ViseGPT: Towards Better Alignment of LLM-generated Data Wrangling Scripts and User Prompts

Jiajun Zhu[*]
Xinyu Cheng[*]
jiajunzhuchris@zju.edu.cn
pikabi@zju.edu.cn
State Key Lab of CAD&CG, Zhejiang University
Hangzhou, Zhejiang, China

Zhongsu Luo
State Key Lab of CAD&CG, Zhejiang University
Hangzhou, Zhejiang, China
zhongsuluo@zju.edu.cn

Yunfan Zhou
State Key Lab of CAD&CG, Zhejiang University
Hangzhou, Zhejiang, China
yunfzhou@zju.edu.cn

Xinhuan Shu[†]
Newcastle University
Newcastle Upon Tyne, United Kingdom
xinhuan.shu@newcastle.ac.uk

Di Weng[†]
School of Software Technology, Zhejiang University
Ningbo, Zhejiang, China
dweng@zju.edu.cn

Yingcai Wu
State Key Lab of CAD&CG, Zhejiang University
Hangzhou, Zhejiang, China
ycwu@zju.edu.cn

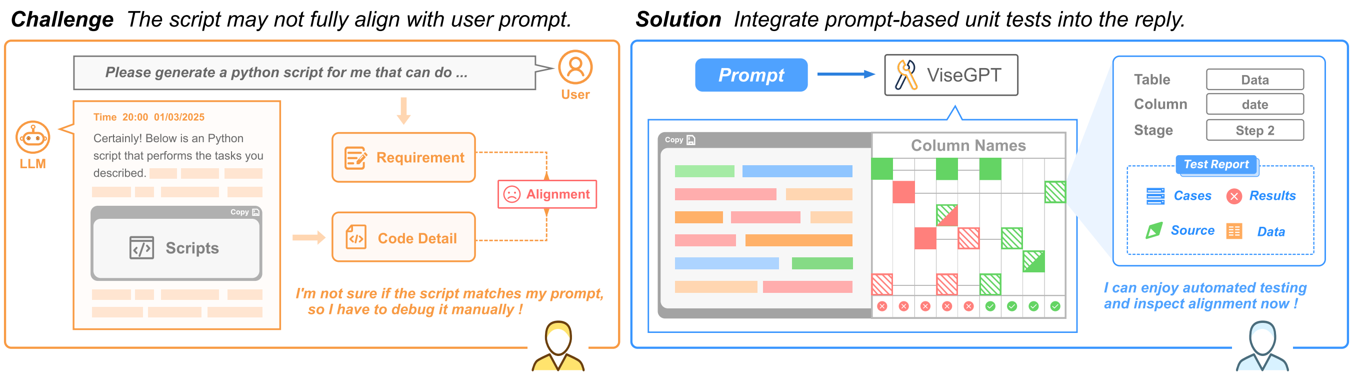

Challenge The script may not fully align with user prompt.
Please generate a python script for me that can do ...
User
LLM
Time 20:00 01/03/2025
Certainly! Below is an Python script that performs the tasks you described.
Copy
Scripts
Requirement
Alignment
Code Detail
I'm not sure if the script matches my prompt, so I have to debug it manually !
Solution Integrate prompt-based unit tests into the reply.
Prompt
ViseGPT
Copy
Column Names
Table
Data
Column
date
Stage
Step 2
Test Report
Cases
Results
Source
Data
I can enjoy automated testing and inspect alignment now !

Figure 1: ViseGPT automatically extracts test cases from the user's prompts for efficient and accuracy debugging.

Abstract

Large language models (LLMs) enable the rapid generation of data-wrangling scripts based on natural language instructions, but these scripts may not fully adhere to user-specified requirements, necessitating careful inspection and iterative refinement. Existing approaches primarily assist users in understanding script logic and spotting potential issues themselves, rather than providing direct validation of correctness. To enhance debugging efficiency and optimize the user experience, we develop ViseGPT, a tool that automatically extracts constraints from user prompts to generate comprehensive test cases for verifying script reliability. The test results are then transformed into a tailored Gantt chart, allowing users to intuitively assess alignment with semantic requirements and iteratively refine their scripts. Our design decisions are informed by a formative study (N=8) that explores user practices and challenges. We further evaluate the effectiveness and usability of ViseGPT through a user study (N=18). Results indicate that ViseGPT significantly improves debugging efficiency for LLM-generated data-wrangling scripts, enhances users' ability to detect and correct issues, and streamlines the workflow experience.

[*]Both authors contributed equally to this research.
[†]Di Weng and Xinhuan Shu are the co-corresponding authors.

CCS Concepts

• **Human-centered computing** → **Natural language interfaces**; *Information visualization*; *User interface programming*; • **Software and its engineering** → *Software testing and debugging*.

Keywords

Data Wrangling Scripts, Debugging Support, Human-AI Interaction

ACM Reference Format:
Jiajun Zhu, Xinyu Cheng, Zhongsu Luo, Yunfan Zhou, Xinhuan Shu, Di Weng, and Yingcai Wu. 2025. ViseGPT: Towards Better Alignment of LLM-generated Data Wrangling Scripts and User Prompts. In *The 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25), September 28-October 1, 2025, Busan, Republic of Korea.* ACM, Busan, Korea, 15 pages. https://doi.org/10.1145/3746059.3747689

1 Introduction

Data wrangling is a process of making raw data suitable for use and analysis [33]. Due to the complexity of data wrangling processes, one common approach is to write custom scripts to perform data cleaning and transformation operations. With recent advances in large language models (LLMs), these data wrangling scripts can be easily generated based on the given instructions, substantially lowering the barrier for less-experienced users to complete complex data wrangling tasks [41, 43, 56, 84]. However, LLM-generated scripts often raise reliability concerns, such as latent defects [6, 50], hallucinations [26, 31], ambiguities [35, 51], and mismatches with user requirements [32, 48], which demand effective approaches to assist users in reviewing and validating these scripts before use.

To help users understand and validate data wrangling scripts, some efforts have been made, such as displaying the script transformation pipeline [79] and directly assisting users in comprehending the input and output space of the scripts [47]. Recent works improve understanding of LLM-generated scripts by introducing tools that represent the inherent data transformation process, including visualizing the behavior of real-time generated scripts as a tree diagram [77] and supporting nodes querying for the operation details [19]. However, users are still required to manually validate whether the LLM-generated scripts can execute correctly and fulfill task-specific requirements. Our formative study further reveals that users often do not seek a comprehensive understanding of the script itself, but rather a direct way to evaluate whether the script aligns with their requirements.

In this paper, we aim to improve the alignment between users' instructional prompts and LLM-generated data wrangling scripts and facilitate efficient verification and maintenance of the functional correctness of these scripts. Inspired by the unit testing approach [64, 73], we propose ViseGPT, a tool that uses automatically generated test cases based on given prompts to assist users in validating LLM-generated data wrangling scripts.

However, leveraging the unit testing approach for LLM-generated data wrangling scripts presents three key challenges:

- First, generating test cases that accurately reflect user requirements from natural language prompts is non-trivial. User prompts often contain implicit information and domain-specific knowledge that are difficult to extract. For example, a prompt like "remove outliers from the sales data" may imply statistical rules (e.g., excluding negative values) that are not explicitly stated. Traditional unit testing relies on precise specifications, but natural language prompts are inherently ambiguous, making automated test case generation prone to misinterpretation.
- Second, validating whether the generated script fully complies with the prompt requires more than simple input-output comparisons. Unlike conventional programming where expected outputs can be predefined, data wrangling tasks often involve complex transformations where intermediate steps influence correctness. Existing testing frameworks focus on functional correctness at the ends but lack mechanisms to assess higher-level semantic alignment with user intent.
- Third, guiding users in refining scripts based on test failures demands an interpretable feedback mechanism. When a script fails to meet certain conditions, users need actionable insights to adjust the script. However, with limited contextual information, users have to manually trace issues according to data flow. This becomes particularly challenging for less-experienced users who may struggle to map test failures back to the original requirements or identify missing edge cases. Without structured guidance, iterative improvement devolves into a trial-and-error process, reducing efficiency gains.

To address these challenges, we developed ViseGPT, a tool that generates test cases based on user prompts, presenting test results within an interactive visualization interface centered around a tailored Gantt chart [76] and supporting iterative sending of expanded prompts for ideal results. Specifically, the system incorporates an analysis model that can extract constraints about LLM-generated data wrangling scripts [47] from users' natural language prompts and process these into systematic test cases, which are then automatically run against the LLM-generated scripts to validate their correctness. This constraint-based approach allows ViseGPT to evaluate whether generated scripts semantically align with user requirements. The test results are presented through an intuitive visualization interface featuring a tailored Gantt chart that leverages symbolic representations of column data transformations and color-coded test status indicators, enabling users to quickly grasp script behavior and identify issues. Furthermore, the interactive interface supports deeper exploration, allowing users to examine detail data at each processing step, review associated test cases and make custom adjustments to existing constraints. To integrate with the test framework, ViseGPT allows users to refine scripts by feeding test reports back into the LLM as expanded prompts for improved script regeneration. This closed-loop system not only helps users verify functional correctness of scripts but also provides actionable feedback for continuous programming improvement. We further evaluate ViseGPT through a user study (N=18), showing improvement and inspiration in debugging LLM-generated data wrangling scripts efficiently and accurately.

2 Related Work

We review LLM-assisted data wrangling script generation, unit test validation, and human-LLM interactive interface design, which are closely related to our study.

2.1 LLM-generated Data Wrangling Scripts

Data wrangling often involves dealing with large volumes of intricate data and varying scripts tailored to different scenarios [17, 49, 68]. Traditionally, the approach requires users to have programming skills to complete the task. Within the data analysis workflow, this is a tedious and error-prone task that can consume up to 80% of a worker's time and effort [12, 34, 47]. Facing these issues, researchers are enthusiastic about using an LLM agent in data wrangling script programming to reduce the learning barrier and improve efficiency [3, 24, 59]. In the HCI community, there is a long history of assisting users in understanding, validating, and refining data wrangling scripts [7, 39, 52, 54, 85].

However, it is impractical to expect these native LLM agents to directly produce correct and user-demand-satisfying data wrangling scripts due to the inherent ambiguities in natural language [35, 51] and the mechanisms of the large models themselves [26, 31, 32, 48]. By taking advantage of the LLM agent itself, some works can effectively help users obtain more accurate data wrangling scripts [10, 40]. In conclusion, current improvement strategies focus on three aspects that display the data transformation process, customize prompts, and backtest generated results.

Displaying the process of data transformation. In the context of data wrangling, it is common to use a script to progressively process one or more tabular data objects [12]. Displaying the detailed changes in the data at each step can help users understand the script's logic and assist in debugging. For example, in XNLI by Feng et al. [19], users can query nodes during the data transformation process. In WaitGPT [77], the script is dynamically updated in a tree structure, providing real-time visualization of the transformations. And in Wrangler [34], the system allows users to directly manipulate the individual steps in the data transformation process. Through reasoning and inspection, users can obtain the logical chain of the script, but this approach remains insufficiently intuitive to present the functional correctness of scripts. When dealing with more complex scripts, users need significant analytical skills to identify issues within the script.

