MLDebugging: Towards Benchmarking Code Debugging Across Multi-Library Scenarios

Anonymous ACL submission

Abstract

Code debugging is a crucial task in software engineering, which attracts increasing attention. While remarkable success has been made in the era of large language models (LLMs), current research still focuses on the simple no-library or single-library setting, ignoring the complex multi-library scenario in real-world applications. To address this limitation, we make the first attempt to introduce MLDebugging (Multi-Library Debugging), a comprehensive benchmark designed to assess debugging challenges within multi-library Python code. Specifically, MLDebugging encompasses 126 distinct Python libraries, covering a wide range of multi-library code issues, categorized into seven distinct types. Furthermore, we conduct a thorough evaluation of MLDebugging using both mainstream open-source and closed-source LLMs and highlight that current LLMs still struggle to correctly perform code debugging across multi-library scenarios. We hope this work can uncover the potential of LLMs in multi-library debugging scenario and offer insights for future research.

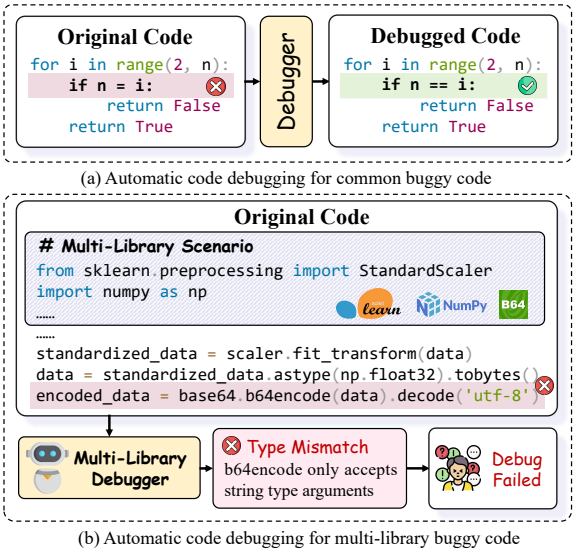


Figure 1: An example of a typical static bug code (a) and our proposed multi-library bug code (b) is presented. In (a), the error arises from using the assignment operator '=' instead of the equality comparison operator '==', while (b) involves an issue of variable adaptation between two library functions.

1 Introduction

Code debugging emerges a significant urge for code review, requiring bug location first and then fix the bug for correct functionality, which has garnered increasing attention for software engineering (Just et al., 2014; Lin et al., 2017). In light of the need to enhance the efficiency of code debugging and repair, a series of work consider adapt Automatic Code Debugging (ACD) techniques to serve as a fast and promising solution to the persistent issue of software defects (Austin et al., 2021; Chen et al., 2021; Li et al., 2024; Shi et al., 2024).

With the advancement of large language models (LLMs), a considerable body of research has been dedicated to effective code debugging. Specifically, Berabi et al. (2021) was the first to reframe the code debugging task as a Text-to-Text problem. Tian et al. (2024) introduced the first debugging dataset specifically designed for LLMs, which leveraged code snippets from the LeetCode (2025) platform. Khan et al. (2024) proposed multiple debugging sub-tasks, thereby expanding the code debugging task to encompass multi-language and multi-task scenarios. Furthermore, Liu et al. (2024b) further extended this benchmark to include multilingual debugging contexts, covering 18 programming languages. This expansion enables a more comprehensive evaluation of LLM performance in debugging across different languages.

Despite its success, as shown in Figure 1 (a), the current researches are still limited to the simple no-library or single-library setting, which which fails to satisfy the requirements of some complex multi-library scenario in a real-world applications. Actually, in real-world software development, the use

Benchmark	MLDebugging	QuickBugs	MdEval	DebugBench	HumanEval	xCodeEval
Multiple Libray	✓	✗	✗	✗	✗	✗
Test Case	✓	✓	✓	✓	✓	✓
Referece Code	✓	✗	✓	✓	✓	✓

Table 1: Compared to the previous Debug Benchmark work, our dataset focuses on Python multi-library.

of multiple libraries is a common practice, as evidenced by research (Feng et al., 2024), which emphasizes the importance of multi-library scenarios for code debugging. Unlike the previous no-library or single-library scenarios, as illustrated in Figure 1 (b), multi-library debugging natrually introduces two distinct challenges: (1) *Understanding multiple libraries for bug location* and (2) *Utilizing multiple libraries for bug fixing*, which cannot be addressed by previous approaches.

Motivated by this, in this work, we introduce MLDebugging (Multi-library Debugging), a benchmark designed to evaluate debugging across 126 libraries, comprising 1,175 samples. As shown in Table 1, the task involves providing a code snippet that integrates multiple libraries, along with descriptions of the required functionality, test cases, and reference code. Specifically, we use GPT-4o (Openai, 2024a) to generate erroneous code based on the multi-library code generation benchmark (Zhuo et al., 2024), which is then debugged by leveraging multiple LLMs. Next, we design a bug category balancing process, enabling the generation of more stable and balanced bugs. Finally, we implement rigorous quality control to measure and validate the quality and authenticity of our dataset by comparing it with the distribution of real-world multi-library bugs.

To assess the limitations of current LLMs, we conduct a thorough evaluation of both open- and closed-source LLMs using MLDebugging. Our experiments reveal the following insights: *(1) Current LLMs excel at debugging method class errors but struggle with conceptual mistakes and missing imports. (2) The structured nature of the MLDebugging, widespread use of libraries, and access to complete runtime information, such as test cases and feedback, enhance LLM performance. (3) In MLDebugging, reasoning models like DeepSeek-r1(Guo et al., 2025), which rely solely on distillation, fail to improve task performance.*

Our contributions are summarized as follows:

(1) We introduce a complex scenario of multi-library code debugging, addressing challenges encountered in real-world development tasks.

