Towards a Holistic Evaluation of LLM Generated Code for Exploratory Visual Analysis

Anamaria Crisan
Tableau Research
USA
acrisan@tableau.com

Enamul Hoque
York University
Canada
enamulh@yorku.ca

ABSTRACT

Large Language Models (LLMs) demonstrate their efficacy in producing code for the analysis and visualization of data. However, evaluating the quality and validity of the generated code poses challenges. While common evaluation metrics assess aspects like code correctness and functionality, they often fall short in capturing the nuances of analytic code, such as whether the code accurately reflects the user's intended data analysis and whether the visualization and textual explanations in the output are appropriate and relevant. To explore these challenges, we study the LLM-generated codes for visual data analysis tasks. Specifically, we use the open coding in the grounded theory method to analyze the generated code and identify six components of analytic code: setup, attribute mapping &creation, data operations, model operations, visualization specification, and summary. We then ideate examples of failure states for each of these components and further derive a set of four types of evaluation (functional, semantic, contextual, and preferential). We conclude by discussing the necessity and challenges of developing multi-criteria metrics for analytic code.

CCS CONCEPTS

• **Human-centered computing** → **HCI design and evaluation methods**; • **Computing methodologies** → Artificial intelligence.

1 INTRODUCTION & MOTIVATION

The program synthesis capabilities of Large Language Models (LLMs) pose exciting opportunities for automating aspects of data science [1, 4]. Already, data scientists and analysts rely on a plethora of AutoDS techniques for a variety of tasks, from data preparation, exploration, modeling, and dissemination [6, 30]. Being able to generate custom code for these tasks, in near real-time, can improve both the speed and reproducibility of data science workflows. However, the capabilities and quality of analytic code the LLM produces can be difficult to assess.

We argue that existing metrics for evaluating LLM-generated code are too coarse to capture the nuances and complexity of data analysis. To motivate this case, we present an example in Figure 1 of an exploratory visual analysis (EVA) task [2]. In this example, a participant from a user study [13] asks a ChatGPT AI agent to visualize *"what decks were the various classes located on?"* using the canonical Titanic dataset[1]. The model outputs a visualization of a tile plot showing the total count (label, hue) of passenger classes (x-axis) by ship deck (y-axis). Generating this analytic code requires the model to correctly map the user's analysis intent to the dataset being analyzed and the plethora of simple (e.g., column mapping, sums) and complex (e.g., feature engineering, model selection) data and model operations. There are myriad ways this could go wrong, from syntactic errors, to failing to understand the nuanced semantics of the data and possible data operations. Readers who are familiar with the Titanic dataset will recognize that there is no 'deck' column in the dataset (Figure 1A). The agent generated this column by extracting the first letter of the cabin code, which it did without user direction but by likely relying on other knowledge it had[2]. The ability to recruit and apply external information, correctly or incorrectly, is both fascinating and makes debugging AI-generated code more complex.

The goal of this preliminary investigation is to develop a framework for discussing these aspects of code generation for data science tasks. By conducting an initial qualitative analysis of ten AI-generated code examples, we propose a framework consisting of two parts. The first part decomposes the generated code into six different types of code components. The second component introduces a set of evaluative lenses to characterize failure states for each of these code components. While our focus in this study is on EVA, further research can expand our approach to other data science tasks, such as data preparation, feature engineering, model selection and optimization, and analysis dissemination. Our objectives in this preliminary investigation are to provide a scaffold for stimulating broader community discussion on AI-generated code for data science.

2 RELATED WORK

We review the recent work on LLMs for visual analytics and evaluation metrics for code generated by these models. While we do not discuss specific models for code generation, interested readers can refer to Liu *et. al.* [19].