Customizing prompts. For LLMs, specific and well-structured prompts can reduce the degree of freedom in generation, thus improving the accuracy of the results [42]. In Dango [8], users clarify their intent by answering multiple-choice questions posed by the LLM, and receive multiple forms of feedback to aid evaluation, thereby enhancing user communication. In the work by Liu et al. [44], users can opt to break down tasks and input prompts step by step when the LLM-generated results are not satisfactory. And in ColDeco [20], users can decompose a generated solution into intermediate helper columns to understand how the problem is solved step by step. Furthermore, in DIY [57], a sandbox is provided for users to interact with information about queries and a subset of the database, in which end-users can evaluate the system's response. Clearly, rich and structured prompts can lead LLMs to generate more demand-satisfying data wrangling scripts. However, this approach increases the cooperating burden on users to preprocess and structure their prompts.

Backtesting the generated results. To validate script correctness, data test sets can be generated based on the prompt or the predefined functionality of the code. Upon obtaining the test results, the report can be provided to the user, or automated iterative modifications can be performed until the accuracy reaches a predefined threshold. In SpoC [36], the system predicts the program line responsible for the failure and focuses the search on alternative translations of that pseudocode line. Additionally, CodeScore [15] implements an approach to assess code generation by predicting code execution. The quality of the dataset is fundamental to traditional unit testing methods. If the test case scale is too small, it may fail to cover edge cases, while an excessively large validation set will require substantial time for repeated script executions [23]. But this approach can be used to align scripts with user requirements and provides valuable insights.

2.2 Unit Testing for Script Validation

Unit testing refers to the process of testing individual hardware or software units or groups of related units [29]. In practice, developers insert assertions [1, 60, 75] or logical statements within code segments [2, 78] to verify whether the program's execution matches their expectations. These assertions are crucial for identifying issues early in the development phase. To improve efficiency and broaden test coverage, researchers have explored methods for automatically executing unit tests. The goal is to achieve comprehensive coverage of all possible scenarios while maintaining clarity and ease of understanding, thereby facilitating effective quality assurance and debugging processes [21, 30, 80, 82, 86]. This challenge extends to the evaluation of LLM-generated outputs, where assertions must validate not only functional correctness but also alignment with implicit requirements in natural language prompts, which is a core concern in prompt engineering and evaluation research [66].

In the context of data wrangling, unit test generation can be more targeted. By extracting descriptive information about target scripts from user prompts, assertions can be set to verify script correctness [83]. However, in practice, users typically require direct validation of statistical properties (e.g., range ordering) rather than exact data comparisons. To enhance flexibility and structural integrity, the concept of constraint solving can be introduced into the test suite framework. Constraints are extensively utilized across various fields, including software engineering, visualization, and databases, to restrict the possible values of variables, thereby representing partial information about those variables [5]. In constraint programming, unknowns, such as variables, are expressed through a series of constraints, and any solution must satisfy all the restrictions of these constraints [55]. Constraint programming standardizes knowledge representation, enabling users to model domain expertise as high-level expressions.

LLMs have demonstrated potential in generating unit tests with high coverage [9, 25, 74, 80]. When using an LLM agent, users usually provide a natural language prompt that both describes their requirements and acts as a specification for the desired script. This dual role of prompts necessitates methods to extract explicit and implicit conditions, echoing Shreya Shankar et al.'s work on aligning evaluations with human needs [66], which can also be involved in the context of LLM-generated script debugging. An analysis program based on LLMs can extract the expected output criteria from the prompt [27, 87], which is then used to evaluate the functional correctness of the generated scripts. By comparing the actual outcomes with the test cases, the analysis ensures the scripts align with user specifications and performs as intended. This process enhances both the flexibility of the mechanisms and the alignment between scripts and user prompts.

2.3 Advancing UIs for Human-LLM Interaction

In the midst of rapid advances in LLMs, the HCI community has made significant strides in enhancing human-LLM interaction, surpassing common chatbot dialogues and basic API interactions.

Similar to our efforts to facilitate comprehension and verification of generated content, several studies focus on accelerating users' ability to understand and validate LLM outputs efficiently [14, 46]. For instance, VIME [11] enables users to more efficiently understand and validate the outputs of sequential ML models. Our work centers on emerging scenarios where LLMs generate and debug data analysis scripts. We introduce features such as code scroll narration, streaming validation of code correctness post-generation, and step-by-step navigation through data transformation processes.

Additionally, other research explores innovative visual representations and interactive designs for LLMs beyond traditional single-text prompts. For example, Xie et al. [77] use interactive tree maps to assist in understanding. Advanced interactive applications using LLMs have been explored in areas such as educational analytics [71], visualization generation [81] , and anxiety mitigation [38]. SHAPE-IT [62] highlights the ability to facilitate rapid ideation of a wide range of creative behaviors with AI. Following similar principles, our work enables users to interact with intermediate visualizations, allowing real-time refinement and modification of generated code. This provides a more intuitive and fine-grained control over LLM outputs, enhancing usability and effectiveness.

3 Formative Study

We conducted a formative study (N=8) to gain a better understanding of how users debug the LLM-generated data wrangling scripts and inform the design considerations.

3.1 Setup

Recruitment & Screening. We posted recruitment advertisements on university forums and social media. Candidates were required to complete a questionnaire to provide their basic information and relevant work experience. We pre-screened volunteers who had experience using LLM-powered tools to generate data wrangling scripts for our interviews.

Participants. We recruited 8 participants in total (P1-P8), with 2 females and 6 males, aged 20 to 30 years. Specifically, there were 4 postgraduate students whose daily research involves data analysis work (P1-P4), 2 undergraduate students majoring in Computer Science and Technology (P5-P6), and 2 data journalists who perform daily statistical reporting (P7-P8). All participants have prior experience in debugging data wrangling scripts generated by LLMs and correcting issues within these scripts. Given the varying levels of expertise they possess in programming skills and debugging such scripts, this allows them to provide rich and diverse perspectives from different angles.

Interview. First, we asked participants to describe their experiences using LLMs to generate data wrangling scripts, including the specific scenarios in which they applied these tools. In addition, participants were requested to bring chat history from real work scenarios where they had requested LLMs to generate data wrangling code but found the output to be suboptimal. We explored how they identified issues within the scripts, performed diagnostic analyses, and ultimately corrected the errors. Each participant was compensated with $10/hour. The formative interviews lasted 20-40 minutes. All interviews were audio-recorded.

Analysis. Following thematic analysis methods, we first set up three key topics of inquiry. These areas focused on the workflows, specific requirements, and common challenges faced by interviewees when debugging and modifying LLM-generated scripts. To gather comprehensive insights, we combined participants' responses to a series of questions with information obtained through guided discussions during the interviews. By incorporating real-world examples and practical solutions from these discussions, we were able to systematically organize our findings and gain deeper insights. This approach enabled us to effectively guide the design of our system, ensuring it addresses the diverse needs and workflows identified through our analysis.

3.2 Findings

In the following, we present a thematic summary of the key points in the interview research results.

3.2.1 *For what tasks do users frequently rely on LLMs in generating data wrangling scripts, and how do these tools perform?*

Participants recognized that, for simple tasks, conversational LLM agents offer a significant advantage in enhancing the efficiency of generating data wrangling scripts. They have experience using LLMs to generate such scripts across various task scenarios, including scientific research (6/8), routine learning tasks (4/8), machine learning training (2/8), business analytics (2/8), and investigative journalism (2/8). *"Nowadays, whenever I face simple and highly repetitive data wrangling tasks, I opt to use LLMs to handle them swiftly. This allows me to allocate more time to other meaningful endeavors."* (P7) With the assistance of LLMs, end users need only describe their task requirements to receive responses in the form of code along with functional reports. This feature has been particularly well received by users with less programming experience. *"I have limited programming experience in data wrangling, and the LLMs allow me to obtain a usable script without learning programming specifically."* (P6)

However, participants noted that, in complex scenarios, the effectiveness of conversational LLM agents in generating data wrangling scripts may not be promising. When the volume of requirements increases or tasks require deeper analysis, LLMs often produce scripts that either do not meet the task requirements or contain syntactic errors. P8 stated, *"When tasks require deep thinking, using LLMs often results in errors, which can make the efficiency of using them lower than if I were to write the code myself directly."* Most participants (7/8) stressed that it is unrealistic to expect LLMs to generate correct and usable data wrangling scripts in a single round of dialogue in specific scenarios. Instead, issue analysis and script refinement are necessary.

3.2.2 *How to identify issues in LLM-generated data wrangling scripts?*

Generally speaking, after obtaining an LLM-generated script, the interviewees may go through the following two phases: First, they verify the script's usability. Second, upon discovering any issues, they have to assess the source of these problems. (Table 1)

Verify Usability. After obtaining initial scripts, most participants (7/8) conduct preliminary syntax and format checks on the scripts output by LLMs, leveraging their experience to visually inspect for any obvious issues. More experienced participants (4/8)

Table 1: Methods for debugging LLM-generated data wrangling scripts and their limitations in practice.

Debugging Methods	Main Limitations
Initial Script Review	Experience-dependent and difficult to detect deep logic flaws.
Test Dataset Execution	Inadequate edge case coverage and challenging validation.
Segmented Print Output	Information overload and time-consuming for scope narrowing.
Local IDE Debugging	High learning curve and complex environment setup.
LLM-based Correction	May introduce new issues and is inefficient.

can often determine whether the script meets the simple requirements by just reviewing it. However, they unanimously agreed that direct observation is not reliable and should only serve as a pre-processing step, requiring additional methods for thorough verification. *"While monitoring the LLM's streaming output, I sometimes terminate responses mid-generation when spotting issues and restart the conversation. Though this helps catch early issues, it remains an inadequate solution overall."* (P1) Subsequently, participants may delve deeper through further dialogue with the LLM (3/8) or by switching to a local coding environment (5/8). The most straightforward method to test if a script is usable is to run it on a test dataset. Most participants (6/8) chose this approach to verify whether the script has compilation errors or fails to meet their requirements. However, P2 pointed out the data scale dilemma inherent in this method: *"When I test with a small dataset, it might not cover all edge cases of the specific scenario. When using a large dataset, the volume of output results becomes too heavy to easily validate correctness."* Participants (3/8) noted that they might not be able to fully determine the script's correctness just by examining the output results because of concerns about uncaught situations.

Locate and Debug Issues. Merely verifying the usability of a script may not suffice. Most participants (6/8) found that simply describing observed anomalies to LLMs often does not lead to accurate understanding and resolution; it can even misdirect efforts. *"Categorizing the types of problems and narrowing down the scope of modifications greatly enhances the effectiveness of LLMs."* (P5) To reduce the effort of debugging, participants opted to analyze the source of issues themselves and focus on specific code segments. Throughout this process, reviewing variable information during script execution proved invaluable for analysis. Participants employed various methods, such as segmenting notebooks (5/8), directly printing outputs (6/8), and using IDEs to inspect register values (3/8), to trace the process. *"In Python, printing several rows of a table is a common and efficient way to preliminarily verify if the script functions correctly, but this method might miss issues in subsequent rows."* (P2) Furthermore, scripts generated by LLMs may not align with users' coding preferences. Participants (2/8) mentioned encountering overly complex single-step code generated by LLMs, which complicates debugging. Most participants (7/8) agreed that simpler, clearly segmented code is much easier to debug.

3.2.3 What challenges have been encountered when resolving issues in LLM-generated data wrangling scripts?

Three themes emerged when participants described their own difficulties in modifying the scripts.

