Towards Efficient Data Wrangling with LLMs using Code Generation

Xue Li
MotherDuck & University of Amsterdam
Amsterdam, Netherlands
x.li3@uva.nl

Till Döhmen
MotherDuck
Amsterdam, Netherlands
till@motherduck.com

ABSTRACT

While LLM-based data wrangling approaches that process each row of data have shown promising benchmark results, computational costs still limit their suitability for real-world use cases on large datasets. We revisit code generation using LLMs for various data wrangling tasks, which show promising results particularly for data transformation tasks (up to 37.2 points improvement on F1 score) at much lower computational costs. We furthermore identify shortcomings of code generation methods especially for semantically challenging tasks, and consequently propose an approach that combines program generation with a routing mechanism using LLMs.

ACM Reference Format:
Xue Li and Till Döhmen. 2024. Towards Efficient Data Wrangling with LLMs using Code Generation. In *Workshop on Data Management for End-to-End Machine Learning (DEEM 24), June 9, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650203.3663334

1 INTRODUCTION

Data wrangling tasks such as data cleaning, data integration, and data transformations are part of almost every data analytics workflow and ETL pipeline when working with real-world data. As wrangling tasks can be tedious and time-consuming [7], methods that automate or assist users with such tasks are a valuable addition to BI and data warehousing systems like MotherDuck. When talking to MotherDuck users, we observed that aside from ease-of-use, quality and computational efficiency, interpretable and deterministic solutions are crucial for the adoption and trust in automated wrangling solutions. For ad-hoc analytics, users want to write simple prompts that describe the desired wrangling task, and iterate over ideas quickly, without the need to extensively label data. At the same time solutions need to be able to process millions of rows at low latency. For ETL pipelines, users want to have fine-grained control over the transformations that are applied to the data, and want the automated methods to behave deterministically.

We see two main directions in existing work for data wrangling. One idea is to apply machine learning or language models to each row of data (LLMPR). Machine learning (ML) or fine-tuned SLMs (small language models) [5, 14, 16] require large amounts of labelled training data to be able to perform a desired task, which makes them not well suitable for ad-hoc analytics scenarios. While large language models (LLMs) [9] perform decently in zero-shot or few-shot settings, they incur high latency and are expensive to apply to each row, which makes them unsuitable for million-row scale datasets. LLMPR methods are furthermore not well suited for structured-to-structured data transformation, such as unit conversion, as they struggle with calculations. Another direction is to use program synthesis or programming-by-example (PBE) methods [1, 4, 7] that derive a program (e.g. Pandas, or Excel-Macro) from a given set of input-and-output examples. Those methods have the desired property of being well interpretable and deterministic, and produce code that can be executed efficiently on millions of rows. However, traditionally, program synthesis and PBE methods were challenging to adapt to new tasks. Now, with LLMs, using PBE methods for data wrangling is becoming more feasible. However, even then, PBE methods struggle with semantically challenging tasks (see BingQL-semantics eval in Section 4.2) if they were not specifically implemented to handle them. Furthermore, giving natural language instruction rather than input-output pairs feels more natural for certain tasks, e.g. "detect faulty entries in this column or "convert Roman numerals to Arabic numbers".

We revisited LLMs as prompt-based code generators for data wrangling tasks, and evaluated our method on existing data wrangling benchmarks. Our experiments show that LLM-generated data wrangling code outperforms existing LLMPR and traditional PBE methods, in particular on data transformation tasks, while the performance on other tasks such as entity matching and error detection varies depending on the task and dataset. We attribute this to some tasks requiring a semantic understanding of the input, where code-based approaches underperform compared to LLMPR methods.

We conclude that neither of both directions alone have the potential to lead to high-quality and cost-efficient automated wrangling methods. To leverage the full potential of code generation for data wrangling we therefore see the need for a routing or planning mechanism that decides which task and subset of the data can be processed by a code-based approach.