(2) We construct a multi-library code debugging benchmark with 1,175 samples, covering 126 commonly used libraries and categorized into 7 distinct bug types relevant to multi-library environments.

(3) We conduct a comprehensive analysis of the dataset's performance across multiple models and provide detailed insights, with further exploration following the experiment.

2 Task Formulation

Consider a complete library set $\mathcal{L}$, an error code $C_{\mathcal{R},l}$ that implements a particular requirement $\mathcal{R}$, and utilizes the subset of libraries $l \subseteq \mathcal{L}$. Given an ideal test case set $\mathcal{T}$, the error code $C_{\mathcal{R},l}$ is defined to satisfy the following condition:

$$\exists t \in \mathcal{T}, \texttt{exec}(C_{\mathcal{R},l}|t) = \texttt{error}, \quad (1)$$

where $\texttt{exec}(x|y)$ denotes the execution of code x with input-output assertion y, returning `error` if execution encounters a fault, or `pass` if the execution is successful without errors.

In the context of multi-library code debugging, the task involves generating the correct code $\hat{C}_{\mathcal{R},l}$ based on the erroneous code $C_{\mathcal{R},l}$. This process can be formally expressed as:

$$\hat{C}_{\mathcal{R},l} = \mathcal{D}(C|C_{\mathcal{R},l}, \mathcal{R}, \mathcal{L}), \quad (2)$$

where $\mathcal{D}$ represents the debugger, which utilizes the library set $\mathcal{L}$ to correct the original code $C_{\mathcal{R},l}$. The resulting corrected code $\hat{C}_{\mathcal{R},l}$ will pass all test cases, satisfying the following condition:

$$\forall t \in \mathcal{T}, \texttt{exec}(C_{\mathcal{R},l}|t) = \texttt{pass}. \quad (3)$$

3 Data Collection

3.1 Source Code Collection

To obtain realistic erroneous code, as shown in Figure 2, we collect practical source code errors as follows: (1) First, to collect queries that involve multiple libraries, we adapt BigCodeBench (Zhuo

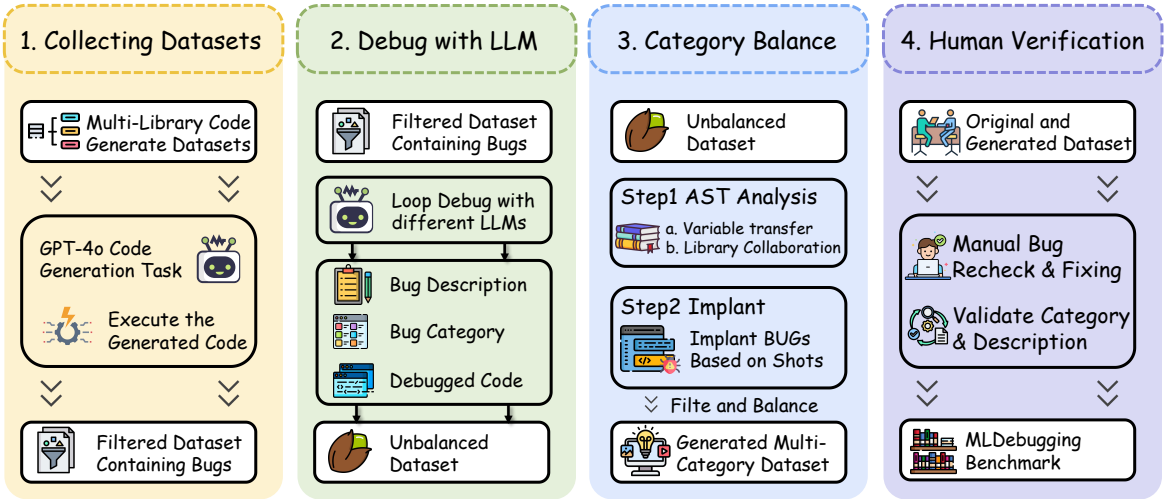


Figure 2: A pipeline diagram illustrating the dataset construction process.

Type	Count
Type Mismatch (TM)	97
Data Transfer Issues (DTI)	127
Function Parameter Errors (FPE)	88
Parameter Configuration Errors (PCE)	60
Function Misuse (FM)	101
Requirement Misunderstanding (RM)	143
Import Errors (IE)	23

Table 2: The number of data instances for each category in the **unbalanced dataset**. In the subsequent sections of the article, we use the first letter shortened forms to replace the full names of the categories (TM, DTI, PFE, PCE, FM, RM, IE).

et al., 2024), a dataset consisting of code snippets that address real-world programming tasks using two or more Python libraries selected from a pool of 179 widely-used libraries. (2) Next, we leverage GPT-4o(Openai, 2024a) following the methodology in (Zhuo et al., 2024) to generate 1,038 code snippets involving multiple libraries, ensuring a broad range of real and diverse code errors. (3) Finally, we test all these generated code snippets using the provided test cases, which successfully identifies 609 buggy code snippets.

To further enable a thorough analysis of the dataset, we examine a set of common multi-library bugs preliminary. As shown in Table 2, we classify these bugs into 7 categories, each based on one of three perspectives: variable transfer between libraries, library function parameters, and function functionality comprehension. Based on this analysis, we introduces a clearer and more analyzable classification framework for evaluating debugging in practice, enabling more precise assessments of debugging performance across different bug types.

3.2 Annotating & Debugging With LLM

As illustrated in Figure 2, based on the previous classification, we manually provide detailed descriptions and examples for each bug category. Using this bug category information, we instruct the LLMs to classify each bug and generate a detailed bug description for each code snippet. This process is designed to assist and accelerate human annotation and next model debugging.