Debugging by fully checking output data is difficult and tedious. Participants (6/8) pointed out that executing scripts on a test dataset and then checking the output data for debugging is a practical approach. But the manual assess methods mentioned above place high demands on users' capabilities. Some participants (4/8) noted that verifying the correctness of data transformations and ensuring that the generated scripts produce accurate results are tedious and error-prone tasks. *"Checking the output (data results) from the generated script cell by cell is cumbersome."* (P4) To address this, participants (5/8) suggested aggregating data results (e.g., calculating the mean and median of a column of data) or print out a small sample of the data (e.g., the first five rows of a data table) during the execution process to form reports as a way to reduce complexity and alleviate the burden caused by large data scales. However, some participants (3/8) expressed concerns that this approach does not have enough representation to describe data entirely, so it does not reliable when assisting in script modification.

Modifying hampered by limited programming skills. Although LLMs can provide clear annotations to explain each step, participants (8/8) still found modifying the generated code challenging. On one hand, LLMs may use unfamiliar packages or advanced syntax, making adjustments difficult. *"Sometimes there are functions I don't recognize or long statements with deeply nested structures, and even after consulting documentation, I still can't figure out how to correct them."* (P3) On the other hand, addressing issues introduced by LLMs often requires both meticulous attention and programming experience. *"It's common to misjudge the source of a problem, which leads to incorrect modifications, wasting time and effort and potentially causing additional issues."* (P1) Even with LLMs, developers still struggle with making effective modifications, particularly when dealing with intricate logic or unfamiliar constructs.

Refining iteratively and identifying hard-to-detect hallucinations. Most participants (7/8) also opt to engage in further dialogue with LLMs to iteratively refine the script, when manual and simple modifications are insufficient. Unfortunately, this process requires significant effort to communicate the nuances of the desired analysis outcomes and carefully avoid introducing new errors. In the scenario of introducing iterative dialogue, participants (3/8) have encountered instances where LLMs amplify hallucinations, leading to an increase in script errors. *"The agents may misinterpret my intended data wrangling steps that are not syntactically valid, or suggest using functions that do not exist in the given context. These errors sometimes accumulate recursively and are difficult to detect."* (P6) Consequently, most participants (6/8) are reluctant to fully rely on LLM-generated results and also use local environments for debugging. However, even with this approach, they still often find themselves caught in iterative cycles, spending more time identifying and resolving emerging issues, or having to abandon the current conversation history and start a new dialogue.

3.3 Design Considerations

In our formative study, all participants expressed significant interest in incorporating LLMs into their workflow to generate and debug data wrangling scripts. Our design goal is to help users quickly verify the reliability, identify issues, and correct discrepancies in the LLM-generated data wrangling scripts. To achieve this, we propose the following design considerations (D1-D5) to guide the design of the system.

D1. Automatically generate scripts and corresponding test cases for reliability detection based on natural language prompts. After issuing requests to LLMs, there is a risk that the generated data wrangling scripts may contain issues, so debugging these scripts is a necessary step. *"Having an integrated tool to assist in debugging after generating the script (with LLMs) would greatly help maintain task coherence and improve work quality."* (P4) Automatically translating implicit information within natural language prompts into test cases for reliability detection can significantly enhance work efficiency.

D2. Execute unit tests, and provide an overview of test results and data transformation progress with intuitive visualization. Unit testing is an effective and low-threshold method to ensure the reliability of scripts. The model should be designed to perform systematic and high coverage verification. However, even after extracting test points from the prompts and performing automated detection, understanding test information in constraint form and interpreting complex test results can still be challenging for users. To address this, a visualization approach can be introduced, aimed at intuitively summarizing test outcomes. This method should be designed to effectively present structured, step-by-step information, aiding users in comprehending data transformations within scripts. By making error propagation more transparent, it helps users identify the key points in the execution process where failures occur, ultimately supporting more efficient debugging.

D3. Allow users to examine the details of the data for a specific step and customize the test set. In addition to providing an overview of test results, the system should also support viewing and modifying detailed information at any stage. Examining the data details of a specific step in the script, similar to breakpoint debugging, is a common practice in existing workflows and is highly beneficial for analyzing the issues. Furthermore, based on our automatically generated test cases, users may want to make additional customized modifications to support further testing. The system needs to allow for adding or modifying existing test cases.

D4. Support script modification based on test reports. Based on our formative study, we found that users often choose to send both the compiler error messages and the script to LLMs during local compilation and debugging. This method is straightforward and effective. Including test results in the prompt provides the LLM with detailed insight into the problem. These contexts, such as error messages, help the LLM better understand the issues and identify the root causes, allowing for more targeted and precise suggestions to solve the problems. Inspired by this workflow, in our testing framework, if users want to debug a script, we should allow users to send the test results as part of the prompt to the LLMs for analysis and problem resolution, in addition to performing manual modifications and providing more natural language descriptions about their requirements.

D5. Integrate with existing data wrangling workflows seamlessly. Current approaches often lead to a disconnected workflow that disrupts the natural flow of data wrangling tasks. For instance, users may need to switch between different platforms to debug code, which can cause mental context-switching and break the continuity of their work. Our design tries to reduce these issues by keeping everything within a cohesive workflow. We can embed elements directly within the response box in the conversational user interface (CUI), allowing users to stay focused on their script and analysis without unnecessary interruptions. To ensure seamless integration with the existing design, our visualization elements and interaction designs are embedded within the response box, maintaining logical consistency with the original output information.

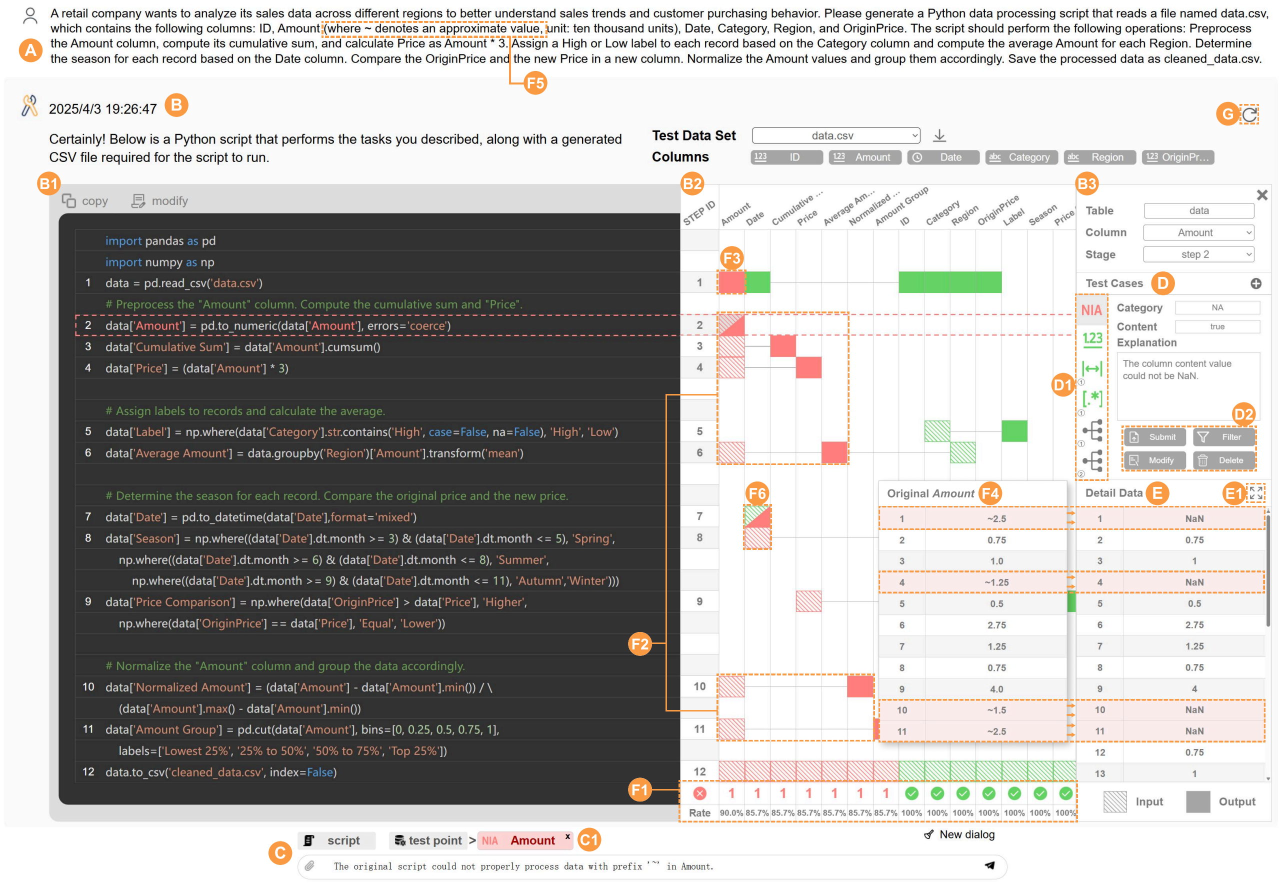
4 ViseGPT

Based on user interviews, ViseGPT should address users' need for an efficient, low-barrier approach to debugging LLM-generated data wrangling scripts. The system comprises three core components: (1) *Automated Test Generation and Execution*: ViseGPT automatically extracts output constraints from natural language prompts, processes them into test cases (**D1**), and conducts unit testing (**D2**); (2) *Workflow Integration and Visualization*: To ensure compatibility with existing data wrangling workflows (**D5**), the system supports unit test customization and provides detailed data visualization (**D3**); (3) *Closed-Loop Iteration*: Test reports are automatically repurposed as enhanced prompts for the LLM code agent, forming an iterative debugging loop (**D4**).

4.1 Usage Scenario

Mary, a marketing specialist at a retail company, intended to analyze sales data across various quarters and regions after processing the original dataset. Although LLMs offer rapid script generation capabilities, the reliability of their output for precise data wrangling requirements remained a concern. She opted to use ViseGPT, a conversational debugging tool designed for LLM-generated data wrangling scripts. After describing her data wrangling requirements in natural language, ViseGPT generated a Python script and executed comprehensive unit tests, displaying the results in the tailored Gantt chart (Figure 2 B2). In the Gantt chart, each column represented an output data column, while each row aligned with the output script step. By observing the summary rows at the bottom (Figure 2 F1), Mary found that some columns in the output data failed to meet the testing criteria, so she decided to inspect the script in detail to resolve the issues.

Exception issues due to type coercion. Mary first reviewed the tailored Gantt chart (Figure 2 B2) for the script. She identified a series of issues that could be traced back to the `Amount` column in the overview (Figure 2 F2), which subsequently propagated through to the `Cumulative Sum`, `Price`, `Price Comparison`, and `Normalized Amount` columns. To address the root cause, she initially focused on the `Amount` column in Step 1 by clicking on its corresponding rectangle in the Gantt chart (Figure 2 F3). Upon inspection, she found that the input data had type issues. After applying a filter, Mary discovered that all the data failing the tests


A retail company wants to analyze its sales data across different regions to better understand sales trends and customer purchasing behavior. Please generate a Python data processing script that reads a file named data.csv, which contains the following columns: ID, Amount (where ~ denotes an approximate value, unit: ten thousand units), Date, Category, Region, and OriginPrice. The script should perform the following operations: Preprocess the Amount column, compute its cumulative sum, and calculate Price as Amount * 3. Assign a High or Low label to each record based on the Category column and compute the average Amount for each Region. Determine the season for each record based on the Date column. Compare the OriginPrice and the new Price in a new column. Normalize the Amount values and group them accordingly. Save the processed data as cleaned_data.csv.