2 MOTIVATING EXAMPLE

Imagine a data scientist working on a table with 1 million rows. She faces several tasks: converting time to decimal hours, transforming Roman numerals to Arabic numbers, and correcting format inconsistencies and potential errors. Ideally, she is looking for an approach that ensures deterministic transformation, with explainable rationales behind the decision. Above all, she favors an approach that is effective and cost-efficient, preferably flexible to adapt to different tasks. These criteria aim to balance the need for explainability, efficiency, flexibility and quality. She explores the following three different strategies for solving the tasks.

DEEM 24, June 9, 2024, Santiago, AA, Chile
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0611-0/24/06
https://doi.org/10.1145/3650203.3663334

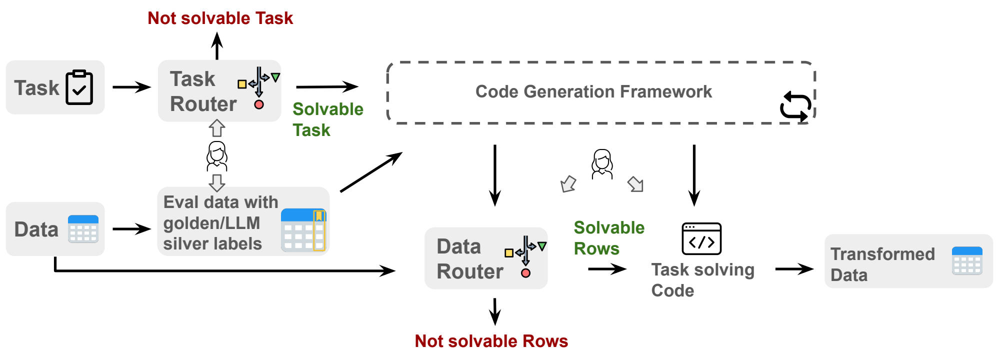


Figure 1: Overview of our envisioned efficient data wrangling workflow.

1) LMs on a Per-Row Basis. Fine-tuning SLMs [16] for specific tasks requires labelled data, which may not be readily available or may require additional effort to prepare. Applying LLMs on a per-row basis [9] can achieve significant capabilities on the tasks, however processing the whole million-rows table through API calls with an LLM can take up to multiple hours depending on the payload size and rate limit constraints, for a single task. Not to mentioned the pricing that can be multiple hundreds of dollars per task. Moreover, the transformation is not deterministic and may lead to different results over multiple runs.

2) Transform-Data-by-Example [1, 3, 4, 6, 13]. This strategy struggles with semantically complex transformations, such as converting between vastly different data formats or correcting errors. Achieving the desired accuracy may require extensive example sets. Moreover, it does not utilize instructions for the task, making it unsuitable for tasks which are more natural to describe with textual instructions rather than input/output pairs.

3) Code Generation and Text2SQL models [2, 8]. They lack of a verification step to ensure the quality and correctness of the generated programs or queries, which can lead to more manual efforts, especially for complex tasks.

Given all these limitations from the existing approaches, she either needs to accept the drawbacks of one of these approaches, or needs to manually combine them, which is tedious and time-consuming. Instead, we envision a new efficient workflow that balances efficiency with the capabilities of handling semantically complex tasks, incorporating a manual or automated verification step to ensure data integrity.

3 ENVISIONED APPROACH

To address the limitations mentioned in Section 2, we propose an efficient data wrangling workflow with code generation using LLMs in Figure 1. The foundational premise of our approach involves a shift from employing LLMs for task execution on a per-row basis to prompting LLMs for function generation that addresses the bulk of records. This proposed workflow leverages LLMs for their reasoning capabilities to reason about complex tasks, diverging from their use as a repetitive task solver.

High-level overview of the approach. In our envisoned approach, the user provides a task instruction alongside an optional set of demonstration examples. The task router determines whether a task can be addressed through a code-based solution. If feasible, the pipeline then direct it to the Code Generation Framework, which ranks multiple candidate code snippets against each other and finally returns the highest scoring code snippet for solving the task. When using the task-solving code snippet, a data router filters rows that are valid for processing and separate the invalid ones. This can be done by generating input filtering code based on the assumptions made in the task solving code. The process then iterates through the unsolvable rows, generating new code iteratively until only a few rows remain that cannot be solved by code. The few unsolvable rows can be subsequently processed by LLMPR or human.