To enhance diversity and improve debugging performance, we employ three LLMs: GPT-4o (Openai, 2024a), DeepSeek-V3 (Liu et al., 2024a) and Claude-3-5-sonnet (Anthropic, 2024), so that systematically collected the corrected code results for subsequent comparative analysis. However, it is worth noting that LLMs cannot always generate and repair code correctly on the first attempt. Inspired by the idea of test-time scaling (Wu et al., 2024; Chen et al., 2025), for any unsuccessful debugging attempts, we conduct up to 5 additional trials to obtain a correct repaired code. After that, we obtain unbalanced dataset, each comprising the bug category, corresponding correct and erroneous code pairs, and relevant test cases.

3.3 Bug Category Balance

As shown in Table 2, an imbalance in bug category distribution leads to evaluation bias, particularly for less frequent errors. To address this, as illustrated in Figure 2, we employ a balancing strategy.

Multi-Library Information Preparation Analyzing source code in isolation often fails to capture

the abstract semantics of code that incorporates multiple libraries, limiting a deeper understanding of its functionality and hindering the generation of debugged code. To address this, we leverage the Abstract Syntax Tree (AST) to represent the program's structure in a hierarchical format. Specifically, we prompt the LLM with the AST, ensuring it reflects variable transfers between libraries, the role of each library at each step, and how they collaborate to accomplish the task.

Category Balance Based on the Information Preparation for Multi-Library code, we select a specific bug type from Table 2 and randomly extract corresponding bug instances from the unbalanced dataset. Specifically, we equally sample each unbalanced category to generated more code with bugs and automatically generate debugged code based on strategies in Sec. 3.1 and Sec. 3.2. Next, we manually filter the generated samples, and finally keep sample size of all categories is left are close. This methodology allowed for the successful injection of 566 bugs, standardizing the number of instances per category to approximately 200.

3.4 Quality Control

To ensure the quality of MLDebugging, we implement comprehensive manual quality control over the dataset.

Manual Bug Recheck & Fixing Due to the model's inability to resolve all bugs, a manual review and correction process is employed for the unresolved code. Specifically, 4 experienced programmers, each with over 4 years of coding experience, are assigned to the task of bug fixing. Prior to beginning their work, these programmers undergo training on 50 sample cases to ensure consistency in labeling and to standardize the review process. To ensure the reliability of the bug-fixing process, overlapping cross-checks are organized, allowing for multiple reviews of the same cases. Any discrepancies identify during these reviews are resolved through collaborative discussions, ensuring accuracy and consistency in the final corrections.

Category & Description Recheck Finally, we manually assess the correctness of bug categories and bug descriptions to ensure that MLDebugging can effectively evaluate the model's ability to debug various types of bugs. Specifically, this process involves annotators comparing the fixed code with the buggy version to validate the correctness of the generated bug category and bug descriptions, as discussed in Sec. 3.2.

Criteria	Correction Count
BUG Descreption	119
BUG Type	340
Manual Debugging	185

Table 3: The number of erroneous samples identified through manual inspection at the quality control threshold.

As a result of these efforts, as detailed in Table 3, we corrected 119 bug descriptions, 340 classifications of multi-library bugs, and manually fixed 185 buggy samples. Additionally, we removed 356 unreasonable samples from the generated dataset.

4 Dataset Analysis

In this study, we design a series of distribution-based experiments to evaluate the alignment between error distributions in our dataset and real-world debugging scenarios. First, we extract question-answer pairs focused on issues related to multiple libraries from Stack Overflow (2025). We then apply textual embeddings to the error descriptions in both the MLDebugging and DebugBench datasets.

To quantify the distributional similarity, we use two key measures: (1) **Centroid-Based Comparison** We calculate the centroids of bug description embedding vectors for MLDebugging, DebugBench, and StackOverflow separately, then evaluate the cosine similarity and Euclidean distance between them. (2) **Real-World Proximity** For each sample, we measure its distance to the nearest real-world sample from StackOverflow. Consider the text embedding vector $\mathbf{a}$ from the benchmark (MLDebugging, DebugBench) and the real bug description embedding vector $\mathbf{b}$ from StackOverflow. We retain the $\mathbf{b_j}$ points closest to $\mathbf{a_i}$ and compute the sum of their Euclidean distances to obtain the Distribution Distance (DD) as:

$$\mathrm{DD} = \sum_{i=1}^{m} \min_{j} \|\mathbf{a}_i - \mathbf{b}_j\|_2. \tag{4}$$

The results in Table 4 show that the cosine similarity between our dataset and StackOverflow exceeds that between DebugBench and StackOverflow. Additionally, the overall distances for our dataset are smaller than those for DebugBench,

Comparison	C Sim(↑)	E Dist(↓)	DD(↓)
ML and ST	0.731	0.376	46.68
DB and ST	0.660	0.432	56.46

Table 4: Comparison of the distances among MLDebugging (ML), DebugBench (DB), and StackOverFlow (ST), where **C Sim** represents the cosine similarity, and **E Dist** represents the Euclidean distance.

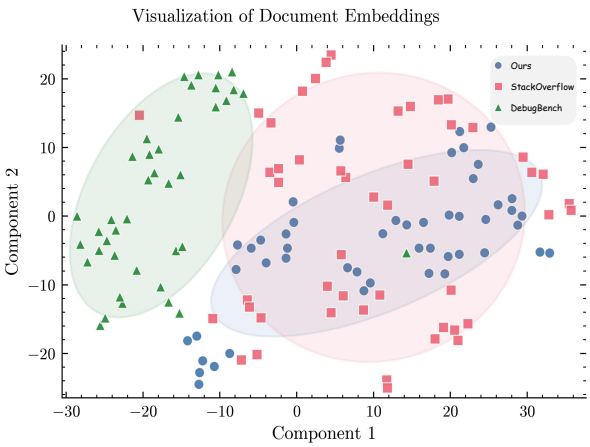


Figure 3: The t-SNE visualization of document embeddings for three datasets in a two-dimensional space.

indicating that, at the script level, MLDebugging more accurately reflects the real-world bug distribution on StackOverflow. Moreover, as shown in Figure 3, the dimensionality reduction visualizations further reveal that our dataset forms more compact clusters, more closely aligning with real-world samples and emphasizing its practical relevance for multi-library code debugging.