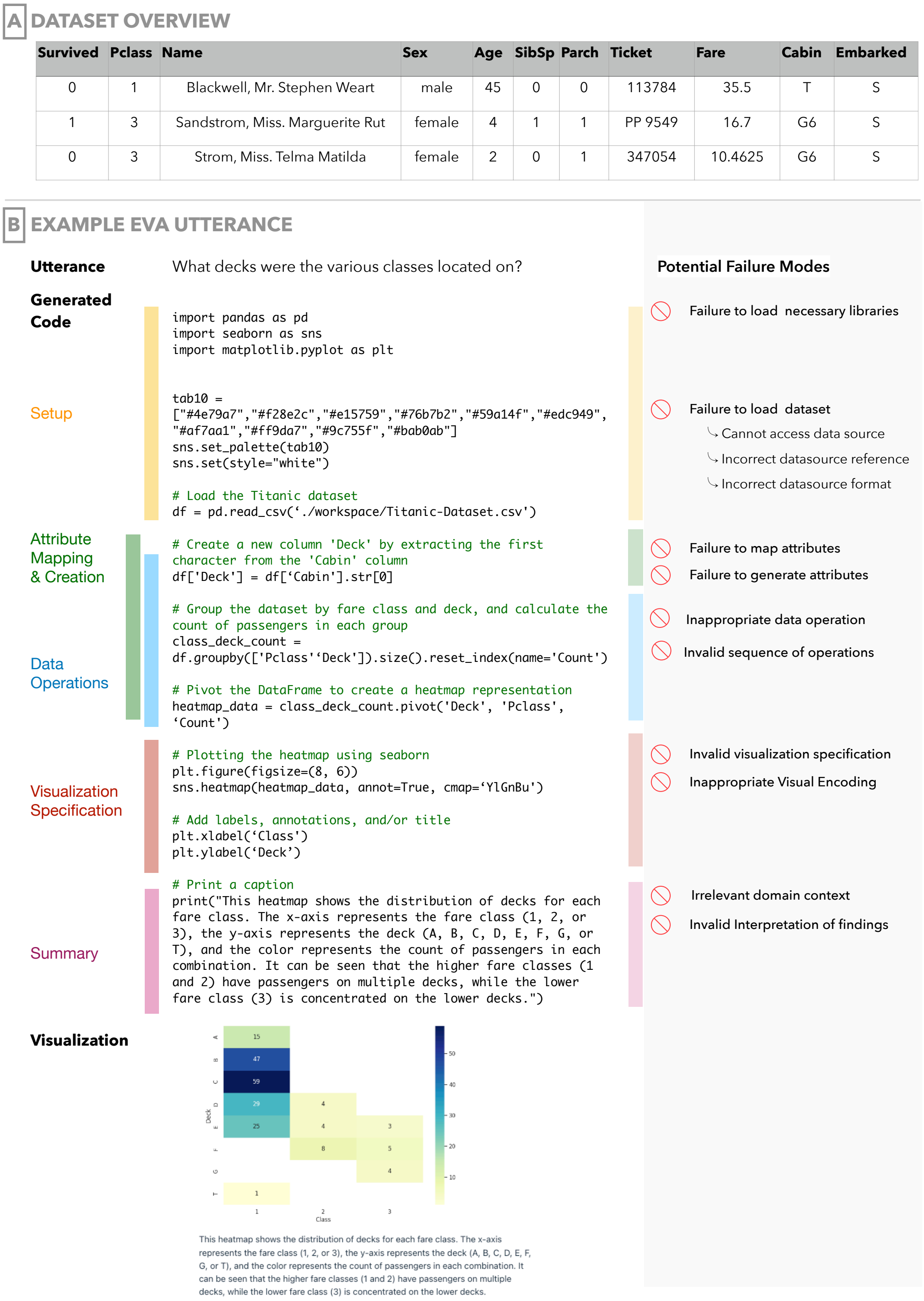
2.1 LLMs for Visual Analytics

With the recent surge of LLMs, studies have explored their applications across diverse data analysis tasks, such as visualization generation [8, 17, 23], automatic chart summarization [14, 28], chart-question answering [9, 24], conversational interface for data analysis [13], and data-driven story generation [12]. A number of benchmark datasets are developed for these tasks, however, these benchmarks typically focus on generating texts [14, 24] or visualizations [21] rather than code. In contrast, common benchmarks for code generation, such as HumanEval [3], MBPP [25] do not