Generating Labels. Conceptually, our envisioned approach will not need human-provided demonstration examples. Instead, we leverage existing LLMPR methods as a labeler [12]. To curate a set of high quality demonstration examples, users can intervene after the labels are generated. Labels will be used as demonstration examples during code generation and for validation of the generated code (see Section 4.1). In the future, we aim to compare different approaches to assist users with curating the labels in a human-in-the-loop manner by intelligently selecting samples to show to the user, e.g. using stratified sampling or approaches like cleanlab [10].

Routing Tasks and Data. The goals for the task and data routers are straightforward: the task router identifies tasks as either solvable or not solvable by code, while the data router determines if an individual row can be handled by the generated code or not.

One possible implementation for the task router is an LLM or a small fine-tuned language model that maps natural language task instruction to an existing taxonomy of task types (see subsection 4.2 for a potential taxonomy). The data router we envision as an LLM-generated code snippet that validates whether a given input conforms to the assumptions that the task solving code makes (e.g. if the task solving code converts Roman to Arabic numerals, the data router code should ensure that input does not contain any characters other than I, V, X, L, C, D, and M). The data router code can be generated alongside the task solving code for each task, and can also be interpreted and adapted by a human-in-the-loop.

The advantages of this workflow are threefold, each contributing to its robustness and utility:

- *Transparency, Explainability and Determinism.* The workflow ensures transparency by making the generated code fully accessible to the user. This enables the outcomes to be deterministic, and enhances the explainability of the workflow by providing the rationale behind the code generation. Except for the demonstrations users chose to provide to the pipeline, majority of the data do not need to go through the LLM.
- *Efficiency.* The efficiency of this workflow is underscored by its reduction in the number of necessary LLM calls. Instead of prompting LLM for each of the N instances, the process requires prompting LLM only $D+C$ times. Given that D (the size of the evaluation dataset) and C (the number of code pieces generated) are typically much smaller than N, this approach can significantly reduce computational resources and time.
- *Flexibility.* This workflow demonstrates flexibility in several aspects, through its ability to generate code across different programming languages (Python, SQL/Macros) and the adaptability introduced by the routing mechanism.

4 EXPERIMENT

The first step towards building the envisioned efficient data wrangling workflow is to explore if, and to what extent, we can use LLMs for code generation that performs better than traditional PBE methods and compliments LLMPR methods. We evaluate the suitability of code generation on common benchmark datasets ([4], [9]), using the provided ground truth labels as demonstration data.

4.1 Code Generation Framework

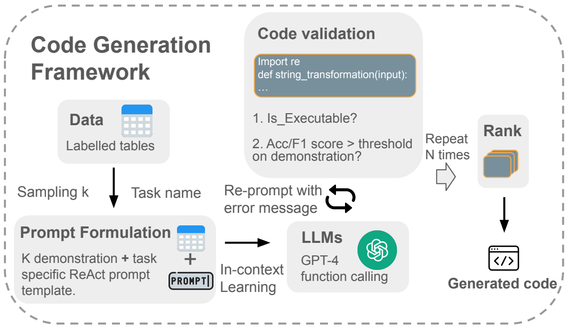

Code Generation Framework
Data
Labelled tables
Sampling k
Task name
Prompt Formulation
K demonstration + task specific ReAct prompt template.
PROMPT
In-context Learning
LLMs
GPT-4 function calling
Re-prompt with error message
Code validation
Import re
def string_transformation(input):
...
1. Is_Executable?
2. Acc/F1 score > threshold on demonstration?
Repeat N times
Rank
Generated code

Figure 2: Code generation framework.

The code generation framework[1] is illustrated in Figure 2, following the *generate*, *verify* and *rank* pipeline. This framework is designed to facilitate the code generation, specifically encouraging the use of regular expressions and string manipulation for tasks like data transformation. Additionally, it leverages substring edit distance metrics for tasks related to entity matching or error detection.