5 Experiment

5.1 Experimental Settings

Model Settings We conducted experiments on extensive models covering both open-source and closed-source ones, aiming to provide a comprehensive understanding in multi-libtrary code debugging. Specifically, we choose open-source models including Qwen2.5 (Yang et al., 2024a), Qwen2.5-Coder (Hui et al., 2024), LLama3.1 (AI, 2024), Mistral (Jiang et al., 2024), Deepseek (DeepSeek, 2024) and closed-source models including the GPT series (Openai, 2024b), Claude (Anthropic, 2024). Additionally, we evaluate emerging reasoning models to explore how reasoning capabilities enhance debugging performance, Deepseek-R1-Distill series (Guo et al., 2025) and QwQ (Team, 2024).

Metric Settings The pass rate represents the proportion of code that pass all test cases. Given test cases $\mathcal{T}_i$ and the code $\hat{C}^i_{\mathcal{R},l}$ for i-th sample, the pass rate can be calculated as:

$$\mathcal{P} = \frac{1}{n}\sum_{i=1}^{n}\Big(\bigwedge_{t\in\mathcal{T}_i}\Big[\texttt{exec}(\hat{C}^i_{\mathcal{R},l}|t) = \texttt{pass}\Big]\Big), \quad (5)$$

where n denotes the number of codes in benchmark. Here, $\bigwedge_{t\in\mathcal{T}}[*]$ denotes the logical "and" operation across all $t \in \mathcal{T}$. This expression returns 1 if all test cases pass, and 0 otherwise.

5.2 Main Results

We evaluate MLDebugging with different LLMs varying in size, including closed- and open-source LLMs ranging from 7B to 72B.

All LLMs face challenges in MLDebugging. To assess the debugging capabilities of LLM in multi-library scenarios, we conduct an in-depth evaluation of various models' performances on MLDebugging. As shown in Table 5, none of the LLMs demonstrate a significantly high pass rate in ML debugging. Specifically, the highest performance observed on the DeepSeek-V3 dataset is only 58.7%, which indicates that all LLMs face substantial challenges in MLDebugging. This result also highlights significant room for improvement in the models' multi-library debugging capabilities.

LLMs with increasing parameter sizes exhibit diminishing returns in MLDebugging. To understand the impact of varying model sizes on our benchmark, we evaluate a series of LLMs with different parameter counts. As shown in Table 5, model performance improves significantly from 7B to 32B parameters. However, from 32B to 72B, the performance gain levels off and may even decline. This suggests that our benchmark cannot be fully addressed by scaling alone and presents unique challenges inherent to multi-library tasks, which require further investigation.

LLMs exhibit varying capabilities across different multi-library debugging categories in MLDebugging. To evaluate the debugging abilities of LLMs on different types of bugs, we conduct a detailed experiment based on the classification in Table 2. Table 5 presents the performance of various models on MLDebugging. While LLMs are somewhat effective for addressing certain bug types, multi-library debugging continues to pose significant challenges. Specifically, the categories TM

Category	7B+						14B+			
	Qwen2.5	*Qwen2.5-coder*	*Llama3.1*	*Mistral*	*DS Qwen*	*DS Llama*	*Qwen2.5*	*Qwen2.5-coder*	*DS Qwen*	*DS-coder-Lite*
TM	47.6	40.0	39.7	28.8	18.8	33.5	50.0	54.1	42.4	30.0
DTI	36.1	33.8	30.5	22.7	15.7	20.5	40.7	43.1	30.6	25.5
PFE	48.4	48.8	43.2	29.1	23.5	26.2	56.8	62.0	49.8	34.7
PCE	57.6	58.0	49.8	42.4	33.1	43.0	66.1	63.8	57.6	40.1
FM	38.2	40.4	38.8	26.8	20.2	23.0	44.1	53.6	35.5	31.7
RM	12.6	7.0	5.6	8.4	2.8	7.5	15.4	16.1	11.2	4.9
IE	26.1	8.7	13.0	4.3	3.3	19.0	30.4	30.0	17.4	17.4
AVG.	**42.7**	40.6	36.7	28.0	20.6	27.6	48.6	**51.4**	40.2	29.9
Category	**32B+**				**72B**		**MOE**			
	Qwen2.5	*Qwen2.5-Coder*	*DS Qwen*	*QwQ*	*Qwen2.5*	*Llama3.1*	*DS-V3*	*Claude*	*GPT3.5*	*GPT-4*
TM	58.8	56.5	56.5	46.5	52.9	53.5	60.0	45.9	50.0	55.3
DTI	50.0	50.5	45.8	42.6	47.2	54.1	52.8	39.8	37.5	49.1
PFE	62.0	59.2	59.6	54.5	62.9	45.4	67.0	43.7	54.9	67.1
PCE	70.4	71.5	67.3	58.8	70.4	62.4	76.3	52.1	60.3	70.4
FM	55.9	54.6	50.5	42.6	53.8	68.9	56.2	41.5	44.8	53.0
RM	20.3	18.2	20.3	19.6	16.1	21.7	23.8	25.2	9.1	21.0
IE	26.1	30.4	21.7	30.4	26.1	30.4	34.8	21.7	21.7	30.4
AVG.	**55.7**	54.8	52.6	46.5	**53.7**	53.5	**58.7**	43.0	45.7	55.6

