Why Stop at One Error? Benchmarking LLMs as Data Science Code Debuggers for Multi-Hop and Multi-Bug Errors

Zhiyu Yang[1] **Shuo Wang**[2] **Yukun Yan**[2] **Yang Deng**[1]
[1]Singapore Management University [2]Tsinghua University
`kelvin.yangzhiyu@outlook.com, ydeng@smu.edu.sg`

Abstract

LLMs are transforming software development, yet most code benchmarks still emphasize syntactic or functional correctness in simple, single-error cases. These settings miss the core difficulty of real-world data science debugging, where runtime errors propagate across multiple lines (multi-hop) and often appear in sets (multi-bug). We introduce **DSDBench**: **D**ata **S**cience **D**ebugging **Bench**mark, the first benchmark to systematically evaluate LLMs on this challenge. Unlike general debugging benchmark suites such as SWE-bench, DSDBench targets non-expert, data-centric scripting, where practitioners rely heavily on black-box libraries and write exploratory code that is error-prone and difficult to debug. Evaluations of state-of-the-art LLMs reveal large performance gaps: even frontier models that excel at code generation fail to reliably trace and resolve these errors, exposing a critical "generation versus understanding" gap. DSDBench provides a resource to drive progress toward more robust and trustworthy AI-assisted data science. [1]

1 Introduction

Recent advancements in Large Language Models (LLMs) have significantly reshaped software development practices, particularly in automating code generation and debugging. Benchmarks like DebugBench (Tian et al., 2024), CodeEditorBench (Guo et al., 2024a), and DebugEval (Yang et al., 2025) have played a pivotal role in evaluating LLMs' capabilities in code repair. However, these benchmarks largely rely on simplified programming exercises from platforms like *LeetCode*, which prioritize **syntactic correctness** and **functional accuracy** in **isolated** and **single-error** scenarios, far removed from real-world software complexity.

Meanwhile, growing research efforts are exploring LLMs' potential in data science coding (Yang et al., 2024; Hu et al., 2024; Zhang et al., 2024b; Hong et al., 2024), where practitioners routinely tackle challenges involving black-box library functions, intricate data transformations, and statistical modeling. Yet, a critical gap persists: despite this emerging focus, there remains a striking lack of investigation into LLMs' ability to *debug dynamic logical errors in data science code*. Such errors, manifesting only at runtime, are endemic to this domain due to hidden dependencies in data pipelines, implicit assumptions in mathematical operations, and unpredictable interactions with external resources.

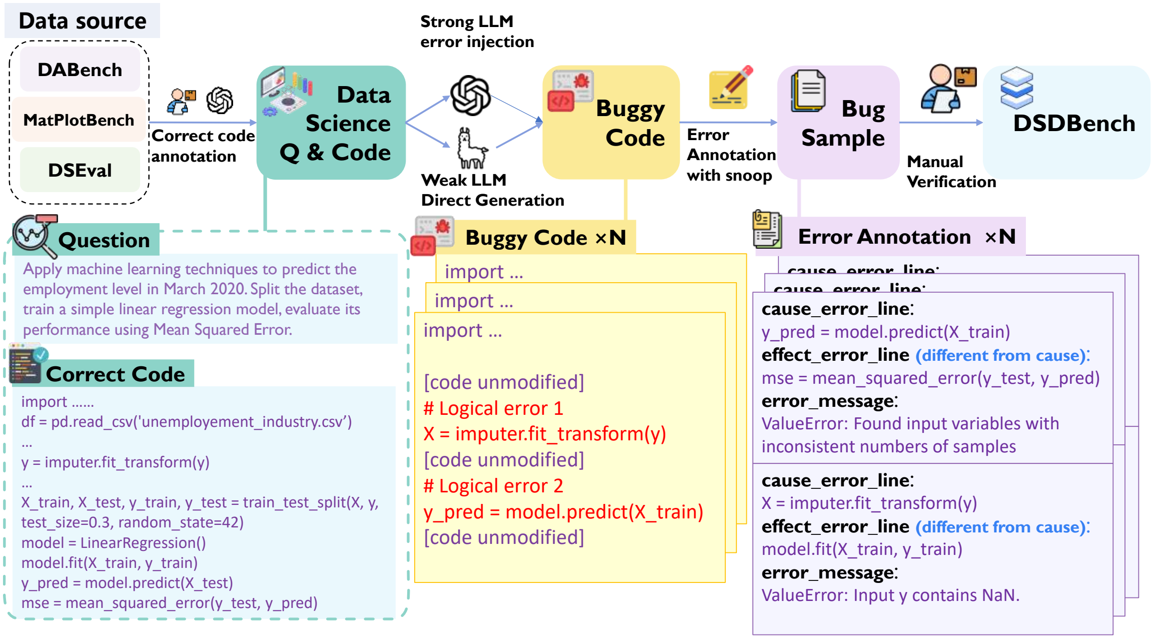
As illustrated in Figure 1, unlike constrained programming exercises, debugging data science codebases presents unique challenges: 1) Its heavy reliance on external libraries (e.g., pandas, NumPy, scikit-learn, matplotlib) means subtle misuses or incorrect data processing steps can easily trigger downstream runtime exceptions. 2) Data scientists often work in interactive environments like Jupyter Notebooks, which lack robust debugging tools. This makes it harder to identify and fix runtime bugs, especially when **multiple subtle errors**, such as incorrect data transformations or misaligned indices, coexist and interact within the code, complicating the debugging process. 3) Standard debugging tools offer limited assistance in diagnosing **multi-hop logical errors** within complex workflows. The root cause of these errors can be distantly located from the point of error manifestation. Standard debuggers typically report the *symptom* (the line of error manifestation in the stack trace) rather than the *root cause* responsible for the program's termination. Linking these two is essential not only for reasoning, but also for *trustworthy* assistance grounded in what users actually see.

Overall, a dedicated benchmark for rigorously

---

[1] DSDBench is publicly available at `https://github.com/KevinCL16/DSDBench`.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 17366–17381
November 4-9, 2025 ©2025 Association for Computational Linguistics



Apply machine learning techniques to predict the employment level in March 2020. Split the dataset, train a simple linear regression model, evaluate its performance using Mean Squared Error.

```
import ......
df = pd.read_csv('unemployement_industry.csv’)
...
y = imputer.fit_transform(y)
...
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
```

```
import ...

[code unmodified]
# Logical error 1
X = imputer.fit_transform(y)
[code unmodified]
# Logical error 2
y_pred = model.predict(X_train)
[code unmodified]
```

```
cause_error_line:
y_pred = model.predict(X_train)
effect_error_line (different from cause):
mse = mean_squared_error(y_test, y_pred)
error_message:
ValueError: Found input variables with
inconsistent numbers of samples
```

```
cause_error_line:
X = imputer.fit_transform(y)
effect_error_line (different from cause):
model.fit(X_train, y_train)
error_message:
ValueError: Input y contains NaN.
```

Figure 1: Dataset construction pipeline of DSDBench.

assessing LLMs’ dynamic debugging of multi-hop logical errors in complex multi-bug data science code is still lacking. Unlike general debugging suites such as SWE-bench that evaluate patch-level fixes in professional OSS repositories via unit tests, our setting targets non-expert, data-centric scripting where failures are often data-dependent and surface as interpreter-visible runtime errors. Crucially, we evaluate whether models can connect *root causes* to these interpreter-visible failure points (*effect lines*), a prerequisite for trustworthy explanations.

Motivated by this evident gap in evaluating LLMs’ dynamic debugging skills for data science, we introduce **DSDBench**: the **D**ata **S**cience **D**ebugging **Bench**mark. Distinct from prior works that primarily focus on repairing single, syntactic and static errors, DSDBench is the first benchmark to systematically evaluate LLMs on: (1) **Multi-Hop Error Tracing**: requiring models to trace runtime errors back through multiple lines of data science code to identify the root cause; and (2) **Multi-Bug Error Detection**: assessing their ability to concurrently detect and reason about multiple logical errors within a single data science code snippet. We *focus on interpreter-visible (crashing) bugs* to enable scalable, deterministic annotation of cause/effect lines and messages. For clarity, we define a “hop” whenever the cause and effect lines differ, and for multi-bug items a prediction is counted correct only if *all* constituent bugs are identified. Table 1 summarizes the comparisons between DSDBench and existing code debugging benchmarks.

