.Clean based on the type of each column, and write proper Python code using the standardization functions, as area ④ shows. Thirdly, the `Code Executor` executes the Python code by the `Python Programmer` and collects the execution messages, as area ⑤ shows. If the `Code Executor` gets an error message when executing generated Python code, the error message is sent to the *Chat Manager* and becomes part of the prompt of the next try. If the `Code Executor` gets the message of successful execution, CleanAgent will report that the data standardization is completed, as area ⑥ shows. Moreover, users can click the "Show Cleaned Table" button to check whether the standardized table matches their requirements. If so, users can download the standardized table directly. Otherwise, users can input their extra requirements with natural language, and CleanAgent will start a new data standardization process accordingly.

6 CONCLUSION

In this paper, we proposed CleanAgent to automate the data standardization process with `Dataprep.Clean` and LLM-based Agents. We implemented CleanAgent as a web application to visualize the conversations among agents. Other tasks in the data science life cycle, such as data cleaning and data visualization, can also be completed by LLM-based agents [7]. In the future, it is promising that the data science life cycle can be automatically planned and completed by LLM-based agents' cooperation.

REFERENCES

[1] Sibei Chen, Hanbing Liu, Weiting Jin, Xiangyu Sun, Xiaoyao Feng, Ju Fan, Xiaoyong Du, and Nan Tang. 2023. ChatPipe: Orchestrating Data Preparation Program by Optimizing Human-ChatGPT Interactions. *CoRR* abs/2304.03540 (2023). https://doi.org/10.48550/ARXIV.2304.03540 arXiv:2304.03540

[2] Mohamed Y. Eltabakh, Zan Ahmad Naeem, Mohammad Shahmeer Ahmad, Mourad Ouzzani, and Nan Tang. 2024. RetClean: Retrieval-Based Tabular Data Cleaning Using LLMs and Data Lakes. *Proc. VLDB Endow.* 17, 12 (2024), 4421–4424. https://doi.org/10.14778/3685800.3685890

[3] Wes McKinney et al. 2024. pandas: powerful Python data analysis toolkit. https://pandas.pydata.org/ Accessed: 2024-01-25.

[4] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (2017), 1190–1201. https://doi.org/10.14778/3137628.3137631

[5] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework. *CoRR* abs/2308.08155 (2023). https://doi.org/10.48550/ARXIV.2308.08155 arXiv:2308.08155

[6] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huan, and Tao Gui. 2023. The Rise and Potential of Large Language Model Based Agents: A Survey. *CoRR* abs/2309.07864 (2023). https://doi.org/10.48550/ARXIV.2309.07864 arXiv:2309.07864

[7] Siqiao Xue, Caigao Jiang, Wenhui Shi, Fangyin Cheng, Keting Chen, Hongjun Yang, Zhiping Zhang, Jianshan He, Hongyang Zhang, Ganglin Wei, et al. 2023. Db-gpt: Empowering database interactions with private large language models. *arXiv preprint arXiv:2312.17449* (2023).

[8] Shuo Zhang, Zezhou Huang, and Eugene Wu. 2024. Data Cleaning Using Large Language Models. *CoRR* abs/2410.15547 (2024). https://doi.org/10.48550/ARXIV.2410.15547 arXiv:2410.15547