Prompt Formulation. The prompts comprise three primary elements: `general-task-prompt`, `dataset-specific-instruction`, and `demonstrations`. These components are designed to guide the Large Language Models (LLMs) in generating task-specific code, with inputs being the task type (e.g., error detection) and a subset of data samples as demonstrations.

- The `general-task-prompt` remains fixed for each type of data wrangling task, outlining the comprehensive objectives and constraints inherent to the task, ensuring that the model has a general understanding of the task's nature and scope.
- The `dataset-specific-instruction` context through the dataset schema or specific directives in natural language. This tailors the model generation to the distinction of the specific dataset. This component is optional.
- The `demonstrations`, consist of input-output pairs extracted from the annotated evaluation dataset, in the context of this work, they are sampled from the benchmark datasets' training split. For tasks characterized by imbalanced output values, such as error detection and entity matching, stratified sampling is employed to maintain category balance.

Adopting the ReAct prompt framework [15], we prompt models to perform reasoning prior to code generation. This can reduce hallucination, and emphasize explainability and transparency throughout the generation process. Moreover, we integrate the function calling feature from OpenAI [11] to ensure that the output conforms to the desired format.

Code validation. The validation phase of our generated code employs a dual-faced approach to ensure both functionality and efficacy, maintaining the reliability of the code output. Firstly, we check *code executability*, identifying any syntax errors or logical inconsistencies that may prevent code from running. Secondly, we have *performance evaluation* on the demonstration data. The ideal outcome is for the generated code to achieve a 100% accuracy or F1 score on these demonstration inputs, underscoring the precision and relevance of the code to the task. If the generated code fails to meet either criteria, the process iteratively refines the code by re-prompting. In this phase, both the generated code and the associated error messages (e.g. not achieving performance threshold) are fed back into the system as inputs. This feedback loop enables the LLM to either refine the existing code or generate an entirely new solution, informed by the specific issues encountered.

Generated code ranking. Following the validation phase, each piece of successfully validated code, along with its accuracy or F1 score derived from performance on the demonstration data, is added into the generated code pool. This process is repeated N iterations, to compose a set of distinct generated code. The inherent variability of LLM outputs ensures that each iteration has the potential to yield unique code variants with distinct demonstration samples.

Dataset	PBE [4]	LLMPR [9]	Code Generation (Ours)
BingQL-semantics	32.0	54.0	**91.6**
BingQL-Unit	**96.0**	N/A	95.0
Stack-overflow	63.0	65.3	**87.4**
FF-GR-Trifacta	**91.0**	N/A	83.7
Head cases	**82.0**	N/A	74.6
Average	72.8	N/A	**86.46**

Table 1: F1 score on Data Transformation task, $k = 3$.

[1]You can find the implementation at: https://github.com/effyli/efficient_llm_data_wrangling.

Task	Dataset	LLMPR[9]	Code Generation (Ours)
EM	Fodors-Zagats	100	95.5
EM	Beer	100	75.0
EM	DBLP-ACM	96.6	19.7
EM	DBLP-GoogleScholar	83.8	69.7
EM	Amazon-Google	63.5	42.1
EM	iTunes-Amazon	98.2	70.0
EM	Walmart-Amazon	87.0	25.5
DI	Buy	98.5	84.6
DI	Restaurant	88.4	50
ED	Hospital	97.8	23.5
ED	Adult	99.1	100*

Table 2: Accuracy on Data Imputation task and F1 on Entity Matching and Error Detection, $k = 10$.
(* score is only evaluated on the "income" column.)

4.2 Preliminary Results

We test the code generation framework on benchmark datasets from Entity Matching (EM), Error Detection (ED), Data Imputation (DI) and Data Transformation (DT) tasks, following prior work [9]. For these experiments, we generate Python code for its wide usages and rich ecosystem for data science and machine learning tasks. We use GPT-4 as our LLM representing the latest advancement at the time. The results of these experiments are in Tables 1 and 2.