2025/4/3 19:26:47

Certainly! Below is a Python script that performs the tasks you described, along with a generated CSV file required for the script to run.

Test Data Set: data.csv

Columns: ID, Amount, Date, Category, Region, OriginPr...

```
    import pandas as pd
    import numpy as np
1   data = pd.read_csv('data.csv')
    # Preprocess the "Amount" column. Compute the cumulative sum and "Price".
2   data['Amount'] = pd.to_numeric(data['Amount'], errors='coerce')
3   data['Cumulative Sum'] = data['Amount'].cumsum()
4   data['Price'] = (data['Amount'] * 3)

    # Assign labels to records and calculate the average.
5   data['Label'] = np.where(data['Category'].str.contains('High', case=False, na=False), 'High', 'Low')
6   data['Average Amount'] = data.groupby('Region')['Amount'].transform('mean')

    # Determine the season for each record. Compare the original price and the new price.
7   data['Date'] = pd.to_datetime(data['Date'],format='mixed')
8   data['Season'] = np.where((data['Date'].dt.month >= 3) & (data['Date'].dt.month <= 5), 'Spring',
      np.where((data['Date'].dt.month >= 6) & (data['Date'].dt.month <= 8), 'Summer',
        np.where((data['Date'].dt.month >= 9) & (data['Date'].dt.month <= 11), 'Autumn','Winter')))
9   data['Price Comparison'] = np.where(data['OriginPrice'] > data['Price'], 'Higher',
      np.where(data['OriginPrice'] == data['Price'], 'Equal', 'Lower'))

    # Normalize the "Amount" column and group the data accordingly.
10  data['Normalized Amount'] = (data['Amount'] - data['Amount'].min()) / \
      (data['Amount'].max() - data['Amount'].min())
11  data['Amount Group'] = pd.cut(data['Amount'], bins=[0, 0.25, 0.5, 0.75, 1],
      labels=['Lowest 25%', '25% to 50%', '50% to 75%', 'Top 25%'])
12  data.to_csv('cleaned_data.csv', index=False)
```

Table	data
Column	Amount
Stage	step 2

Test Cases

Category	NA
Content	true
Explanation	The column content value could not be NaN.

Original Amount

1	~2.5
2	0.75
3	1.0
4	~1.25
5	0.5
6	2.75
7	1.25
8	0.75
9	4.0
10	~1.5
11	~2.5

Detail Data

1	NaN
2	0.75
3	1
4	NaN
5	0.5
6	2.75
7	1.25
8	0.75
9	4
10	NaN
11	NaN
12	0.75
13	1

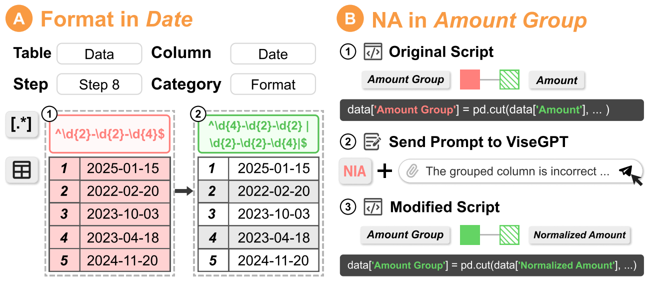
The original script could not properly process data with prefix '~' in Amount.

Figure 2: A screenshot of the ViseGPT user interface. After users send a prompt to ViseGPT (A), the response content in the reply box (B) is organized into three components: (B1) A script view similar to those in common LLM-based code agents; (B2) A tailored Gantt chart providing an overview of test results; (B3) A detail view displaying the test case list and detailed data for one column at one step. Users can resend prompts for script iteration (C), with the option to attach test reports (C1).

contained the "~" prefix in the Amount (Figure 2 F4), used to denote approximate values (Figure 2 F5). This format was incompatible with the float type expected in subsequent operations. However, directly forcing type conversion could result in invalid values. Therefore, it was necessary to strip prefixes while retaining the numerical value (for example, converting "~50" to "50").

Then, in Step 2, she observed that the output did not satisfy the test case of having no missing values in the column (Figure 2 E). It became evident that the initial failure to properly handle the non-float formatted data led to anomalies during type coercion in `pd.to_numeric(data['Amount'])`, resulting in missing values. The missing values subsequently propagated through dependent calculations, affecting the `Cumulative Sum` and `Price` computations with incorrect results.

After identifying the source of the issues, Mary sent ViseGPT the relevant information (Figure 2 C), including details about the failing test case (Figure 2 C1) and the requirement to remove the prefix while retaining the floating-point numbers. In the newly generated script, preprocessing steps were added to handle the format of the `Amount`, resulting in a reduction in the number of test issues.



	①	②
1	2025-01-15	2025-01-15
2	2022-02-20	2022-02-20
3	2023-10-03	2023-10-03
4	2023-04-18	2023-04-18
5	2024-11-20	2024-11-20

```
data['Amount Group'] = pd.cut(data['Amount'], ... )
```

```
data['Amount Group'] = pd.cut(data['Normalized Amount'], ...)
```

Figure 3: Two issues in the usage scenario. (A) Format issues that require expanding the test case. (B) Exception issues arising from grouping by incorrect column.

Format issues that require expanding the test case. Subsequently, Mary started analyzing the `Date` column issues identified in Steps 8 and 9 (Figure 2 F6). By clicking on the rectangles corresponding to these steps, she examined the list of test cases in the detail view and identified that the errors were due to format mismatches during validation (Figure 3 A).

Upon reviewing the prompt, Mary realized that she had not explicitly specified the expected date format for the `Date`. As a result, the automatically generated test cases used '\d{2}-\d{2}-\d{4}' as the default format for validation (Figure 3 A1). However, when the data was processed in `pd.to_datetime(data['Date'],format='mixed')`, it was converted to the '\d{4}-\d{2}-\d{2}' format, causing discrepancies between the test cases and the processed data. To resolve this issue, Mary directly modified the test case content to align with the '\d{4}-\d{2}-\d{2}' format (Figure 3 A2). Then she reran the tests (Figure 2 G) and confirmed that the issues were resolved.

Group label issues arising from grouping by incorrect column. After adjusting the test cases and rerunning them, Mary turned her attention to the issues in the `Amount Group` column in Step 12 (Figure 3 B). This column was supposed to be a label column that categorized the normalized `Amount` values into groups based on their magnitude. However, when examining the tailored Gantt chart, she noticed that the grouping was incorrectly being performed on the original `Amount` column instead of the `Normalized Amount` (Figure 3 B1).

Upon inspecting the specifics of the `Amount Group` in this step, Mary found that it contained missing values. To investigate further, she clicked the magnify button (Figure 2 E1) in the top-right corner of the detail data view for a more detailed look. She found that the original `Amount` column was being used as the basis for grouping, which caused most values — that is, those greater than 1 — to fall outside the expected normalized range. As a result, these values could not be properly grouped, leading to missing values in the `Amount Group` column. To address this issue, Mary uploaded the test results and the requirement to ViseGPT (Figure 3 B2). Following her submission, the grouping column was correctly modified to `Normalized Amount` (Figure 3 B3). Finally, the script passed all the test cases, and Mary got a script that exactly matched her prompt.

4.2 Generating and Executing Test Cases

With the assistance of LLM, ViseGPT can automatically derive output constraints from natural language prompts and processes them into test cases for unit test execution. In another work on data wrangling scripts, Ferry [47], there is already a relatively comprehensive summary of the classification of data constraints. Building on this, besides **Type**, **Format**, **Range**, **Order** and **Exception** (i.e., **Missing Value** in Ferry), ViseGPT's classification framework has also undergone certain adjustments:

- **Unique** verifies that each value in the column appears only once (e.g., an ID column used as a primary key).
- **Forbidden Value** requires that certain specific values must not appear in the data.
- **Relation** describes the dependencies between columns and is further elaborated based on the different data types involved. For example, common equality relationships, magnitude relationships for Integer/Float data (e.g., greater than/less than), and substring/superstring relationships for Character data. Besides these basic relation categories, we enable the use of natural language labels to flexibly describe more complex dependencies.

Under the aforementioned constraint classification framework, the analysis program can extract user requirements from the prompt for subsequently testing the generated script.

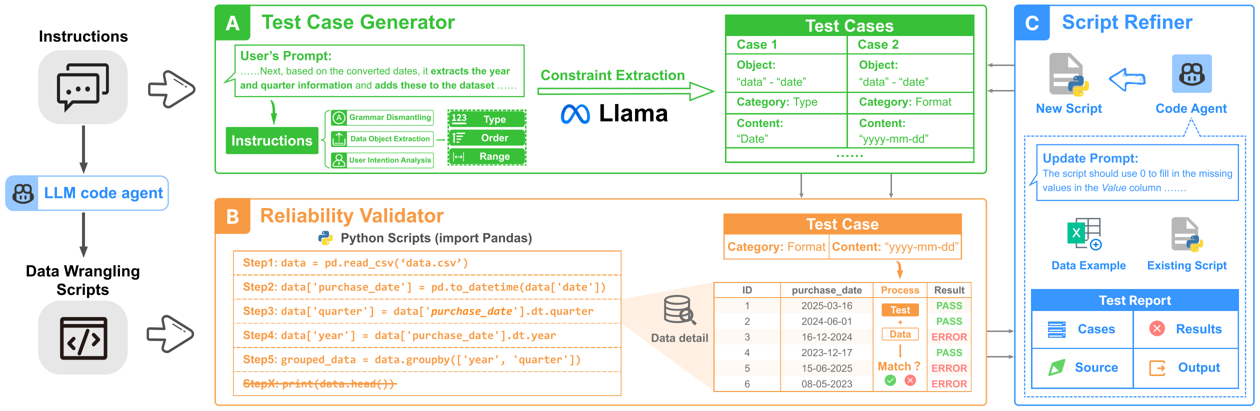
To evaluate the correctness of data wrangling scripts generated by LLMs within our constraint framework and to demonstrate the changes in data correctness throughout the script flow, it is necessary to separate the scripts step-by-step. Then ViseGPT generates test input datasets that meet the prompt requirements and script variable parameters, and systematically match output data results with test cases. Finally, ViseGPT summarizes the test results.

Separating the Scripts Step-by-Step. After sending the user's natural language instructions to LLM embedded in ViseGPT, the system generates the corresponding data wrangling script. ViseGPT can then break down the script into a series of atomic steps forming a chain and ignore any redundant or unnecessary repetitive steps (e.g., steps like print and assert that do not affect the data). The system will use AST parsing based on Python syntax to extract the input-output elements (i.e., tables and columns) from each atomic step, aggregate and collect them for subsequent visualization.

Constraint Matching Testing. ViseGPT executes test datasets through LLM-generated data wrangling scripts to verify if each step's output meets the specified constraint test cases. Users can choose to upload their own test datasets, and the system will also generate a dataset that aligns as closely as possible with the task description based on the user's prompt. During the testing process, a static match will first be conducted to determine whether the output data follows the basic test cases. For instance, during Type inspection, regular expressions can be utilized to ensure that the data format matches specified characteristics; for Range testing, checks are performed to ascertain whether numerical values or string lengths fall within designated intervals. To achieve sufficient coverage, ViseGPT also supports testing with LLMs in the categories of **Format**, **Forbidden Value** and **Relation**. Besides employing static matches, we also leverage natural language descriptions for constraints, utilizing LLMs to assess whether the output data conforms to these described requirements.