Table 5: The table presents the results of various models in the MLDebugging benchmark, including Qwen2.5, Qwen2.5-Coder, Llama3.1, Mistral, closed-source models, and the Deepseek(DS) R1 Distill series(The models we use are all based on the Instruct version). **Bolded numbers** indicate the highest pass rate achieved within models of the same parameter size.The Category column represents the initials of the category names listed in Table2

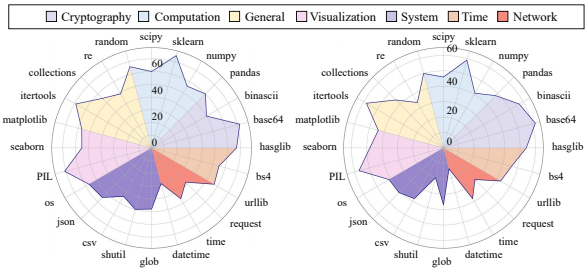
& DTI, which involve *parameter-level* debugging with variable types and specific forms, and the following categories (PFE, PCE, & FM), which focus on *function-level* debugging, demonstrate relatively better performance. In contrast, the last two categories (RM & IE), which require reasoning and debugging at the *library-level*, show notably lower performance, with a gap of nearly 20% in pass rates. This disparity underscores the varying capabilities of LLMs across different multi-library debugging tasks and highlights the need for targeted improvements, particularly in handling debugging challenges of different complexity.

5.3 Analysis

In this section, we analyze the detailed behaviors of LLMs, particularly in investigating the impact from library perspective.

5.3.1 Impact of Library Usage Scenario

To gain a comprehensive understanding of the LLMs' debugging performance across various libraries, we adopt the scenario classification from BigCodeBench (Zhuo et al., 2024). Figure 4 illustrates the debugging pass rates for representative libraries within each scenario. The experimental results indicate that the LLMs' debugging performance varies across different types of libraries, revealing the following insights:



(a) The pass rate distribution of various libraries on GPT-4

(b) The pass rate distribution of various libraries on GPT-3.5-turbo

Figure 4: The usage scenarios of Python libraries are categorized into eight distinct domains, several of the most representative libraries are selected.

LLMs perform well in commonly well-regularized and structured libraries. As illustrated in Figure 4, models perform well in structured and well-regularized libraries covering general algorithms (General), data processing (Computation), and tasks related to encryption and visualization (Visualization). Specifically, they achieve high pass rates in libraries like itertools (0.641), collections (0.570), sklearn (0.654), base64 (0.724), and PIL (0.714) on GPT-4. This strong performance is likely due to the well-structure and clear definition of tasks in large-scale corpora, which allow LLMs to effectively learn and apply general debugging.

LLMs struggle with dynamic and unstructured multi-library debugging. As shown in Figure 4, LLMs underperform in tasks involving network

Variable	Correlation	P-Value
lines of code	-0.0071	0.9654
library count	-0.2113	0.1906
prevalence	0.4094	0.0087

Table 6: The table presents the point-biserial correlation coefficients and p-values between these three factors (lines of code, library count, and library prevalence) and the pass/fail outcomes.

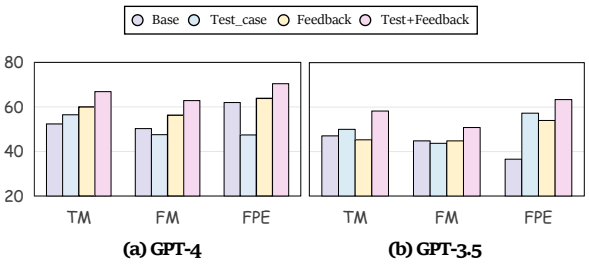


Figure 5: The debugging effect achieved by incorporating runtime information for different bug categories in Table 2.

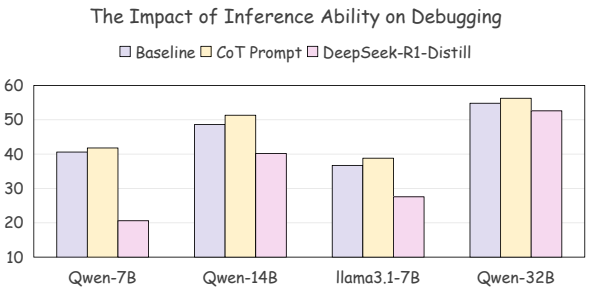


Figure 6: The results of various models under baseline conditions, with CoT prompts, and following R1 Distill training.

communication (NetWork) and time processing (Time), particularly when using libraries like bs4 (0.286), urllib (0.375), and nltk (0.167) on GPT-4. These deficiencies in time-related libraries highlight the model's limitations in dynamic debugging, specifically in its understanding of time logic and cross-timezone processing. This results in challenges when trying to accurately detect anomalous behaviors in dynamic environments.

5.3.2 Impact of Library Prevalence

High prevalence of libraries elicits the models' capacity on MLDebugging. In the experiments presented in Table 5, LLMs demonstrate poor performance in MLDebugging. To explore the underlying causes of these difficulties, we hypothesized that the challenges in debugging are associated with factors such as the number of lines of code, the quantity of libraries used, and the prevalence of LLMs encountered on the internet corpus. To test this hypothesis, we computed the correlation between each of these factors and the pass rate. As shown in Table 6, we find that the prevalence of libraries exhibits the strongest correlation with the debugging difficulty, suggesting that models tend to perform better when handling libraries that are more commonly encountered.

5.3.3 Exploration

Both test cases and runtime error messages are essential for MLDebugging. Execution feedback always serves as a crucial source of information and plays a pivotal role in the debugging process(Zhong et al., 2024). To explore its impact, we introduce detailed test case and runtime error message information separately into prompts to validate the effectiveness for multi-library debugging. As illustrated in Figure 5, the results illustrate the debugging scenarios based on test cases, runtime feedback, and their combined use. The results demonstrate that providing either test cases or runtime feedback individually offers additional information, thereby improving the model's performance in most cases. However, in certain experiments involving only test case feedback or only runtime feedback, we incorporate both information to the LLMs' inputs, leading to better and more robust performance. This suggests that supplying both comprehensive test cases and runtime error messages is essential for ensuring stable performance improvements in debugging tasks.