Benchmark	Domain	Error Complexity	Multi-Hop Error	Error Type
DebugBench	General	Multi-Bug	✗	Static
DebugEval	General	Multi-Bug	✗	Static
CodeEditorBench	General	Single-Bug	✗	Static
DSDBench	**Data Science**	Multi-Bug	✓	**Runtime**

Table 1: Comparison with existing benchmarks.

DSDBench leverages datasets and tasks from established data science coding benchmarks like DABench (Hu et al., 2024), MatPlotBench (Yang et al., 2024), and DSEval (Zhang et al., 2024b). We systematically inject errors into data science code, synthesizing multi-error scenarios by combining individual bugs. Our dataset comprises 1,117 meticulously annotated samples, complete with ground-truth cause-effect error line pairs and captured runtime error messages.

In summary, our contributions are threefold:

- **DSDBench Benchmark:** We release the first dedicated benchmark and dataset for evaluating LLMs in runtime, multi-bug debugging of data science code. DSDBench features realistic logical errors, multi-hop error scenarios, and detailed annotations, addressing a critical gap in current debugging benchmarks.
- **Automated Error Injection and Annotation**

Framework: We develop a robust pipeline for automated error injection, runtime execution tracing, and alignment of interpreter outputs with error-originating code lines, facilitating scalable benchmark creation and future expansion.

- **Empirical Analysis and Insights:** We present a comprehensive empirical evaluation of state-of-the-art closed-source and open-source LLMs on DSDBench. Our findings reveal significant performance gaps and highlight critical challenges in dynamic debugging for complex, real-world data science code.

2 DSDBench Construction

The creation of a high-quality dataset is paramount for a robust benchmark. As illustrated in Figure 1, DSDBench is meticulously constructed through a multi-stage process encompassing data sourcing, correct code preparation, error injection, error annotation, and quality assurance.

2.1 Data Collection

We build the DSDBench upon three widely-adopted data science coding benchmarks for their realistic data science tasks and diverse scenarios, including DABench (Hu et al., 2024), MatPlotBench (Yang et al., 2024), and DSEval (Zhang et al., 2024b). We focus on the hard subset of DABench because error injection in its easy and medium subsets rarely produces runtime exceptions. MatPlotBench and DSEval supplement DABench, expanding task diversity and library coverage (`pandas`, `sklearn`, `scipy`, `matplotlib`, `numpy`) to represent typical data science workflows. These benchmarks cover data manipulation, statistical analysis, machine learning, and visualization.

However, some of these datasets mainly contain the natural language instructions and the final results after running the data science code, while the ground-truth correct codes are not provided. As the first step, we prepare the correct and error-free codes for each benchmark as follows:

DABench We design an agent-based annotation framework, which includes a self-debugging code agent and an error verifier agent. Annotation begins by feeding benchmark questions and metadata to the self-debugging code agent, which generates initial code and debugs it based on error messages. Subsequently, the error verifier agent analyzes this code to correct logical errors, meanwhile ensures the code produces correct answers according to DABench's ground truths. The details of the agent-based annotation framework are presented in Appendix A.

MatPlotBench Similar agent-based code generation is adopted, but automated verification is challenging due to the visual nature of plot outputs. Therefore, manual expert verification is employed, comparing plots to ground truth images and correcting code for accurate visualizations.

DSEval We extract and concatenate code blocks from ground truth Jupyter notebooks provided by DSEval, using concatenated code as our benchmark's correct code.

2.2 Error Injection

To systematically introduce errors, we employ two complementary methods. This dual design ensures both realism and diversity of bug types. The details of prompts are provided in Appendix B.

Strong LLM-based Error Injection We instruct GPT-4o to inject runtime-interrupting bugs into otherwise correct solutions. Step 1: identify lines invoking core data science libraries (`numpy`, `scipy`, `matplotlib`, `sklearn`, `pandas`). Step 2: modify them to introduce plausible faults such as API misuse, incorrect parameter settings, or data-shape mismatches (e.g., NaNs, inconsistent dimensions). These produce realistic logic/runtime bugs. Importantly, injection often yields **multi-hop errors**: the causal bug occurs earlier in the program, but the crash manifests later (e.g., a faulty imputation call causing a failure during model training).

Weak LLM-based Direct Error Generation We also instruct Llama-3.1-8B to write entire solutions from scratch. Due to its weaker capability, the generated code frequently contains beginner-style mistakes (e.g., missing imports, wrong API signatures, incomplete function definitions). These naturally introduce **multi-hop errors**, for example when a sub-function mistake surfaces as an exception in the `main` function.

Rationale Using both approaches diversifies error characteristics: strong LLM injection yields semantically rich runtime bugs, while weak LLM generation reflects novice-style mistakes. This mitigates bias from relying on a single method.

Dataset Size			Example Type		Multi-Error Examples	Code Complexity	Question Complexity
Total # Examples	# Single-Error	# Multi-Error	# Single-hop	# Multi-hop	Avg Errors/Example	Avg Code Length	Avg Question Length
1,117	741	376	385	356	2.87 ± 1.14	65.31 ± 21.31	92.42 ± 55.86

Table 2: Dataset statistics of DSDBench.

2.3 Error Annotation

For each buggy snippet, we annotate three ground truths: `cause_error_line`, `effect_error_line`, and `runtime error message`. We then construct both single-error and multi-error cases, requiring models to reason across causal and manifested locations.

Single-Error Annotation We commence dynamic error capture with `snoop`[2], a Python debugging library that logs execution details, for single-error ground truth. `snoop` monitors the execution of both injected and direct generated error code. We first filter out successfully executed ones. For error-triggering snippets, we analyze `snoop`'s execution traces to extract: `cause_error_line` (error origin), `effect_error_line` (error manifestation), and `runtime error messages`, providing ground truths for single-error annotation.

Multi-Error Annotation We create multi-error cases by systematically combining validated single errors into candidate pools per question, from which random subsets are sampled. An instance is marked *correct* only if *all* constituent bugs are identified. Although each bug is logically independent, runtime interactions (e.g., early crashes masking later ones) can increase difficulty, which we preserve to reflect realistic debugging.

Multi-Hop Definition We define a "hop" as the spatial separation between the cause line (where the bug originates) and the effect line (where execution fails). The hop distance may be a few lines within the same function (no change in call stack) or span across functions (involving call stack depth). The essential characteristic is that the interpreter reports a crash at a different line than where the bug was introduced, forcing models to trace execution flow.

2.4 Human Verification and Quality Control

To ensure the quality and correctness of the constructed dataset, we perform a two-stage verification process: **code-based checks** and **LLM-assisted verification**. 1) Code checks involve printing and manually inspecting annotated cause and effect lines to correct nonsensical annotations by human annotators. We also print error messages, identifying and resolving a common `plt.show()` backend issue by adding backend settings to the MatPlotBench correct code examples. 2) LLM-assisted verification is used to review all annotations, flagging remaining inconsistencies that require human intervention to correct the annotations. Overall, the pass rates of the human verification for the two stages are 83% and 87%, respectively. The high pass rates also validate the effectiveness of the automated annotation process.

Bug Distribution, Realism, and Scope We approximate non-expert data-science scripting, where code is exploratory and loosely structured. To mitigate LLM-specific biases, our injection is constrained and modular: a strong LLM first identifies API-invoking lines and then performs *atomic, plausible* edits (e.g., argument name/value mistakes, missing `axis`, shape mismatches), limiting unrealistic global rewrites and better mimicking small mistakes that propagate during execution. Weak-LLM generation complements this by introducing novice-style patterns at the program level. We presently *focus on interpreter-visible (crashing) bugs* to support reliable, scalable labeling; functional bugs without hard crashes are deferred to future work. Empirically, models with stronger coding/reasoning (e.g., Claude-3.5-Sonnet, DeepSeek-V3, LRMs) often outperform GPT-4o (used for injection), suggesting DSDBench rewards genuine reasoning rather than model-specific shortcuts.

2.5 Dataset Characteristics

This section presents a statistical overview of the DSDBench dataset, characterizing its composition, diversity, and complexity. Table 2 provides a statistical overview of the DSDBench dataset. The **dataset size and splits** are as follows: the total number of examples is 1,117, of which 741 are single-error examples and 376 are multi-error examples. For single-error examples, the number of examples with multi-hop cause and effect error lines is 356, the rest 385 examples contain identical cause and effect error lines *i.e.*, single-hop errors).