[1] https://hbiostat.org/data/repo/titanic.html

[2] see https://www.encyclopedia-titanica.org/cabins.html and for an example see https://www.kaggle.com/code/ccastleberry/titanic-cabin-features

1st HEAL Workshop at CHI Conference on Human Factors in Computing Systems, May 12, Honolulu, HI, USA
. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM



A DATASET OVERVIEW

Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	Blackwell, Mr. Stephen Weart	male	45	0	0	113784	35.5	T	S
1	3	Sandstrom, Miss. Marguerite Rut	female	4	1	1	PP 9549	16.7	G6	S
0	3	Strom, Miss. Telma Matilda	female	2	0	1	347054	10.4625	G6	S

B EXAMPLE EVA UTTERANCE

Utterance What decks were the various classes located on?

Generated Code

Setup

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt


tab10 =
["#4e79a7","#f28e2c","#e15759","#76b7b2","#59a14f","#edc949",
"#af7aa1","#ff9da7","#9c755f","#bab0ab"]
sns.set_palette(tab10)
sns.set(style="white")

# Load the Titanic dataset
df = pd.read_csv(‘./workspace/Titanic-Dataset.csv’)
```

Attribute Mapping & Creation

```
# Create a new column 'Deck' by extracting the first
character from the 'Cabin' column
df['Deck'] = df[‘Cabin'].str[0]
```

Data Operations

```
# Group the dataset by fare class and deck, and calculate the
count of passengers in each group
class_deck_count =
df.groupby(['Pclass'‘Deck']).size().reset_index(name='Count')

# Pivot the DataFrame to create a heatmap representation
heatmap_data = class_deck_count.pivot('Deck', 'Pclass',
‘Count')
```

Visualization Specification

```
# Plotting the heatmap using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(heatmap_data, annot=True, cmap=‘YlGnBu')