CoT achieves superior performance in MLDebugging. The Chain-of-Thought (CoT) prompt-based approach has been shown to exhibit enhanced reasoning capabilities in various tasks (Wei et al., 2022; Chen et al., 2024). To investigate this effectiveness for MLDebugging, we adopt the approach introduced by Kojima et al. (2022), wherein the prompt "Let's think step by step" is used to trigger LLMs to generate a CoT reasoning process. Specifically, we assess the performance of contemporary LLMs with parameter sizes ranging from 7B to 32B, including Qwen2.5 and LLaMA 3.1. As shown in Figure 6, the CoT prompt-based approach consistently demonstrates a significant improvement in reasoning capabilities, particularly in debugging tasks. These enhanced logical reasoning skills prove to be crucial for improving performance in multi-library debugging scenarios.

Reasoning models based solely on distillation fail to improve task performance. Recent advancements suggest that reasoning models with test-time scaling significantly enhance the performance of LLMs (Snell et al., 2024; Jaech et al., 2024; Guo et al., 2025; Chen et al., 2025). Motivated by this insight, we evaluated the Deepseek-R1-Distill models. However, as shown in Figure 6, although CoT demonstrates improved performance, reasoning models trained with distilled long CoT data exhibit worse performance. We attribute it to the fact that the use of supervised fine-tuning (SFT) with distilled data alone may not sufficiently enhance the model's capabilities. To achieve a more substantial improvement, further exploration of alternative strategies, such as reinforcement learning (RL), is warranted.

5.4 Error Analysis

To understand the key challenges in MLDebugging task, we conduct a comprehensive review of the debugged code output by the LLMs. We attribute the primary causes to the following two factors:

Absence of library knowledge A key challenge of large language models (LLMs) is their limited knowledge (Huang et al., 2024). To examine this issue in the context of multi-library debugging, we conduct a detailed manual analysis of LLM-generated outputs, with particular emphasis on their use of specialized knowledge. As demonstrated in Case 1 of Error Analysis, LLM fails to understand the "virtual memory" method in the psutil library, leading to a misidentification that hindered its ability to extract the relevant attributes. This highlights a major limitation of LLMs: a superficial or incomplete understanding of specialized libraries. As a result, effectively localizing errors and reconstructing faulty code with a sufficient grasp of the relevant libraries present significant challenges in the field of MLDebugging.

Requirements for efficient cross-library debugging When dealing with interactions between multiple libraries, the inherent complexities introduce significant challenges. As illustrated in Case 2 in Error Analysis, the model's inability to comprehend cross-library variables impedes its capacity to detect redundant operations on the dataframe. The variation in variable types across libraries, such as differing classes and structures, prevents the model from performing fine-grained debugging, resulting in inaccuracies when handling task-specific details. As a result, effectively correct code with cross-library interaction also show great challenges in MLDebugging.

6 Related Work

The emergence of large language models (LLMs) has also exerted a significant impact on Automated Code Debugging (ACD) tasks, as previous datasets have been vulnerable to data leakage risks (Just et al., 2014; Lin et al., 2017). To facilitate a smoother transition from traditional datasets to those suited for LLM evaluations, numerous remarkable contributions have surfaced. DebugBench Tian et al. (2024) stands as the first dataset designed specifically to assess the debugging capabilities of large models. This work collects data from LeetCode, subsequently introducing bugs via model injection. In the realm of APR, xCodeEval Khan et al. (2024) has proposed three distinct subtasks, spanning multiple programming languages, to comprehensively evaluate a mopdel's ability to repair code. Meanwhile, MdEval (Liu et al., 2024b) adopts a multi-language approach, encompassing a benchmark across 18 programming languages. In addition, the challenges presented by specific real-world scenarios have prompted research into niche areas, with several efforts concentrating on executable code, data processing, and other specialized contexts (Yang et al., 2024b; Prenner and Robbes, 2023; Galimzyanov et al., 2024).

These advancements have undeniably propelled the field forward. However, the majority of these datasets are sourced from algorithmic competition platforms, often overlooking the need for Python multi-library scenarios. Therefore, we have constructed a benchmark specifically aimed at evaluating the debugging abilities of models in the context of multiple libraries, providing an in-depth assessment of their proficiency in both static knowledge comprehension and multi-library code interaction.

7 Conclusion

This work introduces MLDebugging, a benchmark designed to assess debugging challenges in multi-library code. We conduct a comprehensive analysis on MLDebugging and the experimental results reveal that the current LLMs still struggle in in multi-library scenario. This work emphasizes the need for further research to improve LLM performance in multi-library settings and provides insights to guide future developments in this field.

Limitations

We propose MLDebugging, the first multi-library code debugging benchmark, yet there are still two main limitations: (1) Although the analysis results indicate that the data distribution of MLDebugging closely resembles real-world data, most of the data in MLDebugging are automatically generated by models, which means there will still be some differences compared to real data. In the future, we consider incorporating more real-world data to further enhance the realism and usability of MLDebugging. (2) Despite that our experiments comprehensively evaluate various models and error categories, the entire evaluation process can be cumbersome due to the need to configure several external dependencies and complex environments, which consumes a significant amount of time.

References

Meta AI. 2024. Llama3.1.

Anthropic. 2024. Claude 3.5 sonnet. Available at: https://www.anthropic.com/news/claude-3-5-sonnet.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR.