---

[2] https://pypi.org/project/snoop/

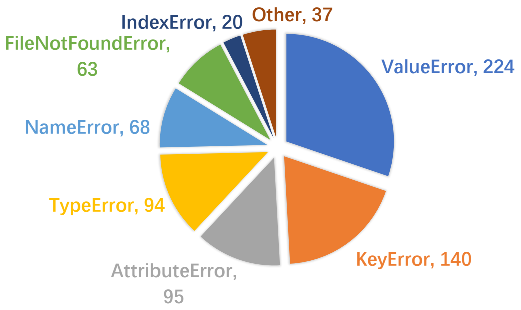


Figure 2: Distribution of different error types. Details of error types are described in Appendix C.

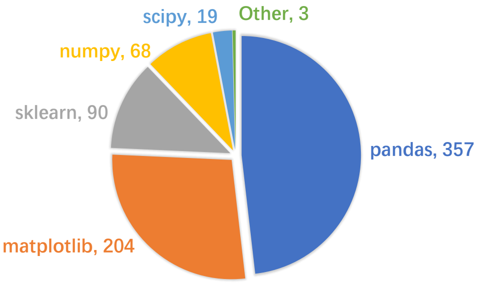


Figure 3: Distribution of different data science libraries.

For multi-error examples, the number of errors per example ranges from 2 to 9, with an average of 2.87 errors per example. Regarding complexity, the average code length is 65.31 lines, and the average question length is 92.42 words. For a more detailed breakdown of these statistics, please refer to Table 2.

Figure 2 summarizes the **error type distribution**, and Figure 3 shows the **library coverage**. Together they indicate broad coverage across common runtime/data-driven failures and core APIs, lending distributional realism despite synthetic injection.

3 Problem Formulation

3.1 Task Definition

This section formally defines the task of **Data Science Code Debugging** for the DSDBench benchmark, outlining the input, desired output, and evaluation settings. The primary objective of DSDBench is to evaluate the capability of LLM-based debuggers to identify and explain **logical errors** in **data science Python code** during simulated **runtime execution**.

The benchmark targets two critical dimensions: **multi-hop error detection** and **multi-bug error detection**. **Multi-hop** evaluates whether models can trace errors to their **root cause** (**`cause_error_line`**) when it differs from the **interpreter's error point** (**`effect_error_line`**). We define a "hop" whenever these two lines are not identical; the distance may be within the same function (no call-stack change) or across function calls (with call-stack depth). This spatial separation forces reasoning over execution flow. **Multi-bug** requires identifying and explaining *all* concurrent logical errors in a snippet, not merely the first encountered, mirroring real debugging where missing one error leaves the program incorrect. We also evaluate **error message reproduction**, i.e., whether models can semantically reproduce the **interpreter-thrown messages** for each identified error.

Formally, for each task instance i, the input is a pair (Q_i, C_i), where Q_i is a natural language question describing a data science task, and C_i is a Python code snippet intended to perform task Q_i, but containing logical errors. The task of the LLM is to predict a structured output $O_i = (L_{cause,i}, L_{effect,i}, M_i)$, where $L_{cause,i}$ is the exact line of code for the cause error, $L_{effect,i}$ is the exact line of code for the effect error, and M_i is the error message that would be produced by a Python interpreter when executing C_i. The DSDBench benchmark dataset can be represented as $D = \{(Q_i, C_i, L^{GT}_{cause,i}, L^{GT}_{effect,i}, M^{GT}_i)\}^N_{i=1}$, where GT denotes the ground truth annotation. The objective is to evaluate LLMs' capabilities to perform the task of $f : (Q_i, C_i) \mapsto O_i$ which localizes and interprets the error.

3.2 Evaluation Metrics

This section details evaluation metrics for LLM debugger performance on DSDBench, focusing on error localization accuracy and description quality. Model performance is evaluated across four dimensions, including **Cause Line Matching**, **Effect Line Matching**, **Error Type Matching**, and **Error Message Matching**.

We compute `cause_line_score`, `effect_line_score`, and `error_type_score` as binary exact matches (1 for exact match with ground truth, 0 otherwise). For error messages, exact string matching is too brittle: LLMs may produce semantically equivalent messages with different surface forms. We therefore adopt a semantic rubric scored by GPT-4o on a five-point Likert scale {0.0, 0.25, 0.5, 0.75, 1.0}. Example: *GT:* "ValueError: Input contains NaN" vs. *Pre-*

diction: "ValueError: The model cannot handle NaN values in the input"—lexically different but semantically equivalent (scored ≥ 0.75). The full rubric and additional examples are provided in Appendix D.

Dimension-Level Definitions: For each evaluated dimension:

- **TP (True Positives):** Number of instances with correct LLM predictions (exact match for lines/types, `error_message_score` ≥ 0.75 for error messages).
- **FP (False Positives):** Number of instances with specific incorrect LLM predictions (commission errors).
- **FN (False Negatives):** Number of instances where LLM failed to provide a relevant prediction, (omission errors) e.g., incorrect output format ; $FN = \mathit{GT\_Instances} - (TP + FP)$.
- ***GT_Instances*:** Total Ground Truth Instances for the dimension.

Evaluation Metrics (per dimension): We employ Precision, Recall, F1-score, and Accuracy to evaluate performance across dimensions. Because DSDBench only contains test cases with errors, meaning there is no True Negatives in model predictions. Therefore, we calculate **Recall** by **(True Positive Rate - TPR)** to measure the completeness of error detection as:

$$\text{Recall (TPR)} = \frac{TP}{\mathit{GT\_Instances}}$$

making Recall (TPR) numerically equivalent to Accuracy. All metrics are calculated dimension-wise to provide a detailed performance profile.

4 Experiments

4.1 Setup

Models We benchmarked a diverse set of state-of-the-art models on the DSDBench dataset, including both closed-source models and open-source models. Specifically, the closed-source models we employed were GPT-4o, GPT-4o-mini, o1-mini (OpenAI, 2024), Gemini 2.0 Flash Thinking (Google, 2024), and Claude 3.5 sonnet-20240620. Open-source model consisted of Llama-3.1-8B-instruct, Llama-3.1-70B-instruct, Llama-3.1-405B-instruct (Meta, 2024), Qwen2.5-7B-Instruct, Qwen2.5-32B-Instruct, Qwen2.5-72B-Instruct (Qwen, 2025), DeepSeek-V3, DeepSeek-R1 (DeepSeek-AI, 2025). Notably, we categorize Gemini 2.0 Flash Thinking, DeepSeek-R1 and o1-mini as Large Reasoning Models (LRMs). All models were used with their default decoding parameters apart from setting temperature to 0. Zero-shot setting were used. We used OpenRouter's API services for all models.

Evaluation Protocol The evaluation prompt is identical across models and includes the task description, the buggy DSDBench Python snippet, and instructions to output a structured JSON diagnosis. We *disallow external tools and execution* to isolate in-context reasoning. For *multi-bug* items, a prediction is counted correct only if *all* constituent bugs are identified (no partial credit), consistent with Section 2.3. The template appears in Appendix D. Metrics follow Section 3.2.

A pilot study with agentic systems is reported in Appendix G; here we focus the main results on in-context localization without execution.

4.2 Main Results

Table 3 and Table 4 present the primary results of our experiments, showing the accuracy of various models in detecting single and multi-bug scenarios across the full and subset DSDBench datasets.

Single-Bug Debugging Performance As shown in Tables 3, top-performing LLMs like Deepseek-V3 and Claude 3.5 sonnet achieve reasonable accuracy across all tasks, indicating a degree of error tracing capability. Conversely, smaller models such as Llama-3.1-8B-instruct and Qwen2.5-7B-Instruct exhibit significantly lower accuracy. Notably, Qwen2.5-72B-instruct demonstrated strong performance, on par with state-of-the-art closed-source LLMs such as GPT-4o and Claude 3.5 sonnet. In general, effect line accuracy is consistently lower than cause line accuracy across models, showing LLMs' deficiency to reason about code execution traces and find the exact location where the program would trigger an error. Error type and error message accuracy vary across different models, suggesting varying levels of understanding and interpretation of runtime errors.