4.3 Visualizing Unit Test Flow

After test execution, in order to create an intuitive overview of the test report for locating the source of issues and assisting debugging, it is essential to use an appropriate method to visualize the script's data transformation process. Our design considered table and tree alternatives that leverage different structures to encode data flows, but the feedback received in iterative progress showed that tables obscure temporal dependencies while trees lack intuitive flow representation. In contrast, Gantt charts offer a stronger balance of information density, temporal clarity, and workflow mapping, with particular advantages in summarizing process information [63], making them well-suited to our application scenario. Hence, a design based on Gantt charts is introduced in ViseGPT, following a user-centered development process that has been iteratively refined. In this framework, shape and color coding are used to generalize the test results, with an "aligned-to-line" design that segments scripts at the granularity of individual steps to align Gantt chart rows with LLM outputs side by side, helping users validate generated code.



```
Step1: data = pd.read_csv('data.csv')
Step2: data['purchase_date'] = pd.to_datetime(data['date'])
Step3: data['quarter'] = data['purchase_date'].dt.quarter
Step4: data['year'] = data['purchase_date'].dt.year
Step5: grouped_data = data.groupby(['year', 'quarter'])
~~StepX: print(data.head())~~
```

ID	purchase_date	Result
1	2025-03-16	PASS
2	2024-06-01	PASS
3	16-12-2024	ERROR
4	2023-12-17	PASS
5	15-06-2025	ERROR
6	08-05-2023	ERROR

Figure 4: ViseGPT Workflow. (A) Test Case Generatorgenerates test cases based on data constraints extracted from users' prompts; (B) Reliability Validator executes the data wrangling scripts generated by the LLMs code agent, matches the output data results with test cases, and produces test reports; (C) Script Refiner can iteratively modify scripts by sending test reports, natural language instructions, and data examples as prompts to LLMs.

Visualization of the data transformation process. In Section 4.2, during unit test execution, ViseGPT has already separated the scripts step-by-step, simplifying the script content and extracting the main operations that impact the test results. Next, a tailored Gantt chart is applied in ViseGPT to visualize the script execution process, facilitating clear presentation of test results. The step ID is shown on the left, aligning with the corresponding lines of scripts to create an intuitive visual mapping. Specifically, we represent the data columns or global variables involved in each step as rectangles, using different legends to indicate whether a variable is an input or output. Variables appearing in the same step are connected by lines, allowing users to quickly locate when issues arise. The chart is arranged from top to bottom, aligning with the execution flow of the script. ViseGPT focuses on sequential execution and does not support branching or loops. This design aligns with typical data wrangling scripts, where batch operations are inherently linear and account for the majority of transformations. In the future, support for nonlinear logic can be expanded. In order to save space, column and variable names at the top are displayed at an angle. While step-based granularity ensures alignment with scripts, we acknowledge there are scalability challenges for long scripts and believe exploring hierarchical block-level abstractions and more detailed single-step interaction is meaningful (Section 6.3).

Visualization of unit test results. Building on the visualization of the script's data transformation process, the system can also present the execution process and final results of unit testing. Different rectangle colors indicate the test pass status of a variable at each step: green represents that the script's output fully meets the constraints of the test cases, while red indicates conflicts. At the bottom of the summary chart, the final test results are displayed (Figure 2 F1), including the number of failed test cases and the test pass rate (the number of passed test cases divided by the total number of test cases). Users can click on the rectangles to view details for a variable at a specific step and customize the test set (**D3**). The detail view that pops up on the right includes variable parameters, step ID, test cases, and detailed data. When a rectangle is clicked, the corresponding script line and chart row will be emphasized, and the variable name in the script will be highlighted.

4.4 Optimizing the Script Iteration Workflow

After presenting the overview of test results, ViseGPT provides the corresponding features when users want to view the test cases and data details to make revisions for continued iteration.

Presentation and customization of test cases. A detailed list of test cases for the corresponding variable is placed in the detail panel (Figure 2 D). The icons on the left side of the list represent different categories, while their colors indicate whether the test case has passed (Figure 2 D1). Clicking an icon reveals detailed information on the right, including a brief description and an explanation about the test case. Below the test case content, there are four buttons that enable custom modifications, sending test reports, and filtering conflicting data, enhancing the debugging and script analysis capabilities (Figure 2 D2). After customizing the test cases, clicking the refresh button (Figure 2 G) at the top will trigger the system to re-run the tests and display an updated test report.

Inspection of single-step detailed data. Below the detailed list of test cases, the default view displays the output of the column at a specific step (Figure 2 E). After applying the data filter in the test case list, only the data associated with failed test cases will be shown. However, inspecting a single column in isolation is not intuitive in complex scenarios involving multi-column and multi-variable transformations. For seeing several columns at the same time, users can click the magnify button in the upper right corner (Figure 2 E1) to view the entire data tables for the current step.

Iteration of the script based on test reports. In daily programming tasks, when encountering system errors while working in a local compiler, it has become a common practice to directly copy the error message and send it to LLMs for analysis and resolution. ViseGPT supports sending test reports as prompts for code iteration (Figure 2 C1). The test report uploaded to the backend includes test case details, test results, script lines, and output data, allowing for more targeted problem-solving within our testing framework.

4.5 Implementation

ViseGPT is a web application, whose front-end is built on the React [69] framework with TypeScript. The back-end utilizes the Llama-3.3-70b-Instruct model hosted by NVIDIA. Based on this model, we have further developed the test case generator. Note that we select this model as a feasible solution, and the system's tooling is largely model-agnostic, designed to work with various LLM outputs. We encourage future research to investigate alternative LLM architectures and their impacts on performance in depth. After the LLM code agent generates a data wrangling script, the system backend parses the prompt to analyze its semantics and processes it into test cases. The backend with WebSockets [61] can progressively stream output to the frontend, allowing script content and test results to be dynamically displayed in real-time.

5 Evaluation

We evaluate ViseGPT through a user study with 18 participants (P1-18) of various backgrounds. The evaluation explores the following four research questions.

- How does ViseGPT improve the efficiency of debugging data wrangling scripts generated by LLMs?
- What impact does ViseGPT have on users' accuracy in evaluating LLM-generated data scripts?
- How does ViseGPT assist users in iterating on LLM-generated data wrangling scripts after identifying issues?
- How can ViseGPT be optimized to enhance its practical utility for debugging LLM-generated data wrangling scripts?

5.1 Experiment Settings

We conducted a comparative study with 18 participants to debug four LLM-generated data wrangling scripts between ViseGPT and Baseline. After completing tasks, we analyzed their performance and collected subjective feedback with the within-subjects design.

Participants. We recruited 18 participants (15 males, 3 females; ages 20-26, M = 22.17, SD = 1.69) through campus forums, social media and promotional referrals. Participants comprised 12 undergraduates, 4 master students, and 2 PhD students majoring in computer science, robotics engineering, bioinformatics, etc. According to their self-ratings on a 5-point Likert scale (1: lowest extent, 5: greatest extent), they were experienced with the Pandas syntax used in ViseGPT (M = 3.28, SD = 0.57) and data wrangling (M = 3.17, SD = 0.92). Additionally, they had extensive experience in using LLM-powered natural language-based conversational interactions products (M = 4.33, SD = 0.59). In summary, all participants met the participation requirements for the user evaluation.

Baseline. We removed the extended views and associated derivative features in ViseGPT, designating the experimental group that subsequently used this system for debugging as Baseline. In other words, the Baseline system is functionally equivalent to LLM-powered dialogue tools (e.g., ChatGPT) that participants commonly interact with. To minimize variables and enhance the reliability of experimental conclusions, Baseline maintains consistent stylistic design and basic functionalities (e.g., conversation context preservation) with ViseGPT, while utilizing the same foundation model (i.e., Llama-3-70b-Instruct).

Tasks. The evaluation included four tasks (A-D) sampled from distinct work contexts (refer to supplementary material for details). For each task, we presented participants with requirements of data wrangling scripts and a corresponding Pandas script containing functional errors. Although these scripts were generated using LLM with the requirements as prompts, they were syntactically correct but deviated from the intended functionality. Participants were required to debug the scripts using either ViseGPT or Baseline by identifying and correcting discrepancies from requirements. All error cases in the experimental scripts were derived from real-world examples documented in our formative study (Section 3).

5.2 Procedure

We opted for a modified balanced Latin square design [13] to control for potential order effects of tool usage on task performance. In our experiment design, each tool (ViseGPT and Baseline) appeared exactly 9 times in each task (A-D), collectively covering all possible tool-task combinations (Total: 6 groups $\times$ 4 tasks $\times$ 3 rounds = 72 trials, 36 trials per tool and 9 trials per tool for each task). This approach enabled statistically robust comparisons of tool performance across multiple tasks while controlling for learning biases and individual differences. The complete experimental procedure lasted approximately 60 minutes.

Prior to experimental procedures, participants took part in a training session consisting of a system tutorial and a practice task, which took about 5 minutes. Subsequently, the participant worked on Task A-D sequentially, with each task allocated a 15-minute time limit. Participants ended each task when they either reached the time limit or completed the debugging. The entire task execution process was screen-captured for subsequent analysis. The evaluation ended with a semi-structured interview and a User Experience Questionnaire (UEQ) [65]. The interviews focused on gathering participants' overall impressions of tool performance across script debugging stages and eliciting improvement suggestions. We adopted the UEQ to measure participants' perceived experience with both ViseGPT and Baseline, and compared the results using the official UEQ data analysis tool [72]. In total, the post-experiment interview and questionnaire took no longer than 15 minutes. The participation fee was \$10 per hour.

5.3 Results

We compare task correctness, time spent and participants' subjective ratings between Baseline and ViseGPT. In addition, we analyze specific interaction behaviors of participants and report insights derived from the interview.

5.3.1 *Task correctness and spent time.*

Table 2 compares participant performance between ViseGPT and Baseline conditions across the four experimental tasks (A-D). The success rate, defined as the percentage of participants who successfully fixed the bug, was higher in the ViseGPT condition for Tasks A, B, and D. However, Task C showed comparable success rates between conditions (ViseGPT: 56% vs. Baseline: 22%). Furthermore, as shown in Figure 5, ViseGPT significantly reduced completion times for Tasks A, B, and D (Mean difference > 120s), while no substantial difference was observed for Task C (ViseGPT: M = 625s vs. Baseline: M = 645s).

Table 2: The success rate (%) and No. query submissions to LLMs in ViseGPT and Baseline for Task A-D (N=9/condition). The issue column describes the mistake made by LLMs in the task. #CR: No. clicking rectangles in ViseGPT. #QT: No. query submissions with test reports in ViseGPT. "(Value)": standard deviance.