# Add labels, annotations, and/or title
plt.xlabel(‘Class')
plt.ylabel(‘Deck’)
```

Summary

```
# Print a caption
print("This heatmap shows the distribution of decks for each
fare class. The x-axis represents the fare class (1, 2, or
3), the y-axis represents the deck (A, B, C, D, E, F, G, or
T), and the color represents the count of passengers in each
combination. It can be seen that the higher fare classes (1
and 2) have passengers on multiple decks, while the lower
fare class (3) is concentrated on the lower decks.")
```

Visualization

This heatmap shows the distribution of decks for each fare class. The x-axis represents the fare class (1, 2, or 3), the y-axis represents the deck (A, B, C, D, E, F, G, or T), and the color represents the count of passengers in each combination. It can be seen that the higher fare classes (1 and 2) have passengers on multiple decks, while the lower fare class (3) is concentrated on the lower decks.

Potential Failure Modes

- Failure to load necessary libraries
- Failure to load dataset
  - Cannot access data source
  - Incorrect datasource reference
  - Incorrect datasource format
- Failure to map attributes
- Failure to generate attributes
- Inappropriate data operation
- Invalid sequence of operations
- Invalid visualization specification
- Inappropriate Visual Encoding
- Irrelevant domain context
- Invalid Interpretation of findings

Figure 1: Example of analytic code generated for utterance by a generative AI chatbot. [A] Shows a snippet of the analysis dataset. [B] Shows DS steps necessary to generate the code and examples of potential failure modes. The utterance, code, and visualization presented here are derived from a user study with an AI chatbot. Absent is a `Model Operations` step.

specifically focus on exploratory data analysis tasks. DS-1000 is a code generation benchmark that is designed for data science problems [16], but it does not particularly focus on generating visualizations or text summaries. Liu *et al.* [17] do generate visualizations using LLMs but evaluate the generated visualization output primarily based on simple accuracy measures. In sum, there is a gap in understanding the capabilities and limitations of LLMs in generating code for visual analytics which motivated our study.

2.2 Evaluation Metrics

Evaluation methods for generated code can be broadly categorized into three groups: exact match, similarity, and functional correctness. Exact match metrics require that the generated code be identical to a reference. While desirable in certain scenarios, this criterion can often be too strict to apply in practice. Reference-based similarity metrics such as BLEU-score may not align with human judgment [10]. Functional correctness metrics are inspired by test-driven software practices, where code is required to pass pre-defined unit tests. While both exact match and similarity metrics may capture functional correctness by proxy, they also penalize working code that differs from the reference. In contrast, the pass@K metric computes the probability that at least one of *k* generated code examples will pass a unit test for a given problem. While some aspects of data science coding practices are amenable to test driven development (TDD) paradigms, we argue that some data science tasks, notably exploratory data analysis, do not easily conform to TDD. Researchers have also explored evaluation metrics beyond these categories. Dibia *et. al.* [7] considers human preference relative to functional correctness. They found that humans prefer code that may fail pre-specified unit tests (e.g., *pass@K*), but can be easier to modify or adapt compared to those that do pass unit tests. ELO rating has also been used to capture preference in head-to-head comparisons of multiple LLMs [31].

While these metrics may be applied to analytic code, they are likely too coarse to capture its nuances. For example, none of these metrics consider whether the model effectively captures the analyst's intents – something that has been more widely studied by the HCI and data visualization researchers [29].

3 A FRAMEWORK FOR THE EVALUATION OF ANALYTIC CODE

We sought to understand the composition and possible failure states of analytic code that include, but also go beyond, functional correctness that is the present primary evaluative metric. Prior research on the structure of data science code, relative to more general program code, is limited and focuses on programming practices and style [26], not data science, or more specifically EVA, tasks [15]. In this section, we describe a preliminary framework that consists of two parts, a set of code components and a set of evaluative lenses. We describe our approach and provide more details of our framework components.

3.1 Approach

We gathered ten examples from a user study with an LLM chatbot that assisted with EVA tasks conducted on the canonical Titanic dataset [13]. While the steps taken to fine-tune model as well as the construction of the prompt itself are essential for ensuring response quality, we will temporarily leave those concerns aside and focus solely on the code that has been generated by the model.