Challenges in Multi-Bug Debugging The results reveal a dramatic decrease in accuracy when models are challenged with multi-bug scenarios, models fails to identify an correct set of errors within a code snippet with multiple bugs. Even for the best-performing models, cause line accuracy drops to around 20% on the full dataset and 30%

Model	**Cause Line**		**Effect Line**		**Error Type**		**Error Message**	
	Single-Bug	**Multi-Bug**	**Single-Bug**	**Multi-Bug**	**Single-Bug**	**Multi-Bug**	**Single-Bug**	**Multi-Bug**
GPT-4o	39.0	20.3	34.3	10.4	30.6	**3.6**	31.4	**4.7**
GPT-4o-mini	40.2	11.2	23.9	2.7	21.7	2.2	21.3	0.8
Claude 3.5 sonnet	43.7	12.3	35.2	4.1	**36.3**	1.9	34.0	2.5
Deepseek-V3	**48.3**	15.1	34.5	6.6	35.9	3.3	**34.7**	**4.7**
Llama-3.1-8B-instruct	25.2	3.0	14.2	0.0	7.7	0.0	7.2	0.0
Llama-3.1-70B-instruct	42.5	0.0	29.3	0.0	20.4	0.0	20.9	0.0
Llama-3.1-405B-instruct	41.7	18.6	31.3	8.5	29.3	1.1	29.3	2.5
Qwen2.5-7B-Instruct	29.3	4.7	19.3	1.1	10.7	0.3	10.9	0.0
Qwen2.5-32B-Instruct	40.9	17.5	30.5	6.3	24.7	2.2	24.7	2.2
Qwen2.5-72B-Instruct	41.6	**21.4**	**36.2**	**11.2**	27.5	3.0	27.4	3.6

Table 3: Overall evaluation results of LLMs on DSDBench. The reported score is the Accuracy (%), while full metrics are presented in Appendix E.

	Model	**Cause Line**		**Effect Line**		**Error Type**		**Error Message**	
		Single-Bug	**Multi-Bug**	**Single-Bug**	**Multi-Bug**	**Single-Bug**	**Multi-Bug**	**Single-Bug**	**Multi-Bug**
LLMs	GPT-4o	35.4	12.5	31.2	5.0	33.3	2.5	33.3	2.5
	GPT-4o-mini	39.6	7.5	29.2	5.0	25.0	2.5	22.9	0.0
	Deepseek-V3	**44.8**	12.5	28.1	7.5	**34.4**	5.0	**34.4**	7.5
LRMs	Gemini 2.0 Flash Thinking	42.7	20.0	**32.3**	12.5	33.3	0.0	35.4	2.5
	Deepseek-R1	**49.0**	32.5	**49.0**	**25.0**	**53.1**	**15.0**	**54.2**	**17.5**
	o1-mini	43.8	**35.0**	36.5	22.5	43.8	**17.5**	46.9	**17.5**

Table 4: Comparison with large reasoning models (LRMs). The reported score is the Accuracy (%), while full metrics are presented in Appendix E. Due to the unstableness of certain LRM APIs, we randomly sample a subset of DSDBench for this evaluation, which comprises of 96 Single-Error and 40 Multi-Error instances.

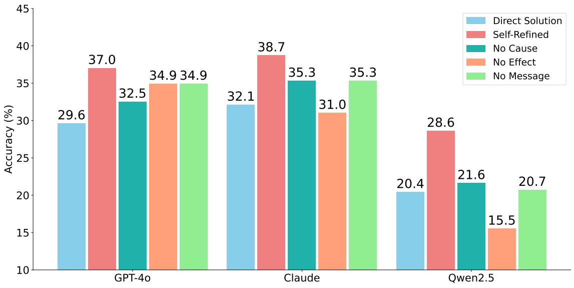


Figure 4: Impact of Self-Refinement.

Error Type	Cause Line			Effect Line		
	GPT-4o	Qwen	DeepSeek	GPT-4o	Qwen	DeepSeek
ValueError	57.9	61.6	**66.1**	50.5	**59.6**	54.0
TypeError	30.8	39.5	**50.0**	31.9	34.6	**37.8**
NameError	68.2	64.0	**85.4**	56.1	**60.0**	52.1
KeyError	22.7	28.4	**37.8**	**22.7**	17.6	27.9
AttributeError	35.1	**40.5**	40.0	**22.3**	14.9	15.0
IndexError	36.8	**41.2**	38.9	36.8	**58.8**	55.6
FileNotFoundError	0.0	9.6	**13.0**	1.6	9.6	**11.1**
Other	38.5	53.3	**66.7**	23.1	**46.7**	33.3

Table 5: Precision w.r.t. different error types. The **bold** scores represent the best model performance across error types and prediction tasks.

on the subset. This substantial performance degradation underscores the increased complexity of debugging multiple bugs concurrently. Furthermore, the low accuracy in error type and error message prediction in multi-bug cases suggests that models struggle to correctly interpret error messages within these more complex contexts.

LRMs Show Promise in Multi-Bug Debugging Comparing LLMs and LRMs on the subset dataset (Table 4) reveals that LRMs generally outperform standard LLMs, particularly in the more demanding multi-bug scenarios, indicating superior reasoning capabilities in LRMs are crucial for tackling complex debugging tasks. A more detailed analysis and case study can be found in Figure 7.

4.3 Impact of Self-Debugging

To study how DSDBench-style diagnosis helps downstream coding, we use models' own debugging outputs as guidance to solve DABench-Hard, comparing direct solutions vs. self-refined solutions that consume *(cause line, effect line, message)*.

Figure 4 shows that **self-refinement** improves accuracy across GPT-4o, Claude 3.5 Sonnet, and Qwen2.5-72B. Ablations confirm that *lines* carry most of the value—removing *Cause*/*Effect* hurts more than removing *Message*.

4.4 Detailed Analysis

This section analyzes model performance across error types, libraries, error counts, and multi-

Library	Cause Line			Effect Line		
	GPT-4o	Qwen	DeepSeek	GPT-4o	Qwen	DeepSeek
matplotlib	46.6	48.4	**55.6**	45.6	52.2	**55.6**
numpy	41.4	40.4	**44.0**	**37.9**	36.8	32.0
pandas	28.1	37.0	**41.0**	21.6	22.0	**24.3**
sklearn	65.1	72.5	**87.7**	58.1	**63.8**	53.8
scipy	36.4	54.5	**72.7**	18.2	36.4	**45.5**

Table 6: Precision w.r.t. different libraries.

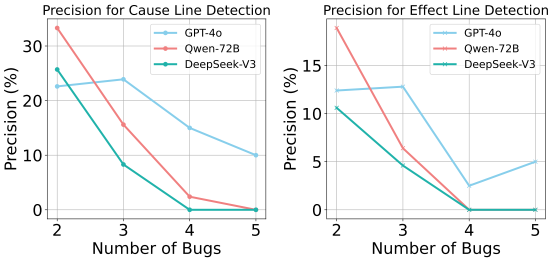


Figure 5: Precision for multi-Bug detection with different number of errors.

hop/single-hop to identify strengths and weaknesses. We adopt GPT-4o, Qwen-72B-Instruct, and DeepSeek-V3 for analysis.

Performance by error types Table 5 shows error type precision. Models exhibit varying performance on different error types. Generally, models perform better on more common error types and less on more obscure error types. Low performance on FileNotFoundError is possibly attributed to models not having access to the coding environment and file system. DeepSeek-V3 performs best on identifying Cause Lines, scoring the highest on every error type except AttributeError and IndexError. Qwen-72B-Instruct performs best on identifying Effect Lines.

Performance by data science libraries Table 6 shows library-specific precision. Pandas is the most difficult library to debug, due to its intricate and black-box data manipulation. Models demonstrated best performance on scikit-learn and reasonable performance on matplotlib, numpy and scipy, with significant room for improvement. DeepSeek-V3 demonstrated consistently better performance than other models across all libraries, showing robustness on all kinds of data science coding tasks.

Performance by number of errors Figure 5 shows precision by error count. Precision decreases significantly with more errors for all models. Precision drops sharply when there exists more than 3 errors, highlighting difficulty in multi-error scenarios. GPT-4o performs best on retaining precision on more than 2 bugs, showing potential in concurrent debugging.

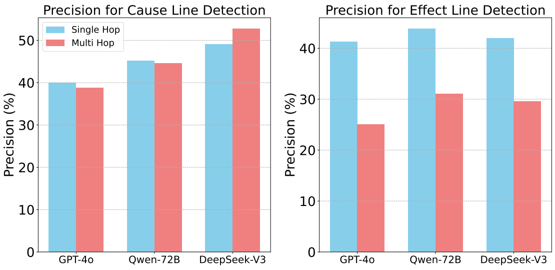


Figure 6: Precision for single-bug detection comparing multi-hop and single-hop errors.