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

Qiguang Chen, Libo Qin, Jinhao Liu, Dengyun Peng, Jiaqi Wang, Mengkang Hu, Zhi Chen, Wanxiang Che, and Ting Liu. 2025. Ecm: A unified electronic circuit model for explaining the emergence of in-context learning and chain-of-thought in large language model. *arXiv preprint arXiv:2502.03325*.

Qiguang Chen, Libo Qin, Jiaqi WANG, Jingxuan Zhou, and Wanxiang Che. 2024. Unlocking the capabilities of thought: A reasoning boundary framework to quantify and optimize chain-of-thought. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

DeepSeek. 2024. deepseek.

Jia Feng, Jiachen Liu, Cuiyun Gao, Chun Yong Chong, Chaozheng Wang, Shan Gao, and Xin Xia. 2024. Complexcodeeval: A benchmark for evaluating large code models on more complex code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1895–1906.

Timur Galimzyanov, Sergey Titov, Yaroslav Golubev, and Egor Bogomolov. 2024. Drawing pandas: A benchmark for llms in generating plotting code. *arXiv preprint arXiv:2412.02764*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2024. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of experts. *Preprint*, arXiv:2401.04088.

René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440.

Mohammad Abdullah Matin Khan, M Saiful Bari, Do Long, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in*

neural information processing systems, 35:22199–22213.

LeetCode. 2025. Leetcode: The world's leading online programming learning platform. https://leetcode.com.

Jierui Li, Hung Le, Yinbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. Codetree: Agent-guided tree search for code generation with large language models. *arXiv preprint arXiv:2411.04329*.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Shukai Liu, Linzheng Chai, Jian Yang, Jiajun Shi, He Zhu, Liran Wang, Ke Jin, Wei Zhang, Hualei Zhu, Shuyue Guo, et al. 2024b. Mdeval: Massively multilingual code debugging. *arXiv preprint arXiv:2411.02310*.

Openai. 2024a. Hello gpt-4o.

Openai. 2024b. Introducing chatgpt.

Julian Aron Prenner and Romain Robbes. 2023. Runbugrun–an executable dataset for automated program repair. *arXiv preprint arXiv:2304.01102*.

Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. 2024. From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215*.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Stack Overflow. 2025. Stack overflow: Newest questions. https://stackoverflow.com/questions.

Qwen Team. 2024. Qwq.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, et al. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024a. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Jian Yang, Jiajun Zhang, Jiaxi Yang, Ke Jin, Lei Zhang, Qiyao Peng, Ken Deng, Yibo Miao, Tianyu Liu, Zeyu Cui, et al. 2024b. Execrepobench: Multi-level executable code completion evaluation. *arXiv preprint arXiv:2412.11990*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand. Association for Computational Linguistics.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

Appendix

A Dataset Construction Details

A.1 Prompts for Dataset Construction

This appendix includes the prompts used during both the dataset construction phase and the evaluation phase.

In the second stage of dataset construction, the prompts provided to the model for debugging tasks include several key components: the instruction, the buggy code, the test case, the correct code from the original dataset, and the error message. The model is then required to format and output the solution accordingly.we present the prompts used during the second stage of dataset construction, which are provided to the model for debugging tasks.

> We will provide a Python code snippet, task_func, which involves multiple libraries and contains bugs. Your task is to debug the code, identify the issues across the libraries, and categorize the bugs. Please follow these steps: Review the requirements of the code and thoroughly analyze the task_func.
> Using the provided test cases and error messages, identify and list the bugs in the code, along with a description of each issue. Correct each identified bug, provide the updated code, and categorize the bugs according to the relevant multi-library issue types. The following information is provided for debugging:
> <instruct>: The task of this code segment.
> <bug_code>: The code that needs debugging.
> <canonical_solution>: The corrected version of the code.
> <test_case>: Sample test cases for validation.
> <error>: Relevant error messages.
>
> The identified bugs are categorized as follows:
> Please output each identified bug in the following format, Write each part's content with only one tag, Each part must be included:
> <bug_des>
> Detailed description of the bug </bug_des>
> <code> # Import necessary package(s) and provide the refined function
> code without comments
> import . . . . . .
> def task_func(
> </code>
> <category> Only output the category names from the seven categories mentioned above </category>
> Few-Shot:

we include the prompts employed during the data generation process, covering the analysis and bug injection phases.

> You will receive a piece of code that is a function designed with multiple packages, and its corresponding multi-library AST structure is also provided. Your tasks are as follows:
> (1) Analyze the relationships between multiple libraries from the perspective of variable passing, based on the provided code and its corresponding multi-library AST (Abstract Syntax Tree) structure.
> (2) Inject a specific type of bug, provide a description of the bug, and the code where the bug is injected (do not include any comments). Type Description:
> The input format is as follows:
> <Instruction> Code Requirements </Instruction>
> <Original_Code> Correct Implementation </Original_Code>
> <AST> Abstract Syntax Tree </<AST>
> The output format is as follows:
> <AST_analysis> Analyze the relationships between multiple libraries, focusing on variable passing, Analyze the relationships between multiple libraries from the perspective of variable passing, based on the provided code and its corresponding multi-library AST (Abstract Syntax Tree) structure.based on the provided code and its corresponding multi-library AST (Abstract Syntax Tree) structure.
> </AST_analysis>
> <bug_des> This is a description of multi-database bugs for the specific category required.
> </bug_des>
> <bug_code> # Import the necessary packages and provide the bug-implanted
> code without comments
> import . . . . . .
> def task_func(
> </bug_code>

Finally, we outline the various prompts utilized during the testing phase. Standard evaluation prompt:

> There is an important info hidden inside a lot of irrelevant text. Find it and memorize them. I will quiz you about the important information there.
> Please review the task_func function for errors. Begin by reading the provided instructions to understand the intended behavior of the function. Ensure the code aligns with the requirements and identify any issues. Correct any errors found and provide the revised code.
> Input format:
> <instruct>: Code requirements and expected functionality
> <bug _ code>: The original (bugged) version of the code.
>
> Please output only the debugged code under the label <corrected_code>, without any additional text or comments:
> Output Format Example
> <corrected_code>
> import . . . . . .
> def task_func(. . . . . .
> </corrected_code>

Reasoning model prompt

Please review the task_func function for errors. Begin by reading the provided instructions to understand the intended behavior of the function. Ensure the code aligns with the requirements and identify any issues. Correct any errors found and provide the revised code.
Input format <instruct>: Code requirements and expected functionality <bug_code>: The original (bugged) version of the code.
Please output the final answer at the end, enclosed in markdown format, without any additional text or comments. Final answer output Format Example: “‘python import ..... def task_func(..... “‘

CoT prompt

Please review the task_func function for errors. Begin by reading the provided instructions to understand the intended behavior of the function. Ensure the code aligns with the requirements and identify any issues. Correct any errors found and provide the revised code.

Input format
<instruct>: Code requirements and expected functionality
<bug_code>: The original (bugged) version of the code.

Let's think Step by Step to Solve this problem. Please output the final answer at the end, enclosed in markdown format, without any additional text or comments.
Final answer Output Format Example
“‘python
import .....
def task_func(.....
“‘

A.2 The Details of Manual Annotation

To ensure the quality of the dataset, we provided training for the data annotate team using a sample of 50 entries and established a structured workflow for dataset annotation:

(1) Each annotator is assigned an equal portion of the dataset for labeling, with overlapping data specifically included for cross-checking purposes.

(2) Each annotator first reviews 50 labeled samples, subsequently following the established guidelines for further annotation.

(3) It is imperative that the data reviewed each day meets the criteria, with the bug code matching both its category and description, and the "Golden" code passing corresponding test cases.

(4) Any unresolved issues are addressed through consensus within the annotator group.

The guidelines for the annotate process are as follows:

Instruction
To construct a Code Debug dataset, we first need to debug the code containing bugs. The debugging process involves a thorough examination of both the model and the corrected code. The dataset includes the following information: the code's instructions, the bug categories, the buggy code, the golden code, and testcase. These elements provide essential support for subsequent debugging and model evaluation.
Workflow
The provided files contain a jsonl file, which can be processed using the check.py script to generate a consolidated output file for easier inspection. The Check folder holds the necessary information for performing code debugging, while the RUN folder offers convenient debugging support for code testing. During the debugging process:

(1) It is essential to verify whether the buggy code aligns with the bug description and to assess it according to the classification standards provided in the accompanying materials. If discrepancies are found, corrections must be made accordingly.

(2) The golden code must be executed, and its correctness verified through the corresponding test cases. If it fails to pass the test cases, manual corrections are required.

(3) The correction process for each data entry should be thoroughly documented to ensure traceability of the modifications.

B Error Analysis

Case 1
In the first example, the model fails to locate and refactor the code due to a lack of knowledge regarding the virtual_memory method in the psutil library. As a result, the outputted repair version remains unchanged.

Buggy Code:

```
import psutil
import platform
def task_func():
  system_info = {}
  system_info['OS'] = platform.
      ↪ system()
  system_info['Architecture'] =
      ↪ platform.architecture()[0]
  memory = psutil.virtual_memory()
  system_info['Memory Usage'] = f"{
      ↪ memory.percent}%"
  return system_info
```

Repaired Code:

```
import psutil
import platform
def task_func():
   system_info = {}
  system_info['OS'] = platform.
      ↪ system()
  system_info['Architecture'] =
      ↪ platform.architecture()[0]
  memory = psutil.virtual_memory()
  system_info['Memory Usage'] = f"{
      ↪ memory.percent}%"
  return system_info
```

Golden Code:

```
import psutil
import platform
def task_func():
   system_info = {}

  system_info['OS'] = platform.
      ↪ system()
  system_info['Architecture'] =
      ↪ platform.architecture()[0]

  total_memory = psutil.
      ↪ virtual_memory().total
  used_memory = psutil.
      ↪ virtual_memory().used
  system_info['Memory Usage'] = f'{
      ↪ used_memory/total_memory
      ↪ *100:.2f}%'

  return system_info
```

Case 2

In the following example, the model fails to recognize the interaction between Pandas and Matplotlib, leading to the omission of redundant operations, such as the removal of missing values and reindexing.

Buggy Code:

```
import pandas as pd
import matplotlib.pyplot as plt
def task_func(data_list):
  df = pd.DataFrame(data_list)
  fig, ax = plt.subplots()
  for column in df:

  ax.plot(df[column].dropna().
      ↪ reset_index(drop=True),
      ↪ label=column)
  ax.set_title("Student Scores over
      ↪  Tests")
  ax.set_xlabel("Test Number")
  ax.set_ylabel("Score")

  return ax
```

Repaired Code:

```
import pandas as pd
import matplotlib.pyplot as plt
def task_func(data_list):
  df = pd.DataFrame(data_list)
  fig, ax = plt.subplots()
  for column in df:
  ax.plot(df[column].dropna().
      ↪ reset_index(drop=True),
      ↪ label=column)
  ax.set_title("Student Scores over
      ↪  Tests")
  ax.set_xlabel("Test Number")
  ax.set_ylabel("Score")
  ax.legend()
  return ax
```

Golden Code:

```
import pandas as pd
import matplotlib.pyplot as plt
def task_func(data_list):
  df = pd.DataFrame(data_list)
  fig, ax = plt.subplots()
  for column in df:
  ax.plot(df[column], label=column)
  ax.set_title("Student Scores over
      ↪  Tests")
  ax.set_xlabel("Test Number")
  ax.set_ylabel("Score")

  return ax
```