In this user study, participants posed utterances as analytic statements or questions (e.g., "show me the average income for each class?") based on the given data table, rather than utterances for specific code generation (e.g., 'pandas code to group data by pclass column and then compute the average fare using fare column'). Prompted by these analytic statements or questions, the AI agent was tasked with generating executable Python code, and the resulting output was presented to the end user. The output included a visualization and an associated summary tailored to address the specific user query. We gathered and examined ten such examples (utterances, generated code, and output) and analyzed the code using the open-coding approach. We began by doing a line-by-line annotation of the generated code, collapsing lines with similar annotations. We compared annotations across the code examples and collapsed these annotations into six high-level categories (setup, attribute mapping& creation, data operations, visualization specification, and summary). We next ideated on possible errors that could arise and that were specific to these code annotations. We further collapsed this list of errors into four types of evaluations that could capture them (functional, semantic, contextual, and preferential). Future extensions of this work will explore additional data science tasks and a broader set of annotated human and AI-generated code; for this preliminary investigation, we focus on EVA.

3.2 The Components of Analytic Code

We define six categories of annotations for analytic code, and EVA tasks specifically. These are summarized in Figure 1. When working collaboratively with an AI agent, the analyst could ask the agent to generate specific code pertinent to some category. For example, for attribute mapping/creation and in a computational notebook environment, the analyst could prompt an agent to *"write a lambda function to parse the first character from the Cabin column"*. The example shown in Figure 1 considers a more complex scenario, where an AI agent must generate multi-step and functional analytic code and where the analyst might not have access to modify the generated code. This scenario is similar to what a tool like OpenAI's Code Interpreter, as well as more EVA-specific tools like Chat2Vis [23], LIDA [8], and AI Threads [13].

`Setup` code concerns identifying and loading appropriate libraries, the data, and if applicable, setting options for these libraries. In this example, three libraries are loaded, a color palette is defined, and the options of the `seaborn` are set. The data here is a simple CSV file that must be loaded from the correction location on the file systems. This EVA example forgoes data preparation steps, such as missing value imputation, or other checks on data quality. It also does not consider operations that may be necessary to construct this data, for example, by joining multiple tables in a database. At present, we consider such data preparation tasks out of scope for our preliminary investigation, but, to be an important consideration for extending this work.

The errors that occur in the setup phase can include the failure to load the correct libraries or connect to an appropriate data source. Loading the correct libraries can be a complex task, because unless these libraries are specifically provided to the agent, via system instructions or in the prompt, the AI agent had to infer the necessary libraries. In this example, the AI agent must load data processes and visualization libraries. The Chat2Vis [23] tool loads a standard set of libraries in the event the LLM fails to do so. However, this workaround is not without its limitations. Loading data similarity presents its own challenges. Loading files from disk may be easier because the LLM may be provided with specific instructions as to its location, but, there are no guarantees it use this information when it generates the setup code. Loading files from databases, including performing appropriate queries to extract the data, is a difficult problem [18].

`Attribute Mapping and Creation` code concerns identifying appropriate attributes(columns) of the data that are pertinent to the end-user's utterance. In our example, two attributes of the dataset are used (`PClass`, and `Cabin`). Prior research has identified different types of utterance ambiguity when mapping to dataset attributes. In this example, neither of the attributes necessary for computation is explicitly referred to in the utterance. Moreover, we highlight this example over others because the AI agent must also create a new column, `Deck`, using an existing attribute and the additional knowledge that the first letter of cabin refers to the deck.

In the small set of code that we reviewed, attribute mapping was more common than attribute creation. For the problem of attribute mapping, there are different levels of complexity depending on how ambiguous the reference to the attribute is [27]. Ambiguity can result in semantic misalignment between the utterance and the dataset, leading to errors.