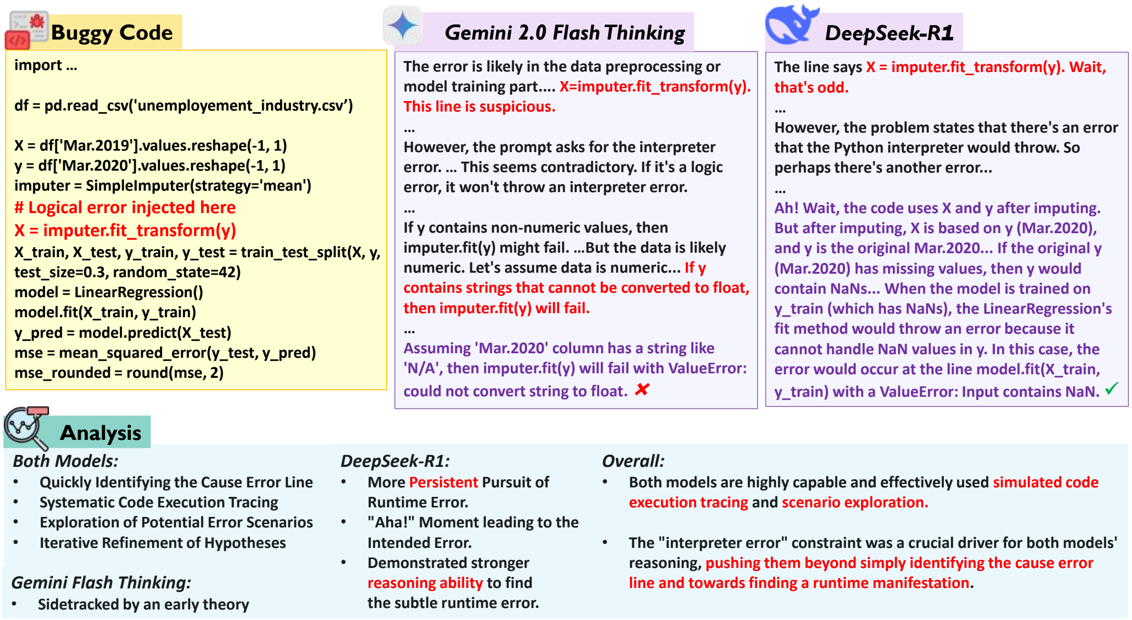
Performance on single-hop vs. multi-hop errors Figure 6 compares single-hop vs. multi-hop error precision. Cause Line precision is similar for both types of errors across models. Effect Line precision is significantly lower for multi-hop errors, indicating increased difficulty when locating the point where the program triggers an exception. DeepSeek-V3 demonstrated best performance on Cause Line detection on both single-hop and multi-hop errors, while Qwen2.5-72B-Instruct performs slightly better on Effect Line detection.

Prompting Strategies We additionally benchmark several test-time prompting strategies (Zero-shot/One-shot CoT, Self-Refine, Self-Consistency) on a single-bug subset. Self-Consistency yields the largest gains for some models, but overall performance remains far from saturation. Full setup and results appear in Appendix F.

Agentic Systems We also ran a pilot with agentic coding systems (Cursor Agent with Claude 3.5 Sonnet, and OpenHands). Single-attempt repair boosts performance over standalone localization, while unconstrained workflows can approach 100% pass rate by broadly rewriting code. Full setup, tables, and analysis are provided in Appendix G.

4.5 Case Study

According to Figure 7, LRMs have demonstrated intriguing capabilities on reasoning multi-hop erroneous code. Both Gemini 2.0 Flash Thinking and DeepSeek-R1 can promptly identify the `cause error line`, then mentally simulate code execution trace and explore multiple possible scenarios that could lead to runtime exception. However, Gemini 2.0 Flash Thinking was distracted by one



Buggy Code

```
import ...

df = pd.read_csv('unemployement_industry.csv')

X = df['Mar.2019'].values.reshape(-1, 1)
y = df['Mar.2020'].values.reshape(-1, 1)
imputer = SimpleImputer(strategy='mean')
# Logical error injected here
X = imputer.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
mse_rounded = round(mse, 2)
```

Gemini 2.0 Flash Thinking

The error is likely in the data preprocessing or model training part.... X=imputer.fit_transform(y). This line is suspicious.

...

However, the prompt asks for the interpreter error. ... This seems contradictory. If it's a logic error, it won't throw an interpreter error.

...

If y contains non-numeric values, then imputer.fit(y) might fail. ...But the data is likely numeric. Let's assume data is numeric... If y contains strings that cannot be converted to float, then imputer.fit(y) will fail.

...

Assuming 'Mar.2020' column has a string like 'N/A', then imputer.fit(y) will fail with ValueError: could not convert string to float. ✗

DeepSeek-R1

The line says X = imputer.fit_transform(y). Wait, that's odd.

...

However, the problem states that there's an error that the Python interpreter would throw. So perhaps there's another error...

...

Ah! Wait, the code uses X and y after imputing. But after imputing, X is based on y (Mar.2020), and y is the original Mar.2020... If the original y (Mar.2020) has missing values, then y would contain NaNs... When the model is trained on y_train (which has NaNs), the LinearRegression's fit method would throw an error because it cannot handle NaN values in y. In this case, the error would occur at the line model.fit(X_train, y_train) with a ValueError: Input contains NaN. ✓

Analysis

Both Models:

- Quickly Identifying the Cause Error Line
- Systematic Code Execution Tracing
- Exploration of Potential Error Scenarios
- Iterative Refinement of Hypotheses

Gemini Flash Thinking:

- Sidetracked by an early theory

DeepSeek-R1:

- More Persistent Pursuit of Runtime Error.
- "Aha!" Moment leading to the Intended Error.
- Demonstrated stronger reasoning ability to find the subtle runtime error.

Overall:

- Both models are highly capable and effectively used simulated code execution tracing and scenario exploration.
- The "interpreter error" constraint was a crucial driver for both models' reasoning, pushing them beyond simply identifying the cause error line and towards finding a runtime manifestation.

Figure 7: Case study of LRMs.

of its early theories and produced an incorrect answer. On the other hand, DeepSeek-R1 ruled out all implausible possibilities after relentlessly pursuing an explanation for triggering a runtime error, eventually came up with the correct answer.

5 Related Work

LLMs for Coding and Debugging LLM progress in code tasks led to benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) for code synthesis, assessing syntactic correctness and functional accuracy. Runtime information is increasingly used in LLM debuggers (Zhong et al., 2024). Multiple benchmarks (Yang et al., 2025; Tian et al., 2024; Jimenez et al., 2024; Ni et al., 2024; Gu et al., 2024; Jain et al., 2024; Liu et al., 2024a) have focused on LLM debugging and code reasoning.

In data science coding, the landscape is evolving with general tools like the Data Interpreter (Hong et al., 2024) and specialized agents such as MatPlotAgent (Yang et al., 2024) and DSAgent (Guo et al., 2024b). Benchmarks such as DSBench (Jing et al., 2024), InfiAgent-DABench (Hu et al., 2024), DSEval (Zhang et al., 2024b), and PyBench (Zhang et al., 2024a) are emerging to evaluate the performance of LLMs in data science coding. However, they do not capture the complexities of real-world data science debugging.

DSDBench shifts focus to dynamic debugging of *logical* errors, especially in complex data science workflows where runtime behavior and data dependencies are key.

LLM Self-Verification Self-correction enhance LLM reliability (Liang et al., 2024). But, LLMs struggle to identify their own errors, especially in complex reasoning (Stechly et al., 2024; Tyen et al., 2024). While some intrinsic self-correction exists (Liu et al., 2024b), its effectiveness for subtle logical errors is debated (Stechly et al., 2024). Approaches to improve self-correction include confidence-guided methods (Li et al., 2024), critique-focused training (Lin et al., 2024).

However, self-verification research mainly targets general language tasks or simplified reasoning. DSDBench uniquely targets dynamic debugging of runtime errors in data science code.

6 Conclusion

We introduced DSDBench, a novel benchmark filling a critical gap in LLM evaluation by focusing on dynamic debugging of logical runtime errors in data science code, specifically multi-hop error tracing and multi-bug detection, built with a rigorous dataset construction process, reveals significant performance limitations of current state-of-the-art LLMs in these complex debugging scenarios.

Acknowledgments

This research was supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant (No. MSS24C004).