Task	Issue	#CR	#QT	Average Query		Success (%)	
				ViseGPT	Baseline	ViseGPT	Baseline
A	Length mismatch in string	7.56 (6.45)	1.33 (0.67)	2.00 (0.94)	3.33 (2.26)	100 (0.00)	78 (0.42)
B	Missing boundary in grouping	3.89 (3.38)	1.33 (0.47)	1.78 (1.03)	4.11 (1.97)	100 (0.00)	33 (0.47)
C	Inconsistent letter casing	9.67 (8.38)	1.78 (0.79)	2.33 (0.67)	3.11 (1.97)	56 (0.50)	22 (0.42)
D	Non-compliant rounding	8.89 (5.07)	2.00 (0.94)	3.00 (1.63)	2.78 (1.75)	89 (0.31)	11 (0.31)

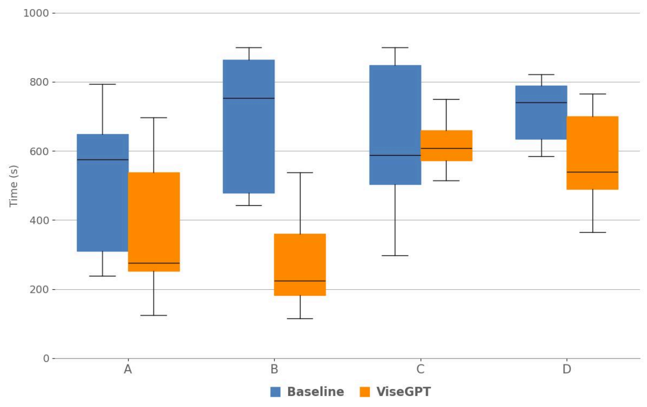


Figure 5: Completion time in four tasks.

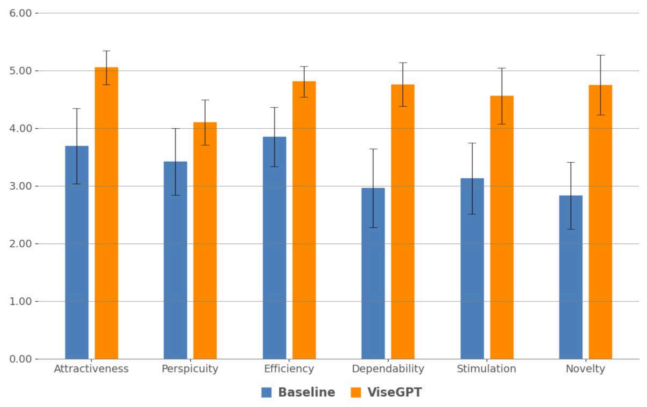


Figure 6: Rating results in UEQ.

We analyze the factors underlying ViseGPT's lower performance in Task C. The task required participants to calculate global influence scores for **five specified colors** in images. The raw data used standard case-insensitive #RRGGBB color codes, while the initial script lacked case normalization. Consequently, the color sorting operation treated differently-cased representations (e.g., #FF0000 and #ff0000) as distinct colors, resulting in erroneous counts exceeding the five target colors. First, ViseGPT successfully generated test cases to verify that the `rank_label` column (representing color influence rankings) should contain integers between 1 and 5, as specified in the prompt. Consequently, the system correctly flagged issues when `rank_label` values exceeded 5. We observed that although all participants detected this issue, their subsequent analysis of the issues and the prompts they submitted for script regeneration impacted ViseGPT's performance. A minority of participants (3/9) promptly identified the range issues arising from duplicate color code counts. By directly instructing ViseGPT to normalize color code casing, they resolved the issue. However, the majority of participants (6/9) opted to describe the situation directly to ViseGPT (e.g.,*"The rank_label column contained invalid values exceeding the upper bound of 5."*) without addressing the underlying cause of color duplication. This led ViseGPT to implement suboptimal solutions, such as using `pd.head()` to limit output to five rows, instead of standardizing the color format. This resulted in extended debugging cycles and reduced success rates.

5.3.2 Subjective ratings. After UEQ investigation, we used Shapiro-Wilk tests to test normality (all $p > 0.05$), and used Student's paired t-tests ($\alpha = 0.05$). The results demonstrated that ViseGPT was significantly better ($p < 0.05$) in all six dimensions compared with Baseline (Figure 6).

5.3.3 Specific interaction behaviors. We recorded three interaction metrics during the study (Table 2): (1) the number of queries sent, (2) the number of queries with test reports attached in ViseGPT, and (3) counts of clicking rectangles in the tailored Gantt chart.

Our analysis revealed that participants preferred attaching test reports when submitting prompts in ViseGPT across all four tasks, with this behavior accounting for over 60% of total query submissions when using ViseGPT. The situation demonstrates that participants in the ViseGPT workflow showed a preference for uploading test results as supplementary material when submitting prompts. We infer that this extended functionality reduces users' manual issue-summarizing efforts, while its interactive usability could potentially enhance confidence in script modifications.

We also observed that in all the tasks except Task D, participants sent more prompts in the Baseline condition than ViseGPT. With ViseGPT's intuitive visual debugging interface and advanced interactions, participants were typically able to obtain correct scripts within fewer iterations. For Task D, we noticed that two participants (P17 and P18) using the Baseline requested early termination after sending just one prompt, but their final scripts still contained issues. During interviews, we specifically inquired about this situation.

Both stated that they ended the task because the Baseline system indicated the provided script had no issues, and they detected no problems during manual inspection.

5.3.4 *Overall impressions around two stages.*

Upon completion of all tasks, we conducted semi-structured interviews to evaluate participants' experiences. The interviews specifically focused on participants' debugging processes across two critical stages: (1) script verification and (2) code modification implementation.

About verifying script correctness and analyzing issues. The majority of participants (16/18) reported that ViseGPT's visualization provided more intuitive debugging assistance compared to Baseline's generic responses, facilitating faster issue identification. *"I can directly click the red rectangles (in the tailored Gantt chart) to view failed test cases rather than reviewing the entire script from scratch."* (P12) Additionally, participants (12/18) appreciated the automatically generated test cases for their logical categorization, which enhanced debugging precision. Notably, some of participants (3/18) raised concerns about potential undetected issues persisting in scripts despite full test case validation. *"Due to inherent limitations in constrained data description methods, the test cases may lack rigor and fail to achieve full coverage."* (P3) And participants (P2 and P9) noted that compared to Baseline's textual explanations, ViseGPT offered less flexibility in elucidating script logic.

About modifying the scripts iteratively. Participants (17/18) reported that ViseGPT enabled context-aware rapid corrections, significantly boosting their confidence in script modifications. The auto-generated test cases (7/18) and test report attachment feature (14/18) reduced descriptive burden during debugging. However, some users (6/18) noted that Baseline's free-form questioning allowed direct explanation of modification logic, whereas ViseGPT's revision process was harder to track. *"Although I didn't comprehend the source of issues, ViseGPT successfully corrected based solely on my description of the situation - but the lack of explanatory feedback left me somewhat confused about the resolution process. "* (P14)

5.3.5 *Suggestions from participants.*

To enhance ViseGPT's practical utility, we collected optimization suggestions from participants during user studies.

Expand test case categories and severity classification. Beyond the 8 predefined test case categories of ViseGPT, participants recommended expanding more categories (6/18) and even support user-defined test case category (3/18) to improve test coverage and flexibility. Moreover, some participants (2/18) expressed that the current color-coding scheme - using red for issues and green for passed tests - appears overly simplistic for complex debugging scenarios. *"ViseGPT could adopt modern compiler conventions by classifying less severe issues as warnings and marking them in yellow, thereby developing a three-tier status indicator system."* (P17) With severity classification, users can prioritize debugging efforts, focusing first on critical issues while deprioritizing minor warnings.

Implement data causality analysis to enhance tracing. Participants (P5, P10 and P13) pointed out that ViseGPT's insufficient presentation of dataflow causality made it difficult to trace the transformation process of erroneous data during debugging. For example, in Task D, the erroneous value 20.5 in the `purchase_amount` column was originally positioned mid-column. However, after executing `df.sort_values(by='purchase_amount'`, the value relocated to the column's final row. *"Tracing the data transformation process consumed significant time - while I could identify the erroneous output, determining its origin proved still challenging."* (P10) Therefore, in addition to the existing single-step data display, providing visualization of erroneous data's transformation path across steps would improve debugging efficiency.

6 Discussion

This paper introduces ViseGPT, a novel tool for enhancing the efficiency of users in debugging LLM-generated data wrangling scripts, aiming to better align the scripts with user prompts. User evaluation revealed that using ViseGPT makes issue validation in scripts more intuitive and boosts confidence in correctness after modifications (Section 5.3.4). In this section, we discuss the implications of the system design (Section 6.1) and outline the limitations of the research as well as future work directions (Section 6.3).

6.1 Design Implications

Throughout the design and development of ViseGPT, we have gained valuable implications beyond the system itself.

Implementing a "testing-as-conversation" interaction model to refine debugging workflow. While traditional testing remains a post-development phase [53], ViseGPT compresses the entire data wrangling script development process into a simplified workflow, using a "testing-as-conversation" paradigm [28]. This design approach adheres to M. Fowler's "Continuous Integration" principle [22], which is one of the core practices in Agile development [67]. First, developers can immediately transform failing cases into debugging clues through interactive workflows. *"I believe that even users with limited (data wrangling script) programming expertise can derive novel insights into script debugging with ViseGPT."* (P11) Based on this perspective, ViseGPT adapts to users with varying levels of expertise in data wrangling script development.

Establishing a framework of constraints enhances clarity and boosts users' confidence. ViseGPT systematically extracts and validates constraints from natural language prompts. By making expectations explicit, such a framework reduces ambiguity in both the user intent and the behavior of generated scripts. This approach aligns with the "Specification as Code" paradigm [4, 18] from formal methods. Interacting with a well-structured testing framework can enhance users' confidence in the final results [45]. *"The end-to-end process—from sending a prompt, to generating results, to validating results with the original prompt—serves as a confidence bridge between me and ViseGPT."* (P16) The design tries to expand the trust boundaries of Human-LLM collaboration.

Enhancing debug reasoning through visualization. Observing how users debug LLM-generated scripts (Section 5.3.3), we found that visualization in ViseGPT serves not just as an infomation presentation medium, but as a central mechanism for supporting logical reasoning and identifying issues. Unlike conventional debugging tools that provide textual descriptions of results and counts of bugs [37], the tailored Gantt chart summarizes the execution process, enabling users to quickly identify critical steps and spot issues. *"During debugging, observing visual elements is more engaging and effective than focusing on code. "* (P7) This opinion serves as validation for attraction of our visualization design. However, we should remain mindful of its inherent limitations, possibly introducing new issues rather than resolving them (Section 5.3.1).

6.2 Aligned Visualizations for AI Explainability

Building upon our visualization design for script debugging (Section 4), we argue that the aligned-to-output-line paradigm can bridge the "gulf of evaluation" in Human-LLM collaboration across domains [70]. Norman's gulfs of execution and evaluation [58] highlight the cognitive gap between system outputs and user interpretation, which is also a challenge acutely present in LLM interactions where users struggle to trace how outputs relate to their intent. ViseGPT's Gantt chart displays that structural alignment between output components (code lines) and their visual explanations (test results) enhances interpretability. The key point of this approach lies in its ability to externalize the implicit relationships between input prompts, intermediate reasoning, and final outputs. Our participant feedback (Section 5.3.4) supports that such spatial organization aids pattern recognition and issue localization, which is transferable to scenarios requiring stepwise validation of LLM outputs.