Attribute creation introduces challenges around integrating the present dataset with additional information that the analyst may not specify. Aside from this `Deck` example, there are also instances of the LLM creating ad hoc categories for age groups. While we did not see initial evidence of this, it may be possible for the LLM to make up an attribute that has no proper grounding in the data or analysis. To the best of our knowledge, the integration of data with the LLM's knowledge has not been thoroughly examined in prior research, but it can make debugging AI-generated code difficult because there is no provenance of the agent's reasoning when creating new attributes.

`Data Operations` code concerns transformations of the raw data into its analytic form. Using the raw data exactly could result plotting every single data point into the visualization. While this could be desirable in some circumstances, it is often impractical and difficult to work with. Whether for visualization or further downstream analysis, data is typically transformed, for example, by being aggregated or grouped, and results in an intermediary table that is then visualized. Sometimes multiple data operations need to be executed in sequence. In our examples, these data operations involve first grouping the data by `PClass` and the newly created `Deck` attributes and then aggregating the total number of passengers. Not only does the AI agent need to identify the correct operations to perform, but it must also perform them in the appropriate sequence. Again, utterance does not specify these steps explicitly, they are inferred by the agent.

The errors that can arise in data operation steps are similar to those for attribute mapping and creation.

`Model operations` concern code that fits simple models. Among the code examples we reviewed, the most common model operation was conducting a simple logistic regression to understand the trend of a bivariate relationship. Analysts could have also applied other types of models (e.g., trees, forests, svm etc.) or cluster methods to support their EVA process. While we did not find a lot of evidence for model operations as part of EVA, we include this as a consideration. We consider more intensive model building attempts, which would involve more consideration to feature engineering, model selection, and hyperparameter tuning, to be non-EVA tasks.

AutoML tooling has existed for some time and may make model operations steps more reliable than its predecessors [6]. However, once again, errors and challenges with this step are similar to previous steps.

`Visualization specifications` code defines the part of the output that is provided to the analyst. Similar to prior steps, the utterance may contain a specific reference to an encoding (e.g., *"show a bar chart of.."*) or not. In our example, there is no reference to a specific chart type and the agent 'chose' to generate a tile chart. The choice of the chart is not easily obvious, as it requires some consideration of the attribute types and the charts that support them [22]. For example, a single pie chart would be inappropriate because it cannot effectively represent both `PClass` and `Deck`. However, there is also an element of analyst preference that adds subjectivity to evaluating the efficacy of the result.

Along with previously stated challenges, visualization specification introduces new issues. In tools like Code Interpreter, the underlying code is hidden, and so the visualization, or possibly a text response, can be the first, and sometimes only, point of contact between the analyst and AI agent.

`Summary` code produces a description of the visualization. Prior research by Lungard *et. al* [20] suggested a four-level semantic model for generating visualization captions at the level of the encoding (chart type), descriptive statistics (e.g., averages, correlations), cognitive (patterns, trends), and the domain. The example caption includes references to the encoding (heatmap and a description of *x* and *y-axis* and use of color) and trends (*"it can be seen that.."*). Generating a summary poses various challenges, ranging from the absence of insights at certain semantic levels (e.g., no sentence describing trends or patterns) to the generation of factual errors and hallucinations [14].

3.3 Evaluative Lenses for Analysis Code

After delineating the distinct steps of analytic code, we brainstormed the types of errors that could occur, consolidating our insights into a set of four evaluative lenses presented in Table 1.

While evaluative lenses from the perspective of functional correctness remain relevant, it is crucial to tailor tests specific to different analytic code steps. For example, tests for `setup` would be different than `visualization specifications` or `captions`. Having good coverage of all these different steps is essential for making functional correctness tests useful. However, there are also more nuanced aspects of analytic code and its potential failure states that cannot be easily reduced to functional correctness. Semantic evaluative lenses consider whether the initial utterance can correctly be mapped to the data and computational operations needed to

Evaluation Type	Definition	Examples
Functional	Whether the generated code runs as intended and produces expected results from pre-specified test	Specification Errors; Operation Order
Semantic	Whether the meaning and interpretation of the generated code align with the analyst's intent	Mapping to analytic intent;
Contextual	Whether the generated code is appropriate and relevant to a given domain and/or context	Recruitment of external knowledge