Limitations

Our proposed DSDBench benchmark primarily focuses on the data science coding domain. While data science is a complex real-world task, our benchmark can be further expanded to encompass a wider range of practical coding scenarios, enabling a more comprehensive evaluation of LLMs' debugging performance in real-world coding pipelines. Additionally, future work could prioritize investigating LLMs' performance in debugging repository-level code with multi-file dependencies.

Ethical Considerations

To construct the DSDBench benchmark, we employed human annotators for data labeling and verification tasks. We recruited annotators from our research institution holding at least a master degree in Computer Science. All annotators participated voluntarily and were provided with comprehensive information regarding the task's purpose, content, workload, and compensation prior to annotating.

References

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Gemini Team Google. 2024. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *Preprint*, arXiv:2401.03065.

Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi LI, Ruibo Liu, Yue Wang, Shuyue Guo, Xingwei Qu, Xiang Yue, Ge Zhang, Wenhu Chen, and Jie Fu. 2024a. Codeeditorbench: Evaluating code editing capability of large language models. *Preprint*, arXiv:2404.03543.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. 2024b. Ds-agent: Automated data science by empowering large language models with case-based reasoning. *Preprint*, arXiv:2402.17453.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Chenxing Wei, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Li Zhang, Lingyao Zhang, Min Yang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Xiangru Tang, Xiangtao Lu, Xiawu Zheng, Xinbing Liang, Yaying Fei, Yuheng Cheng, Zhibin Gou, Zongze Xu, and Chenglin Wu. 2024. Data interpreter: An llm agent for data science. *Preprint*, arXiv:2402.18679.

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, Yao Cheng, Jianbo Yuan, Jiwei Li, Kun Kuang, Yang Yang, Hongxia Yang, and Fei Wu. 2024. Infiagent-dabench: Evaluating agents on data analysis tasks. *Preprint*, arXiv:2401.05507.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *Preprint*, arXiv:2403.07974.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. 2024. Dsbench: How far are data science agents to becoming data science experts? *Preprint*, arXiv:2409.07703.

Loka Li, Zhenhao Chen, Guangyi Chen, Yixuan Zhang, Yusheng Su, Eric Xing, and Kun Zhang. 2024. Confidence matters: Revisiting intrinsic self-correction capabilities of large language models. *Preprint*, arXiv:2402.12563.

Xun Liang, Shichao Song, Zifan Zheng, Hanyu Wang, Qingchen Yu, Xunkai Li, Rong-Hua Li, Yi Wang, Zhonghao Wang, Feiyu Xiong, and Zhiyu Li. 2024. Internal consistency and self-feedback in large language models: A survey. *Preprint*, arXiv:2407.14507.

Zicheng Lin, Zhibin Gou, Tian Liang, Ruilin Luo, Haowei Liu, and Yujiu Yang. 2024. Criticbench: Benchmarking llms for critique-correct reasoning. *Preprint*, arXiv:2402.14809.

Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. 2024a. Codemind: A framework to challenge large language models for code reasoning. *Preprint*, arXiv:2402.09664.

Dancheng Liu, Amir Nassereldine, Ziming Yang, Chenhui Xu, Yuting Hu, Jiajie Li, Utkarsh Kumar, Changjae Lee, Ruiyang Qin, Yiyu Shi, and Jinjun Xiong. 2024b. Large language models have intrinsic self-correction ability. *Preprint*, arXiv:2406.15673.

Meta. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution. *Preprint*, arXiv:2404.14662.

OpenAI. 2024. Openai o1 system card. *Preprint*, arXiv:2412.16720.

Qwen. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. 2024. On the self-verification limitations of large language models on reasoning and planning tasks. *Preprint*, arXiv:2402.08115.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Hui Haotian, Liu Weichuan, Zhiyuan Liu, and Maosong Sun. 2024. DebugBench: Evaluating debugging capability of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4173–4198, Bangkok, Thailand. Association for Computational Linguistics.

Gladys Tyen, Hassan Mansoor, Victor Cărbune, Peter Chen, and Tony Mak. 2024. Llms cannot find reasoning errors, but can correct them given the error location. *Preprint*, arXiv:2311.08516.

Weiqing Yang, Hanbin Wang, Zhenghao Liu, Xinze Li, Yukun Yan, Shuo Wang, Yu Gu, Minghe Yu, Zhiyuan Liu, and Ge Yu. 2025. Coast: Enhancing the code debugging ability of llms through communicative agent based data synthesis. *Preprint*, arXiv:2408.05006.

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. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 11789–11804, Bangkok, Thailand. Association for Computational Linguistics.

Yaolun Zhang, Yinxu Pan, Yudong Wang, and Jie Cai. 2024a. Pybench: Evaluating llm agent on various real-world coding tasks. *Preprint*, arXiv:2407.16732.

Yuge Zhang, Qiyang Jiang, Xingyu Han, Nan Chen, Yuqing Yang, and Kan Ren. 2024b. Benchmarking data science agents. *Preprint*, arXiv:2402.17168.

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.

Appendix

A Data Annotation Agent

Our automatic data annotation agent is comprised of two components, a self-debugging code agent and an error verifier agent. The prompts used for these agents are in Figure 8, 9, 10.

the code agent receives benchmark questions as input, generate a draft code according to the requirements in the questions. Then, the system environment in which the agent framework operates

SYSTEM PROMPT: `You are a cutting-edge super capable code generation LLM. You will be given a natural language query, generate a runnable python code to satisfy all the requirements in the query. You can use any python library you want. When you complete a plot, remember to save it to a png file.`

USER PROMPT: `Here is the query: """ {{query}} """ If the query requires data manipulation from a csv file, process the data from the csv file and draw the plot in one piece of code. When you complete a plot, remember to save it to a png file. The file name should be """{{file_name}}""".`

Figure 8: The code generation prompt for code agent in Data Annotation.

USER PROMPT: `There are some errors in the code you gave: {{error_message}} please correct the errors. Then give the complete code and don’t omit anything even though you have given it in the above code.`

Figure 9: The self-debugging prompt for code agent in Data Annotation.

executes the draft code. If not successfully executed, the interpreter error message will be passed to the self-debugging code agent, prompting the agent to generate another draft code according to the error message and original benchmark question. The agent will be given a set amount of chances to refine its code according to the error message, if the code is still not executable after 5 rounds, the agent stops. If the code successfully executed within 5 retry times, then the error verifier agent will step in and check the code for further logical errors that may not elicit an interpreter error. If the error verifier agent deems the code correct, the system environment will execute the code and extract the answers from the code. Then we will compare the model generated answers with ground truth answers in each benchmark, if the answers match, we will collect the code that produces these answers as the correct code for our subsequent annotation process.

B Prompts for Error Injection

Figure 11 demonstrates the prompt for error injection, the LLM injector is required to inject plausible runtime logical error into existing correct code with meta information such as benchmark question, data file information. The output format should be a well-formatted JSON dict.

C Error Types

The error types collected in our benchmark are all Python Built-in Exceptions, more information can be accessed at: `https://docs.python.org/3/library/exceptions.html`

D Prompts for Evaluation

Figure 12 and 13 demonstrates the prompts used for evaluating LLMs and LRMs on single bug and multi bug detection. The models are provided with a benchmark question and a snippet of buggy code. The models should identify the error and locate cause and effect error line of code and reproduce error message thrown by the Python Interpreter. The output for single bug detection should be a well-formatted JSON dict, the output for multi bug detection should a list of aforementioned JSON dict.

E Full Evaluation Results

We provide the full results of Single-Bug and Multi-Bug evaluation with all four metrics in Table 7, 8, 9 and 10.

F Prompting Strategies and Test-Time Compute

We evaluate four test-time reasoning strategies beyond direct zero-shot prompting on the *single-bug subset* used for LRMs: *Zero-shot CoT*, *One-shot CoT*, *Self-Refine*, and *Self-Consistency*. All runs disallow external tools and execution to isolate in-context reasoning. See results in Table 11.

Observations. Self-Consistency is most helpful for GPT-4o; DeepSeek-V3 benefits more on effect-line localization; Qwen2.5-72B experiences large gains, reflecting stronger utility from added scaffolds. Despite improvements, the absolute accuracies remain modest, underscoring DSDBench’s difficulty.