In conclusion, the "aligned-to-output-line" strategy offers a scaffold for cross-domain explainability standards. By adopting this paradigm, tools could mitigate the "black-box" effect of LLMs, transforming opaque reasoning chains into inspectable, interactive artifacts [16]. We envision extensions of this work adapting the alignment principle to multimodal outputs (e.g., image-text pairs) or collaborative settings where visual explanations serve as shared referents for team debugging—ultimately advancing the goal of human-AI interaction transparency.

6.3 Limitations and Future Work

This section examines three key limitations of the ViseGPT system in data wrangling script debugging and proposes potential directions for future work.

Step Granularity Trade-offs. The step-based segmentation in ViseGPT follows an "aligned-to-line" design (Section 4.3). However, this granularity may introduce scalability limitations: long scripts suffer from reduced readability due to vertical sprawl, while complex single-step operations (e.g., multi-column transformations) may be hard to understand in a step-level view. Future work can investigate a framework with hierarchical abstractions, balancing coarser block-level summaries for scalability with finer sub-step breakdowns for intricate operations.

Task Representativeness and Complexity. In the evaluation, the four tasks (Section 6.1) were selected from formative study to ensure practical relevance and they focus on common violations (e.g., formats, ranges) that directly manifest in output data, as such issues represent high-frequency pain points identified. However, we acknowledge that real-world data wrangling scripts often involve more complex situations. The current evaluation demonstrates ViseGPT's effectiveness in handling common debugging scenarios, but future work should expand to include more diverse and sophisticated issue types to further validate the versatility.

Test Case Coverage and Flexibility. While ViseGPT's automated test case generation improves debugging accuracy efficiency, its current framework is limited to predefined constraint categories as a concern expressed by the participants in Section 5.3.4. This may not fully capture nuanced or domain-specific requirements, leading to potential gaps in test coverage. Future work could expand the system's flexibility by allowing users to define custom test case categories or integrate dynamic learning [88] from iterative feedback, thereby enhancing adaptability to diverse scenarios.

Language and Framework Generalization. Currently, ViseGPT primarily supports Python-based data wrangling scripts, limiting its applicability to users working with other programming languages (e.g., R, SQL, or Julia). Future research should explore extending the system's requirement analysis and test generation capabilities to additional languages, as well as integrating with domain-specific frameworks. This expansion would broaden ViseGPT's utility across different programming ecosystems.

Debugging Transparency and Workflow Integration. The tool's visualization effectively highlights test failures but lacks detailed tracing of data causality across script steps (Section 5.3.5), which can complicate root cause analysis. Additionally, its standalone nature may disrupt existing workflows, such as integration with IDEs or version control systems. Future work could enhance debugging transparency with interactive dataflow graphs and improve usability through seamless integration with popular development environments (e.g., Jupyter Notebook, VS Code) and collaborative features for team-based debugging.

7 Conclusion

This paper presents ViseGPT, a tool that addresses the challenge of debugging LLM-generated data wrangling scripts by achieving better alignment between natural language prompts and results. Through automatically extracting constraints from user instructions and generating systematic test cases, ViseGPT enables users to efficiently verify whether generated scripts align with their requirements without requiring deep programming expertise. The system's constraint-based validation approach, combined with its intuitive Gantt-chart visualization and interactive debugging features, provides users with actionable insights to identify and resolve script issues. Our user evaluation (N=18) demonstrates that ViseGPT improves the efficiency and accuracy of debugging LLM-generated data wrangling scripts, making complex tasks more accessible. This work advances the field of human-AI collaboration by offering a practical solution that enhances both the trustworthiness and iterability of AI-generated code, ultimately empowering users to harness the full potential of LLMs with greater confidence.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments. This work was supported in part by Zhejiang Provincial Natural Science Foundation of China under Grant No. LD25F020003, in part by NSFC (U22A2032, 62402421), and in part by Ningbo Yongjiang Talent Programme (2024A-399-G).

References

[1] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3test: Assertion-augmented automated test case generation. *Information and Software Technology* 176 (2024), 107565. doi:10.1016/j.infsof.2024.107565
[2] Aleftina Andrianova and Vladimir Itsykson. 2013. Generating unit tests using static analysis and contracts. In *Proceedings of Tools & Methods of Program Analysis*. 83–88. doi:10.1109/TMPA.2013.7163725

[3] Anthropic. 2025. Claude 3.7 Sonnet and Claude Code. Retrieved February 25, 2025 from https://www.anthropic.com/news/claude-3-7-sonnet

[4] S. Antoy, P. Forcheri, and M.T. Molfino. 1990. Specification-based code generation. In *Proceedings of Annual Hawaii International Conference on System Sciences*. 165–173. doi:10.1109/HICSS.1990.205185

[5] R. Barták. 1999. Constraint Programming: What is Behind. In *Proceedings of Workshop on Constraint Programming for Decision and Control*. 7–15.

[6] Hongbo Chen, Yifan Zhang, Xing Han, Huanyao Rong, Yuheng Zhang, Tianhao Mao, Hang Zhang, XiaoFeng Wang, Luyi Xing, and Xun Chen. 2024. WitheredLeaf: Finding Entity-Inconsistency Bugs with LLMs. https://arxiv.org/abs/2405.01668

[7] Weihao Chen, Xiaoyu Liu, Jiacheng Zhang, Ian Iong Lam, Zhicheng Huang, Rui Dong, Xinyu Wang, and Tianyi Zhang. 2023. MIWA: Mixed-Initiative Web Automation for Better User Control and Confidence. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery. https://doi.org/10.1145/3586183.3606720

[8] Wei-Hao Chen, Weixi Tong, Amanda Case, and Tianyi Zhang. 2025. Dango: A Mixed-Initiative Data Wrangling System using Large Language Model. In *Proceedings of CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery. https://doi.org/10.1145/3706598.3714135

[9] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of ACM International Conference on the Foundations of Software Engineering*. 572–576. doi:10.1145/3663529.3663801

[10] Yibin Chen, Yifu Yuan, Zeyu Zhang, Yan Zheng, Jinyi Liu, Fei Ni, Jianye Hao, Hangyu Mao, and Fuzheng Zhang. 2025. SheetAgent: Towards a Generalist Agent for Spreadsheet Reasoning and Manipulation via Large Language Models. In *Proceedings of ACM on Web Conference*. Association for Computing Machinery, 158–177. https://doi.org/10.1145/3696410.3714962

[11] Anindya Das Antar, Somayeh Molaei, Yan-Ying Chen, Matthew L Lee, and Nikola Banovic. 2024. VIME: Visual Interactive Model Explorer for Identifying Capabilities and Limitations of Machine Learning Models for Sequential Decision-Making. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*. doi:10.1145/3654777.3676323

[12] Tamraparni Dasu and Theodore Johnson. 2003. Exploratory Data Mining and Data Cleaning. John Wiley & Sons, Inc., New York, NY.

[13] A.M. Dean. 2001. Experimental Design: Overview. In *International Encyclopedia of the Social & Behavioral Sciences*. 5090–5096. doi:10.1016/B0-08-043076-7/00417-4

[14] Ximing Dong, Dayi Lin, Shaowei Wang, and Ahmed E. Hassan. 2024. A Framework for Real-time Safeguarding the Text Generation of Large Language Model. https://arxiv.org/abs/2404.19048

[15] Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2025. CodeScore: Evaluating Code Generation by Learning Code Execution. *ACM Transactions on Software Engineering and Methodology* 34, 3 (2025). doi:10.1145/3695991

[16] Finale Doshi-Velez and Been Kim. 2017. Towards A Rigorous Science of Interpretable Machine Learning. https://arxiv.org/abs/1702.08608

[17] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 1–12. https://doi.org/10.1145/3313831.3376442

[18] David Evans, John Guttag, James Horning, and Yang Meng Tan. 1994. LCLint: a tool for using specifications to check code. *ACM SIGSOFT Software Engineering Notes* (1994), 87–96. doi:10.1145/195274.195297

[19] Yingchaojie Feng, Xingbo Wang, Bo Pan, Kam Kwai Wong, Yi Ren, Shi Liu, Zihan Yan, Yuxin Ma, Huamin Qu, and Wei Chen. 2024. XNLI: Explaining and Diagnosing NLI-Based Visual Data Analysis. *IEEE Transactions on Visualization and Computer Graphics* 30, 7 (2024), 3813–3827. doi:10.1109/TVCG.2023.3240003

[20] Kasra Ferdowsi, Jack Williams, Ian Drosos, Andrew D. Gordon, Carina Negreanu, Nadia Polikarpova, Advait Sarkar, and Benjamin Zorn. 2023. COLDECO: An End User Spreadsheet Inspection Tool for AI-Generated Code. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*. 82–91. doi:10.1109/VL-HCC57772.2023.00017

[21] Afonso Fontes, Gregory Gay, Francisco Gomes de Oliveira Neto, and Robert Feldt. 2023. Automated support for unit test generation. In *Optimising the Software Development Process with Artificial Intelligence*. 179–219. doi:10.1007/978-981-19-9948-2_7

[22] Martin Fowler and Matthew Foemmel. 2006. Continuous integration.

[23] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology* 24, 2, Article 8 (2014). doi:10.1145/2685612

[24] Google. 2024. Gemini Advanced: Release updates. Retrieved December 17, 2024 from https://gemini.google.com/updates

[25] Vitor Guilherme and Auri Vincenzi. 2023. An initial investigation of ChatGPT unit test generation capability. In *Proceedings of Brazilian Symposium on Systematic and Automated Software Testing*. 15–24. doi:10.1145/3624032.3624035

[26] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Transactions on Management Information Systems* 43, 2, Article 42 (2025). doi:10.1145/3703155

[27] Yihao Huang, Chong Wang, Xiaojun Jia, Qing Guo, Felix Juefei-Xu, Jian Zhang, Geguang Pu, and Yang Liu. 2024. Semantic-guided Prompt Organization for Universal Goal Hijacking against LLMs. https://arxiv.org/abs/2405.14189

[28] Anita M. Hubley. 2017. Expanding Views on Response Processes Evidence for Validity. *Measurement: Interdisciplinary Research and Perspectives* (2017), 140–142. doi:10.1080/15366367.2017.1404366

[29] IEEE. 1990. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990* (1990), 1–84. doi:10.1109/IEEESTD.1990.101064

[30] Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. 2024. TestGenEval: A Real World Unit Test Generation and Test Completion Benchmark. https://arxiv.org/abs/2410.00752

[31] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. 2023. Towards Mitigating LLM Hallucination via Self Reflection. In *Proceedings of Findings of the Association for Computational Linguistics: EMNLP*. 1827–1843. doi:10.18653/v1/2023.findings-emnlp.123

[32] Fengqing Jiang, Zhangchen Xu, Luyao Niu, Bill Yuchen Lin, and Radha Poovendran. 2025. ChatBug: A Common Vulnerability of Aligned LLMs Induced by Chat Templates. https://arxiv.org/abs/2406.12935

[33] Sean Kandel, Jeffrey Heer, Catherine Plaisant, Jessie Kennedy, Frank van Ham, Nathalie Henry Riche, Chris Weaver, Bongshin Lee, Dominique Brodbeck, and Paolo Buono. 2011. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization* (2011), 271–288. doi:10.1177/1473871611415994

[34] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 3363–3372. doi:10.1145/1978942.1979444

[35] Aryan Keluskar, Amrita Bhattacharjee, and Huan Liu. 2024. Do LLMs Understand Ambiguity in Text? A Case Study in Open-world Question Answering. In *Proceedings of IEEE International Conference on Big Data*. 7485–7490. doi:10.1109/BigData62323.2024.10825265

[36] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. SPoC: Search-based Pseudocode to Code. In *Proceedings of Advances in Neural Information Processing Systems*, Vol. 32. https://proceedings.neurips.cc/paper_files/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf

[37] Rob Law. 1997. An overview of debugging tools. *ACM SIGSOFT Software Engineering Notes* (1997), 43–47. doi:10.1145/251880.251926

[38] Sera Lee, Dae R. Jeong, Junyoung Choi, Jaeheon Kwak, Seoyun Son, Jean Y Song, and Insik Shin. 2024. SERENUS: Alleviating Low-Battery Anxiety Through Real-time, Accurate, and User-Friendly Energy Consumption Prediction of Mobile Applications. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*. doi:10.1145/3654777.3676437

[39] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of SIGCHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 1719–1728. https://doi.org/10.1145/1357054.1357323

[40] Hongxin Li, Jingran Su, Yuntao Chen, Qing Li, and ZHAO-XIANG ZHANG. 2023. SheetCopilot: Bringing Software Productivity to the Next Level through Large Language Models. In *Proceedings of Advances in Neural Information Processing Systems*, Vol. 36. Curran Associates, Inc., 4952–4984.