For **Data Transformation**, our experimentation has led to significant advancements. As shown in Table 1, our framework showcased new SoTA performance on BingQueryLogs-semantics, Stackoverflow, achieving F1 scores of 91.6 and 87.4 respectively. The performance on FF-GR-Trifacta and Head cases datasets are slightly lower than the one from PBE method, possibly contributed by the lack of natural language instruction for each task.

We subsequently analyzed the failing cases qualitatively, attempted to derive the taxonomy on categories that are code-solvable and not-code-solvable. Based on 100 tasks in the Bing-query-logs dataset and the domain knowledge, we conclude the categories in Figure 3. In our future work, we will map the tasks to this category, and route the tasks into using code-based solution or not.

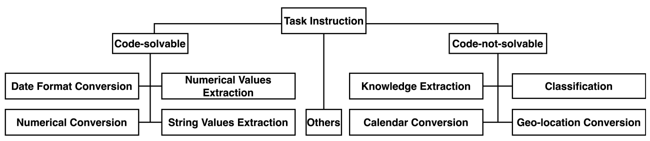


Figure 3: Categories of code-solvable and not-code-solvable.

For **Entity Matching**, our findings presented in Table 2 for the Entity Matching tasks indicate performances generally below those achieved through per-row processing approaches. This outcome aligns with our expectations, given the semantic complexities often inherent in Entity Matching tasks. Our objective was to show to what extent to which datasets could be addressed through code generation. We see that our framework achieved an F1 score of 95.5 on Fodors-Zagats dataset, indicating that a significant portion of this dataset can be managed effectively through string manipulation. However we also observe lower F1 score on datasets such as DBLP-ACM, suggesting the dataset might be more semantically challenging to address through one piece of generated code. With the development of the data router, multiple code can be generated and hence improve the performance.

For **Data Cleaning**, including Data Imputation and Error Detection, we observe similar trends in Table 2 compared to the results from Entity Matching. In the context of the Data Imputation task on the Buy dataset, we achieved an accuracy of 84.6%. This particular example indicates that a relatively straightforward imputation strategy can address a significant portion of the dataset. For the Adult dataset within the Error Detection task, evaluating solely on the income column allowed us to achieve a 100% accuracy. This score was possible because the errors were typos introduced by the original work on developing this dataset. This type of errors can often be systematically detected. For the Hospital dataset, though we only achieved 23 F1 scores, for some columns we can achieve 100% and for some columns we only achieve 0.9%. When we took a closer look on the generated code, we noticed that the generated code are overfitting to the demonstration data, by checking if there are typos on a specific vocabulary curated from the demonstration data, resulting in bad generalization towards the whole table. This is a common type of limitation we observe for the framework.

Faulty Labels During our experiments, we encountered a few faulty labels for the Adult dataset within the Error Detection task. Several entries were incorrectly marked as containing errors in the age and race columns, upon inspection, these errors were not evident. Given this discrepancy, we adjusted our evaluation solely on the income column. This decision was made to ensure the accuracy of our error detection analysis.

5 NEXT STEPS

Based on our encouraging preliminary results, it is clear that the code generation framework holds promise for addressing a substantial portion of data wrangling challenges across various datasets. However, it also shows the need for our envisioned router component. The results are positive for us to further development and refinement of our proposed workflow. Our next steps will focus on the following several key areas.

Generating SQL Macros While our current framework focusing on generating Python code, expanding our capabilities to include SQL macros is a strategy to enhance efficiency especially for large-scale datasets. SQL macros can streamline data processing tasks by leveraging the inherent optimization and parallel processing capabilities of relational database management systems.

Better Benchmark Datasets The limitation of existing benchmark datasets - often small-scale and erroneous - emphasizes the need for more representative data that can mirror real-world complexities. Our experiences thus far highlight the necessity for cleaner, larger-scale benchmark datasets to truly assess the performance and efficiency bottlenecks of current methodologies.

Router Development We plan to develop the envisioned task and data routers and evaluate the effectiveness of the proposed routing mechanisms.