Preference	A subjective evaluation of whether the analysts likes the visualization and/or analysis code style	Interpretability; Adaptability; Encoding Choice

Table 1: Four evaluative lenses that could be applied to assess analytic code.

process it ahead of visualizations. Contextual lenses assess whether the response is pertinent to the particular domain, which can include retrieving pertinent knowledge that is external to the dataset and applying it appropriately. Preference lenses, which Dibia *et. al.* [7] have explored for general code generation, are also pertinent. However, given the nature of EVAs, we extend their definition to consider whether the analyst also has a preference for the output (e.g., chart type, caption).

Whether a multi-step analytic program, as we show in Figure 1, or, some specific steps, these evaluative lenses allow us to apply multiple criteria for holistically understanding the capabilities of LLMs to generate analytic code.

4 CHALLENGES AND CALL TO ACTION

The generation of code that analyzes and visualizes data entails several distinct yet interconnected steps. Errors at these stages can be difficult to distill into unit tests, as the current evaluation paradigm predominantly relies on functional correctness. To demonstrate the limitations of present evaluative paradigms, we conduct a preliminary assessment of AI-generated code from a user study. We develop a framework that compartmentalizes analytic code for exploratory visual analysis into six steps and also proposes a set of four evaluative lenses to assess those steps. With this framework and our motivating example as a starting point, we hope to stimulate a broader conversation within Human-Computer Interaction, Visualization, and Machine Learning Communities.

4.1 Extending our findings

There is a pressing need to provide greater consideration to analytic code, for both EVA and other data science tasks, when evaluating the code-generating capabilities of LLMs. Our preliminary examination can serve as a baseline to solicit community input and participation. Exploring data through the use of visualizations is a common task that occurs across many stages of data science tasks, including data preparation and model building [2, 5]. Extending our findings using further examples of data science code, from repositories like Kaggle or Github, will help us to further enhance our understanding of the nuances of analytic code and the capabilities of LLMs to support its generation.

One particular challenge that we call attention to is the need to develop multi-criteria evaluation metrics for LLM-generated code. While our evaluative lenses propose considerations beyond functional correctness, further extensions of our findings should define additional metrics and their integration. Again, this is not trivial as different components of analytic code can fail in distinct ways. Thus, in addition to proposing benchmarks and metrics specific to EVA and/or other data science tasks, we also need to consider *how* they might be reasonably evaluated by people and machines. As an example, Grunde-Mclaughlin *et. al.* [11] propose methods of adapting crowd-sourcing techniques for addressing errors that arise in LLM chains. Similar innovations may also be necessary for evaluating analytic code generation. Overall, we currently lack insight into the weaknesses of LLMs when it comes to generating analytic code. Using a multi-criteria evaluative metric can help us to understand the scope of its limitations and investigate ways to improve them.

5 CONCLUSION

We present preliminary work in progress that demonstrates the challenges of evaluating LLM-generated code for exploratory visual analysis. Our aim is to stimulate a community dialogue around these challenges and ideate potential solutions. The HCI and VIS research communities have long examined the difficulties that data workers have with EVA, and more generally, developing, debugging, and collaborating around code. For this reason, our communities have much to offer valuable insights that complement and extend the predominately ML perspective on evaluating LLM-generated code.

REFERENCES

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]

[2] Leilani Battle and Jeffrey Heer. 2019. Characterizing Exploratory Visual Analysis: A Literature Review and Evaluation of Analytic Provenance in Tableau. *Computer Graphics Forum* 38, 3 (2019), 145–159. https://doi.org/10.1111/cgf.13678

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex

Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[4] Liying Cheng, Xingxuan Li, and Lidong Bing. 2023. Is GPT-4 a Good Data Analyst? arXiv:2305.15038 [cs.CL]

[5] Anamaria Crisan, Brittany Fiore-Gartland, and Melanie Tory. 2021. Passing the Data Baton : A Retrospective Analysis on Data Science Work and Workers. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 1860–1870. https://doi.org/10.1109/TVCG.2020.3030340

[6] Tijl De Bie, Luc De Raedt, José Hernández-Orallo, Holger H. Hoos, Padhraic Smyth, and Christopher K. I. Williams. 2022. Automating data science. *Commun. ACM* 65, 3 (feb 2022), 76–87. https://doi.org/10.1145/3495256