[41] Xue Li and Till Döhmen. 2024. Towards Efficient Data Wrangling with LLMs using Code Generation. In *Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning*. 62–66. doi:10.1145/3650203.3663334

[42] Chi-Liang Liu, Hung yi Lee, and Wen tau Yih. 2022. Structured Prompt Tuning. https://arxiv.org/abs/2205.12309

[43] Lei Liu, So Hasegawa, Shailaja Keyur Sampat, Maria Xenochristou, Wei-Peng Chen, Takashi Kato, Taisei Kakibuchi, and Tatsuya Asai. 2024. AutoDW: Automatic Data Wrangling Leveraging Large Language Models. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. 2041–2052. doi:10.1145/3691620.3695267

[44] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D. Gordon. 2023. "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of CHI Conference on Human Factors in Computing Systems*. doi:10.1145/3544548.3580817

[45] Shaoying Liu, Tetsuo Tamai, and Shin Nakajima. 2011. A framework for integrating formal specification, review, and testing to enhance software reliability. *International Journal of Software Engineering and Knowledge Engineering* (2011), 259–288.

[46] Ziao Liu, Xiao Xie, Moqi He, Wenshuo Zhao, Yihong Wu, Liqi Cheng, Hui Zhang, and Yingcai Wu. 2025. Smartboard: Visual Exploration of Team Tactics with LLM Agent. *IEEE Transactions on Visualization and Computer Graphics* 31, 1 (2025), 23–33. doi:10.1109/TVCG.2024.3456200

[47] Zhongsu Luo, Kai Xiong, Jiajun Zhu, Ran Chen, Xinhuan Shu, Di Weng, and Yingcai Wu. 2025. Ferry: Toward Better Understanding of Input/Output Space for Data Wrangling Scripts. *IEEE Transactions on Visualization and Computer Graphics* 31, 1 (2025), 1202–1212. doi:10.1109/TVCG.2024.3456328

[48] Qianou Ma, Weirui Peng, Chenyang Yang, Hua Shen, Kenneth Koedinger, and Tongshuang Wu. 2024. What Should We Engineer in Prompts? Training Humans in Requirement-Driven LLM Use. https://arxiv.org/abs/2409.08775

[49] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of Annual ACM Symposium on User Interface Software & Technology*. Association for Computing Machinery, 291–301. https://doi.org/10.1145/2807442.2807459

[50] Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. 2024. LLM Critics Help Catch LLM Bugs. https://arxiv.org/abs/2407.00215

[51] Behrang Mehrparvar and Sandro Pezzelle. 2024. Detecting and Translating Language Ambiguity with Multilingual LLMs. In *Proceedings of Workshop on Multilingual Representation Learning*. 310–323. doi:10.18653/v1/2024.mrl-1.26

[52] Robert C. Miller and Brad A. Myers. 2001. Outlier finding: focusing user attention on possible errors. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, 81–90. https://doi.org/10.1145/502348.502361

[53] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. 2009. Test coverage and post-verification defects: A multiple case study. In *Proceedings of International Symposium on Empirical Software Engineering and Measurement*. 291–301. doi:10.1109/ESEM.2009.5315981

[54] F. MODUGNO and B.A. MYERS. 1997. Visual Programming in a Visual Shell—A Unified Approach. *Journal of Visual Languages & Computing* 8, 5 (1997), 491–522. doi:10.1006/jvlc.1997.0049

[55] Dominik Moritz, Chenglong Wang, Greg L. Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2019), 438–448. doi:10.1109/TVCG.2018.2865240

[56] Zan Ahmad Naeem, Mohammad Shahmeer Ahmad, Mohamed Eltabakh, Mourad Ouzzani, and Nan Tang. 2024. RetClean: Retrieval-Based Data Cleaning Using LLMs and Data Lakes. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4421–4424. doi:10.14778/3685800.3685890

[57] Arpit Narechania, Adam Fourney, Bongshin Lee, and Gonzalo Ramos. 2021. DIY: Assessing the Correctness of Natural Language to SQL Systems. In *Proceedings of International Conference on Intelligent User Interfaces*. 597–607. doi:10.1145/3397481.3450667

[58] Donald A Norman. 1986. Cognitive engineering. *User centered system design* 31, 61 (1986), 2.

[59] OpenAI. 2024. Data analysis with ChatGPT. Retrieved June 1, 2024 from https://help.openai.com/en/articles/8437071-data-analysis-with-chatgpt

[60] Anthony Peruma, Taryn Takebayashi, Rocky Huang, Joseph Carmelo Averion, Veronica Hodapp, Christian D. Newman, and Mohamed Wiem Mkaouer. 2024. On the Rationale and Use of Assertion Messages in Test Code: Insights from Software Practitioners. In *Proceedings of IEEE International Conference on Software Maintenance and Evolution*. 538–549. doi:10.1109/ICSME58944.2024.00055

[61] Luigi Pinca. 2025. WebSockets. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[62] Wanli Qian, Chenfeng Gao, Anup Sathya, Ryo Suzuki, and Ken Nakagaki. 2024. SHAPE-IT: Exploring Text-to-Shape-Display for Generative Shape-Changing Behaviors with LLMs. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*. doi:10.1145/3654777.3676348

[63] KK Ramachandran and KK Karthick. 2019. Gantt chart: An important tool of management. *International Journal of Innovative Technology and Exploring Engineering* 8, 7 (2019), 140–142.

[64] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[65] Martin Schrepp, Andreas Hinderks, and Jörg Thomaschewski. 2014. Applying the User Experience Questionnaire (UEQ) in Different Evaluation Scenarios. In *Design, User Experience, and Usability. Theories, Methods, and Tools for Designing the User Experience*. 383–392.

[66] Shreya Shankar, J.D. Zamfirescu-Pereira, Bjoern Hartmann, Aditya Parameswaran, and Ian Arawjo. 2024. Who Validates the Validators? Aligning LLM-Assisted Evaluation of LLM Outputs with Human Preferences. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*. doi:10.1145/3654777.3676450

[67] James Shore and Shane Warden. 2021. *The art of agile development.* " O'Reilly Media, Inc.".

[68] Nischal Shrestha, Titus Barik, and Chris Parnin. 2021. Unravel: A Fluent Code Explorer for Data Wrangling. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, 198–207. https://doi.org/10.1145/3472749.3474744

[69] Meta Open Source. 2024. React. https://react.dev/

[70] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive Science* 12, 2 (1988), 257–285. doi:10.1016/0364-0213(88)90023-7

[71] Xiaohang Tang, Sam Wong, Kevin Pu, Xi Chen, Yalong Yang, and Yan Chen. 2024. VizGroup: An AI-assisted Event-driven System for Collaborative Programming Learning Analytics. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*. doi:10.1145/3654777.3676347

[72] Team UEQ. 2024. UEQ. https://www.ueq-online.org/

[73] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268. doi:10.1145/3691620.3695501

[74] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268. doi:10.1145/3691620.3695501

[75] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of ACM/IEEE International Conference on Software Engineering*. 1398–1409. doi:10.1145/3377811.3380429

[76] James M. Wilson. 2003. Gantt charts: A centenary appreciation. *European Journal of Operational Research* 149, 2 (2003), 430–437. doi:10.1016/S0377-2217(02)00769-5

[77] Liwenhan Xie, Chengbo Zheng, Haijun Xia, Huamin Qu, and Chen Zhu-Tian. 2024. WaitGPT: Monitoring and Steering Conversational LLM Agent in Data Analysis with On-the-Fly Code Visualization. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*. Article 119. doi:10.1145/3654777.3676374

[78] Tao Xie and David Notkin. 2006. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering* 13, 3 (2006), 345–371. doi:10.1007/s10851-006-8530-6

[79] Kai Xiong, Siwei Fu, Guoming Ding, Zhongsu Luo, Rong Yu, Wei Chen, Hujun Bao, and Yingcai Wu. 2023. Visualizing the Scripts of Data Wrangling With Somnus. *IEEE Transactions on Visualization and Computer Graphics* (2023), 2950–2964. doi:10.1109/TVCG.2022.3144975

[80] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. On the Evaluation of Large Language Models in Unit Test Generation. https://arxiv.org/abs/2406.18181

[81] Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, Zhiyuan Liu, Xiaodong Shi, and Maosong Sun. 2024. MatPlotAgent: Method and Evaluation for LLM-Based Agentic Scientific Data Visualization. https://arxiv.org/abs/2402.11453

[82] Xin Yin, Chao Ni, Xiaodan Xu, and Xiaohu Yang. 2024. What You See Is What You Get: Attention-based Self-guided Automatic Unit Test Generation. https://arxiv.org/abs/2412.00828

[83] Andy Yu. 2023. *Improving Efficiency in Data Wrangling With Semantic Type Detection.* Ph. D. Dissertation. University of Hawai'i at Manoa.

[84] Zeyu Zhang, Paul Groth, Iacer Calixto, and Sebastian Schelter. 2024. Directions Towards Efficient and Automated Data Wrangling with Large Language Models. In *Proceedings of IEEE International Conference on Data Engineering Workshops*. 301–304. doi:10.1109/ICDEW61823.2024.00044

[85] Zhanhui Zhou, Man To Tang, Qiping Pan, Shangyin Tan, Xinyu Wang, and Tianyi Zhang. 2022. INTENT: Interactive Tensor Transformation Synthesis. In *Proceedings of Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery. https://doi.org/10.1145/3526113.3545653

[86] Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, Xiapu Luo, Jingzhu He, and Yutian Tang. 2024. Coverage Goal Selector for Combining Multiple Criteria in Search-Based Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 4 (2024), 854–883. doi:10.1109/TSE.2024.3366613

[87] Xinhua Zhu, Zhongjie Kuang, and Lanfang Zhang. 2023. A prompt model with combined semantic refinement for aspect sentiment analysis. *Information Processing & Management* 60, 5 (2023), 103462. doi:10.1016/j.ipm.2023.103462

[88] Jan Felix Zolitschka. 2020. A novel multi-agent-based chatbot approach to orchestrate conversational assistants. In *Proceedings of Business Information Systems International Conference*. 103–117.