G Agentic Systems: Pilot Evaluation

We evaluate whether agentic workflows mitigate DSDBench’s challenges using Cursor Agent (Claude 3.5 Sonnet backend) and OpenHands on the *single-bug subset*. We consider two settings:

- **Single-Attempt Repair (pass@1):** Canonicalize each item into a one-line repair task; the agent may change *exactly one line* once. Success requires a clean reference run.
- **Unconstrained Agentic Workflow:** The agent may iteratively invoke tools and execute code until success (no edit budget).

See results in Table 12

Takeaways. (i) Single-attempt agentic repair outperforms standalone localization, indicating that reframing from diagnosis to generation aligns better with autoregressive training. (ii) Unconstrained agents can brute-force success by extensive regeneration, masking the *reasoning* gap DSDBench is designed to expose. This justifies our focus on line-level localization and semantic message reproduction as complementary to patch-level pass rates.

You will be provided with an original query and a data analysis code. Your task is to:
1. Read the Question carefully, determine whether the code has followed the query requirements, if so, further identify any errors in its data analysis process. If the code faithfully followed seemingly wrong data analysis practices explicitly stated in the Question. Deem it as correct.
2. Explain any errors found, including: Explanation: Explain why this is an error and what issues it may cause. Expected Outcome: Explain how this error will affect the data analysis results, such as misleading outcomes, degraded performance, or incorrect interpretations.
Output Format:
json
"is_error": "true/false",
"error_explanation":
"error_type": "Describe the type of error",
"explanation": "Detailed explanation of why this is an error and its impact",
"expected_outcome": "How this error will affect model performance or results",
"suggestions": "Specific suggestions for fixing the error",
"error_type": "Another error type if multiple errors exist",
"explanation": "Explanation for the second error", "expected_outcome": "Expected outcome for the second error",
"suggestions": "Suggestions for fixing the second error"
Important Notes:
1. Always provide the output in the exact JSON format specified above
2. Set "is_error" to "false" if no errors are found
3. If "is_error" is "false", provide an empty array for error_explanation
4. If "is_error" is "true", include all identified errors in the error_explanation array
5. Consider the original query requirements carefully, if the code follows the query's explicit requirements, even if they seem incorrect, consider it correct

Figure 10: The error verifying prompt in Data Annotation.

You will receive three components:
1. Original Query: A user query that contains specific requirements related to data analysis.
2. Correct Data Analysis Code: A working code snippet designed to analyze the data according to the original query.
3. CSV Information: Details about the structure content and sample data from the CSV file being analyzed.
Your task is to:
1. Identify sklearn and pandas code: Analyze the provided code and extract all lines where sklearn or pandas libraries are used. Organize these lines in a structured format.
2. Inject errors that will cause runtime interruptions: For EACH AND EVERY identified sklearn and pandas lines inject errors with the following guidelines:
Error Type: Inject errors that lead to runtime interruptions such as syntax errors attribute errors type errors or value errors.
Plausibility: The modified lines should still appear logical and plausible at first glance but contain mistakes that will cause the code to fail during execution.
Contextual alignment: Ensure the errors take into account the structure and content of the CSV file to create mistakes that are realistic and aligned with potential data issues.
Impact downstream processes: Errors should trigger runtime interruptions effectively halting the program before it completes execution.
3. Explain each error: For every injected error:
Describe why this is an error and the conditions under which it would fail.
Provide details on the likely runtime error e.g. KeyError ValueError AttributeError etc..
4. Output the structured results:
Provide the original sklearn and pandas code in a structured list.
Include the complete modified code with runtimeinterrupting errors injected.
Clearly explain each injected error in a concise and structured format.
Return your output in the following JSON format:
original_sklearn_pandas_code:
Original sklearn or pandas code line
...
errors:
code: Modified whole code file with the injected error
error_type: Specify the type of runtimeinterrupting error e.g. KeyError ValueError etc.
explanation: Describe why this is an error and the conditions under which it will cause a runtime interruption

Figure 11: The error injection prompt in Data Annotation.

SYSTEM PROMPT: You will be provided with an original query and a data analysis code. Your task is to:
1. Read the question carefully and identify if there are any logic error injected into the code.
2. For each logic error:
- Locate the Cause: Specify the exact line of code that causes the issue.
- Locate the Effect: Identify the line of code where the error will be triggered and the interpreter will throw an error.
- Error Description: Provide a concise description of the error message thrown by the Python Interpreter (not the full traceback).
Output Format:
json
cause_line: Specify the exact line of code causing the issue
effect_line: Specify the exact line of code where the error will be triggered
error_message: Provide a concise description of the error message thrown by the Python Interpreter not the full traceback
There will be only one error in the code. Output only ONE json dict in your response.

Figure 12: The single error evaluation prompt for tested models.

SYSTEM PROMPT: You will be provided with a data analysis code. Your task is to:
1. Read the code carefully and identify all logic errors injected into the code. There will be two or more logic errors in the code.
2. For each logic error you identify:
- Locate the Cause: Specify the exact line of code that causes the issue.
- Locate the Effect: Identify the line of code where the error will be triggered and the interpreter will throw an error or where the incorrect behavior is observed.
- Error Description: Provide a concise description of the error message thrown by the Python Interpreter not the full traceback. Focus on the type of error and the reason if possible from the output.
Output Format:
json
cause_line: Specify the exact line of code causing error 1
effect_line: Specify the exact line of code where error 1 is triggered
error_message: Concise error message for error 1 cause_line: Specify the exact line of code causing error 2
effect_line: Specify the exact line of code where error 2 is triggered
error_message: Concise error message for error 2 ... and so on for all identified errors There will be more than one error in the code. BUT output only ONE json block in your response.

Figure 13: The multi error evaluation prompt for tested models.

Model	Cause Line				Effect Line				Error Type				Error Message			
	P	R	F1	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1	Acc
gpt-4o	39.5	39.0	39.2	39.0	34.7	34.3	34.5	34.3	31.0	30.6	30.8	30.6	31.8	31.4	31.6	31.4
gpt-4o-mini	43.3	40.2	41.7	40.2	25.7	23.9	24.8	23.9	23.4	21.7	22.5	21.7	23.0	21.3	22.1	21.3
claude-3-5-sonnet	45.4	43.7	44.6	43.7	36.6	35.2	35.9	35.2	37.7	36.3	37.0	36.3	35.3	34.0	34.7	34.0
llama-3.1-8b-instant	32.4	25.2	28.4	25.2	18.2	14.2	15.9	14.2	9.9	7.7	8.6	7.7	9.2	7.2	8.0	7.2
llama-3.1-70b-versatile	45.7	42.5	44.0	42.5	31.4	29.3	30.3	29.3	21.9	20.4	21.1	20.4	22.5	20.9	21.7	20.9
llama-3.1-405b-instruct	46.9	41.7	44.1	41.7	35.2	31.3	33.1	31.3	32.9	29.3	31.0	29.3	32.9	29.3	31.0	29.3
Qwen2.5-7B-Instruct	31.0	29.3	30.1	29.3	20.4	19.3	19.8	19.3	11.3	10.7	11.0	10.7	11.6	10.9	11.2	10.9
Qwen2.5-32B-Instruct	43.5	40.9	42.1	40.9	32.4	30.5	31.4	30.5	26.3	24.7	25.5	24.7	26.3	24.7	25.5	24.7
Qwen2.5-72B-Instruct	43.8	41.6	42.6	41.6	38.1	36.2	37.1	36.2	29.0	27.5	28.2	27.5	28.8	27.4	28.1	27.4
deepseek-chat	50.6	48.3	49.4	48.3	36.2	34.5	35.4	34.5	37.6	35.9	36.7	35.9	36.4	34.7	35.5	34.7

Table 7: Overall evaluation results of Single-Bug Detection on DSDBench. P=Precision, R=Recall, F1=F1-Score, Acc=Accuracy.