[7] Victor Dibia, Adam Fourney, Gagan Bansal, Forough Poursabzi-Sangdeh, Han Liu, and Saleema Amershi. 2023. Aligning Offline Metrics and Human Judgments of Value for Code Generation Models. In *Findings of the Association for Computational Linguistics: ACL 2023*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 8516–8528. https://doi.org/10.18653/v1/2023.findings-acl.540

[8] Victor C. Dibia. 2023. LIDA: A Tool for Automatic Generation of Grammar-Agnostic Visualizations and Infographics using Large Language Models. *ArXiv* abs/2303.02927 (2023). https://arxiv.org/abs/2303.02927

[9] Xuan Long Do, Mohammad Hassanpour, Ahmed Masry, Parsa Kavehzadeh, Enamul Hoque, and Shafiq Joty. 2023. Do LLMs Work on Charts? Designing Few-Shot Prompts for Chart Question Answering and Summarization. *arXiv preprint arXiv:2312.10610* (2023).

[10] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software* 203 (2023), 111741.

[11] Madeleine Grunde-McLaughlin, Michelle S. Lam, Ranjay Krishna, Daniel S. Weld, and Jeffrey Heer. 2023. Designing LLM Chains by Adapting Techniques from Crowdsourcing Workflows. arXiv:2312.11681 [cs.HC]

[12] Yi He, Shixiong Cao, Yang Shi, Qing Chen, Ke Xu, and Nan Cao. 2024. Leveraging Large Models for Crafting Narrative Visualization: A Survey. *arXiv preprint arXiv:2401.14010* (2024).

[13] Matt-Heun Hong and Anamaria Crisan. 2023. Conversational AI Threads for Visualizing Multidimensional Datasets. arXiv:2311.05590 [cs.HC]

[14] Shankar Kantharaj, Rixie Tiffany Leong, Xiang Lin, Ahmed Masry, Megh Thakkar, Enamul Hoque, and Shafiq Joty. 2022. Chart-to-Text: A Large-Scale Benchmark for Chart Summarization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 4005–4023.

[15] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2018. Data Scientists in Software Teams: State of the Art and Challenges. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1024–1038. https://doi.org/10.1109/TSE.2017.2754374

[16] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 18319–18345. https://proceedings.mlr.press/v202/lai23b.html

[17] Guozheng Li, Xinyu Wang, Gerile Aodeng, Shunyuan Zheng, Yu Zhang, Chuangxin Ou, Song Wang, and Chi Harold Liu. 2024. Visualization Generation with Large Language Models: An Evaluation. *arXiv preprint arXiv:2401.11255* (2024).

[18] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. arXiv:2305.03111 [cs.CL]

[19] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[20] Alan Lundgard and Arvind Satyanarayan. 2022. Accessible Visualization via Natural Language Descriptions: A Four-Level Model of Semantic Content. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2022), 1073–1083. https://doi.org/10.1109/TVCG.2021.3114770

[21] Yuyu Luo, Nan Tang, Guoliang Li, Chengliang Chai, Wenbo Li, and Xuedi Qin. 2021. Synthesizing natural language to visualization (NL2VIS) benchmarks from NL2SQL benchmarks. In *Proceedings of the 2021 International Conference on Management of Data*. 1235–1247.

[22] Jock Mackinlay, Pat Hanrahan, and Chris Stolte. 2007. Show Me: Automatic Presentation for Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics* 13, 6 (2007), 1137–1144. https://doi.org/10.1109/TVCG.2007.70594

[23] Paula Maddigan and Teo Susnjak. 2023. Chat2VIS: Generating Data Visualizations via Natural Language Using ChatGPT, Codex and GPT-3 Large Language Models. *IEEE Access* 11 (2023), 45181–45193. https://doi.org/10.1109/ACCESS.2023.3274199

[24] Ahmed Masry, Xuan Long Do, Jia Qing Tan, Shafiq Joty, and Enamul Hoque. 2022. ChartQA: A Benchmark for Question Answering about Charts with Visual and Logical Reasoning. In *Findings of the Association for Computational Linguistics: ACL 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 2263–2279. https://doi.org/10.18653/v1/2022.findings-acl.177

[25] Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. 2021. Program Synthesis with Large Language Models. In *n/a*. n/a, n/a. n/a.

[26] Andrew J. Simmons, Scott Barnett, Jessica Rivera-Villicana, Akshat Bajaj, and Rajesh Vasa. 2020. A large-scale comparative analysis of Coding Standard conformance in Open-Source Data Science projects. In *Proc. ESEM '20*. Article 1, 11 pages. https://doi.org/10.1145/3382494.3410680

[27] Arjun Srinivasan, Nikhila Nyapathy, Bongshin Lee, Steven M. Drucker, and John Stasko. 2021. Collecting and Characterizing Natural Language Utterances for Specifying Data Visualizations. In *Proc. CHI'21*. Article 464, 10 pages. https://doi.org/10.1145/3411764.3445400

[28] Benny Tang, Angie Boggust, and Arvind Satyanarayan. 2023. VisText: A Benchmark for Semantically Rich Chart Captioning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7268–7298.

[29] Melanie Tory and Vidya Setlur. 2019. Do What I Mean, Not What I Say! Design Considerations for Supporting Intent and Context in Analytical Conversation. In *2019 IEEE Conference on Visual Analytics Science and Technology (VAST)*. 93–103. https://doi.org/10.1109/VAST47406.2019.8986918

[30] Dakuo Wang, Q Vera Liao, Yunfeng Zhang, Udayan Khurana, Horst Samulowitz, Soya Park, Michael Muller, and Lisa Amini. 2021. How Much Automation Does a Data Scientist Want? *arXiv preprint arXiv:2101.03970* (2021). https://arxiv.org/pdf/2101.03970.pdf

[31] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. https://arxiv.org/abs/2306.05685

Received 01 March 2024; revised 30 March 2024; accepted 07 April 2024