REFERENCES

[1] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. Flashfill++: Scaling programming by example by cutting to the chase. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 952–981.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[3] Qiaochu Chen, Arko Banerjee, Çağatay Demiralp, Greg Durrett, and Işıl Dillig. 2023. Data Extraction via Semantic Regular Expression Synthesis. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 1848–1877.

[4] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-data-by-example (TDE): an extensible search engine for data transformations. *Proc. VLDB Endow.* 11, 10 (jun 2018), 1165–1177. https://doi.org/10.14778/3231751.3231766

[5] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data.* 829–846.

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

[7] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An extensible synthesis framework for data science. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1914–1917.

[8] MotherDuck. [n. d.]. AI That Quacks: Introducing DuckDB-NSQL-7B, A LLM for DuckDB SQL — motherduck.com. https://motherduck.com/blog/duckdb-text2sql-llm/. [Accessed 13-03-2024].

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

[10] Curtis Northcutt, Lu Jiang, and Isaac Chuang. 2021. Confident Learning: Estimating Uncertainty in Dataset Labels. *J. Artif. Int. Res.* 70 (may 2021), 1373–1411. https://doi.org/10.1613/jair.1.12125

[11] openai. [n. d.]. Function calling API. https://platform.openai.com/docs/guides/function-calling. [Accessed 15-03-2024].

[12] Aditya G. Parameswaran, Shreya Shankar, Parth Asawa, Naman Jain, and Yujie Wang. 2024. Revisiting Prompt Engineering via Declarative Crowdsourcing. *14th Annual Conference on Innovative Data Systems Research (CIDR '24).* (2024).

[13] Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic programming by example with pre-trained models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 100 (oct 2021), 25 pages. https://doi.org/10.1145/3485477

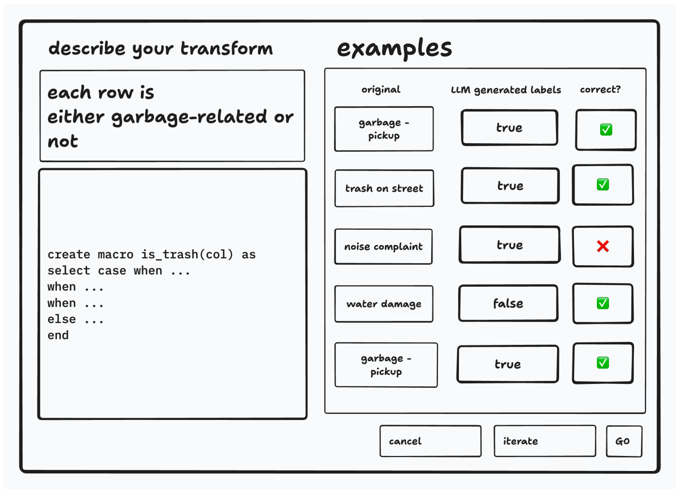
[14] David Vos, Till Döhmen, and Sebastian Schelter. 2022. Towards parameter-efficient automation of data wrangling tasks with prefix-tuning. In *NeurIPS 2022 First Table Representation Workshop.*

[15] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL]

[16] Haochen Zhang, Yuyang Dong, Chuan Xiao, and Masafumi Oyamada. 2023. Jellyfish: A Large Language Model for Data Preprocessing. *arXiv preprint arXiv:2312.01678* (2023).

A ILLUSTRATION ON USERS EXPERIENCE

We provide an illustration of the envisioned User Experience for human-in-the-loop code generation in Figure 4. The illustration consists of three sections, the prompt, the generated code/macro, and the data with LLM generated labels. Users can manually correct the LLM-generated labels and iterate to generate a new code snippet.



```
create macro is_trash(col) as
select case when ...
when ...
when ...
else ...
end
```

original	LLM generated labels	correct?
garbage - pickup	true	✅
trash on street	true	✅
noise complaint	true	❌
water damage	false	✅
garbage - pickup	true	✅

Figure 4: An illustration on UX for human-in-the-loop code generation.