Model	Cause Line				Effect Line				Error Type				Error Message			
	P	R	F1	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1	Acc
gpt-4o	20.5	20.3	20.4	20.3	10.5	10.4	10.5	10.4	3.6	3.6	3.6	3.6	4.7	4.7	4.7	4.7
gpt-4o-mini	11.3	11.2	11.2	11.2	2.7	2.7	2.7	2.7	2.2	2.2	2.2	2.2	0.8	0.8	0.8	0.8
claude-3-5-sonnet	12.5	12.3	12.4	12.3	4.2	4.1	4.1	4.1	1.9	1.9	1.9	1.9	2.5	2.5	2.5	2.5
llama-3.1-8b-instant	5.1	3.0	3.8	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
llama-3.1-70b-versatile	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
llama-3.1-405b-instruct	24.2	18.6	21.1	18.6	11.0	8.5	9.6	8.5	1.4	1.1	1.2	1.1	3.2	2.5	2.8	2.5
Qwen2.5-7B-Instruct	5.9	4.7	5.2	4.7	1.4	1.1	1.2	1.1	0.3	0.3	0.3	0.3	0.0	0.0	0.0	0.0
Qwen2.5-32B-Instruct	17.6	17.5	17.6	17.5	6.3	6.3	6.3	6.3	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2
Qwen2.5-72B-Instruct	21.4	21.4	21.4	21.4	11.2	11.2	11.2	11.2	3.0	3.0	3.0	3.0	3.6	3.6	3.6	3.6
deepseek-chat	15.2	15.1	15.1	15.1	6.6	6.6	6.6	6.6	3.3	3.3	3.3	3.3	4.7	4.7	4.7	4.7

Table 8: Overall evaluation results of Multi-Bug Detection on DSDBench. P=Precision, R=Recall, F1=F1-Score, Acc=Accuracy.

Model	Cause Line				Effect Line				Error Type				Error Message			
	P	R	F1	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1	Acc
gpt-4o	35.8	35.4	35.6	35.4	31.6	31.2	31.4	31.2	33.7	33.3	33.5	33.3	33.7	33.3	33.5	33.3
gpt-4o-mini	42.7	39.6	41.1	39.6	31.5	29.2	30.3	29.2	27.0	25.0	25.9	25.0	24.7	22.9	23.8	22.9
claude-3-5-sonnet	37.0	35.4	36.2	35.4	27.2	26.0	26.6	26.0	34.8	33.3	34.0	33.3	32.6	31.2	31.9	31.2
llama-3.1-8b-instant	24.1	13.5	17.3	13.5	20.4	11.5	14.7	11.5	11.1	6.2	8.0	6.2	9.3	5.2	6.7	5.2
llama-3.1-70b-versatile	36.7	34.4	35.5	34.4	23.3	21.9	22.6	21.9	20.0	18.8	19.4	18.8	20.0	18.8	19.4	18.8
llama-3.1-405b-instruct	51.2	43.8	47.2	43.8	37.8	32.3	34.8	32.3	36.6	31.2	33.7	31.2	40.2	34.4	37.1	34.4
Qwen2.5-7B-Instruct	30.8	29.2	29.9	29.2	24.2	22.9	23.5	22.9	12.1	11.5	11.8	11.5	13.2	12.5	12.8	12.5
Qwen2.5-32B-Instruct	35.2	32.3	33.7	32.3	28.4	26.0	27.2	26.0	33.0	30.2	31.5	30.2	26.1	24.0	25.0	24.0
Qwen2.5-72B-Instruct	26.7	25.0	25.8	25.0	32.2	30.2	31.2	30.2	30.0	28.1	29.0	28.1	27.8	26.0	26.9	26.0
deepseek-chat	49.4	44.8	47.0	44.8	31.0	28.1	29.5	28.1	37.9	34.4	36.1	34.4	37.9	34.4	36.1	34.4
gemini-2.0-flash	49.4	42.7	45.8	42.7	37.3	32.3	34.6	32.3	38.6	33.3	35.8	33.3	41.0	35.4	38.0	35.4
deepseek-r1	51.6	49.0	50.3	49.0	51.6	49.0	50.3	49.0	56.0	53.1	54.5	53.1	57.1	54.2	55.6	54.2
o1-mini	46.2	43.8	44.9	43.8	38.5	36.5	37.4	36.5	46.2	43.8	44.9	43.8	49.5	46.9	48.1	46.9

Table 9: Comparison with large reasoning models (LRMs) on Single-Bug Detection. P=Precision, R=Recall, F1=F1-Score, Acc=Accuracy.

Model	Cause Line				Effect Line				Error Type				Error Message			
	P	R	F1	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1	Acc
gpt-4o	12.8	12.5	12.7	12.5	5.1	5.0	5.1	5.0	2.6	2.5	2.5	2.5	2.6	2.5	2.5	2.5
gpt-4o-mini	7.5	7.5	7.5	7.5	5.0	5.0	5.0	5.0	2.5	2.5	2.5	2.5	0.0	0.0	0.0	0.0
claude-3-5-sonnet	10.3	10.0	10.1	10.0	7.7	7.5	7.6	7.5	5.1	5.0	5.1	5.0	7.7	7.5	7.6	7.5
llama-3.1-8b-instant	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
llama-3.1-70b-versatile	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
llama-3.1-405b-instruct	23.3	17.5	20.0	17.5	16.7	12.5	14.3	12.5	6.7	5.0	5.7	5.0	6.7	5.0	5.7	5.0
Qwen2.5-7B-Instruct	3.3	2.5	2.9	2.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Qwen2.5-32B-Instruct	17.5	17.5	17.5	17.5	0.0	0.0	0.0	0.0	5.0	5.0	5.0	5.0	2.5	2.5	2.5	2.5
Qwen2.5-72B-Instruct	22.5	22.5	22.5	22.5	17.5	17.5	17.5	17.5	2.5	2.5	2.5	2.5	5.0	5.0	5.0	5.0
deepseek-chat	12.8	12.5	12.7	12.5	7.7	7.5	7.6	7.5	5.1	5.0	5.1	5.0	7.7	7.5	7.6	7.5
o1-mini	37.8	35.0	36.4	35.0	24.3	22.5	23.4	22.5	18.9	17.5	18.2	17.5	18.9	17.5	18.2	17.5
gemini-2.0-flash	21.1	20.0	20.5	20.0	13.2	12.5	12.8	12.5	0.0	0.0	0.0	0.0	2.6	2.5	2.6	2.5
deepseek-r1	32.5	32.5	32.5	32.5	25.0	25.0	25.0	25.0	15.0	15.0	15.0	15.0	15.0	15.0	15.0	15.0

Table 10: Comparison with large reasoning models (LRMs) on Multi-Bug Detection. P=Precision, R=Recall, F1=F1-Score, Acc=Accuracy.

Model	Prompting	Cause Acc.	Effect Acc.	∆ Cause vs. Direct	∆ Effect vs. Direct
GPT-4o	Direct (Baseline)	35.4	31.2	–	–
GPT-4o	Zero-shot CoT	33.3	30.2	-2.1	-1.0
GPT-4o	One-shot CoT	33.3	31.3	-2.1	+0.1
GPT-4o	Self-Refine	33.3	30.2	-2.1	-1.0
GPT-4o	Self-Consistency	37.5	34.4	+2.1	+3.2
DeepSeek-V3	Direct (Baseline)	44.8	28.1	–	–
DeepSeek-V3	Zero-shot CoT	44.8	33.3	+0.0	+5.2
DeepSeek-V3	One-shot CoT	46.9	33.3	+2.1	+5.2
DeepSeek-V3	Self-Refine	47.9	31.3	+3.1	+3.2
DeepSeek-V3	Self-Consistency	42.7	37.5	-2.1	+9.4
Qwen2.5-72B	Direct (Baseline)	26.0	26.0	–	–
Qwen2.5-72B	Zero-shot CoT	37.5	31.3	+11.5	+5.3
Qwen2.5-72B	One-shot CoT	37.5	25.0	+11.5	-1.0
Qwen2.5-72B	Self-Refine	37.5	31.3	+11.5	+5.3
Qwen2.5-72B	Self-Consistency	40.6	36.5	+14.6	+10.5

Table 11: Prompting strategies on the *single-bug subset*. Gains show sensitivity to test-time compute and scaffolding; however, all models remain far from saturation.

Model / Approach	Task	Metric	Performance (%)	Key Insight
Claude-3.5-Sonnet (Standalone)	Cause Line Localization	Accuracy	34.0	Baseline “understanding” is low.
Cursor Agent (Claude-3.5-Sonnet)	Single-Attempt Repair	pass@1	48.9	Recasting to generation yields +14.9 points.
Cursor Agent (Claude-3.5-Sonnet)	Unconstrained Repair	Pass Rate	∼100	Often succeeds via broad, non-minimal rewrites.
OpenHands (Cloud GUI)[†]	Single-Attempt Repair	pass@1	18.6	Large variance across agents/backbones.
OpenHands (Cloud GUI)[†]	Unconstrained Repair	Pass Rate	41.7	Iterative repair remains non-trivial.

Table 12: Agentic pilot on the *single-bug subset*. [†]Backbone LLM unknown for the cloud GUI; results are indicative only.