RepoDebug: Repository-Level Multi-Task and Multi-Language Debugging Evaluation of Large Language Models

Jingjing Liu[1], Zeming Liu[1‡], Zihao Cheng[1], Mengliang He[2], Xiaoming Shi[2], Yuhang Guo[3], Xiangrong Zhu[1], Yuanfang Guo[1], Yunhong Wang[1], Haifeng Wang[4]

[1]School of Computer Science and Engineering, Beihang University, Beijing, China,
[2]East China Normal University, Shanghai, China,
[3]School of Computer Science and Technology, Beijing Institute of Technology,
[4]Baidu Inc., Beijing, China,
`{20373181, zmliu, zihaocheng}@buaa.edu.cn`

Abstract

Large Language Models (LLMs) have exhibited significant proficiency in code debugging, especially in automatic program repair, which may substantially reduce the time consumption of developers and enhance their efficiency. Significant advancements in debugging datasets have been made to promote the development of code debugging. However, these datasets primarily focus on assessing the LLM's function-level code repair capabilities, neglecting the more complex and realistic repository-level scenarios, which leads to an incomplete understanding of the LLM's challenges in repository-level debugging. While several repository-level datasets have been proposed, they often suffer from limitations such as limited diversity of tasks, languages, and error types. To mitigate this challenge, this paper introduces **RepoDebug**, a multi-task and multi-language repository-level code debugging dataset with 22 subtypes of errors that supports 8 commonly used programming languages and 3 debugging tasks. Furthermore, we conduct evaluation experiments on 10 LLMs, where Claude 3.5 Sonnect, the best-performing model, still cannot perform well in repository-level debugging [1].

1 Introduction

Large Language Model (LLM) based code debugging refers to automatically detecting(Zhong et al., 2024b; Zhang et al., 2024a), locating(Bui et al., 2022; Guo et al., 2024), and repairing (Lu et al., 2021; Wang et al., 2023; Shi et al., 2024; Zhong et al., 2024a; Zhao et al., 2024) errors to improve functionality and reliability. It has shown great potential in improving software development efficiency and reducing the time and effort required for software engineering (SE)(Jiang et al., 2024). To evaluate the code debugging performance of large language models (LLMs), various benchmarks have been developed that mainly focus on evaluating the Automatic Program Repair capacity of LLMs(Tian et al., 2024a; Yasunaga and Liang, 2021; Huq et al., 2022). Notably, DebugEval(Yang et al., 2024b) introduces four debugging-related tasks (Bug Location, Bug Identification, Code Review, and Code Repair). Meanwhile, MdEval(Liu et al., 2024a) is a multilingual benchmark for three similar tasks. SWE-Bench(Jimenez et al., 2024) and SWE-PolyBench(Rashid et al., 2025) are repository-level benchmarks that focus on evaluating the end-to-end ability of LLMs to fix issues based on GitHub issue reports.

However, these studies either center on **function-level** code debugging, overlooking the substantial challenges in evaluating the **repository-level** code debugging of LLMs, or lack evaluation diversity on different tasks, languages, and error types.

To mitigate the gap, we propose a multi-task and multi-language repository-level debugging dataset, RepoDebug, which spans 8 programming languages (C, C#, Go, Java, JavaScript, Python, Ruby, and Rust) and 3 tasks (Bug Identification, Bug Localization, and Automatic Program Repair). Following Tian et al. (2024a), RepoDebug is meticulously constructed with 22 distinct subtypes of bugs systematically classified into 4 types: syntax errors, logic errors, reference errors, and multiple errors. Each instance within the RepoDebug comprises a buggy code file, the subtype of the bug, and the precise location of the bug. Specifically, this paper collects data in the RepoDebug from 63 GitHub repositories, all created after 2022, to mitigate data leakage. Of these, 17 repositories are in the test set, and 46 repositories are in the training set. Subsequently, this paper introduces 22 subtypes of bugs to the code files in these repositories by constructing abstract syntax trees using the tree-sitter[2] and

‡ Corresponding Author: Zeming Liu

[1]Our code and dataset will be available at https://github.com/BUAA-IRIP-LLM/RepoDebug.

[2]https://tree-sitter.github.io/tree-sitter

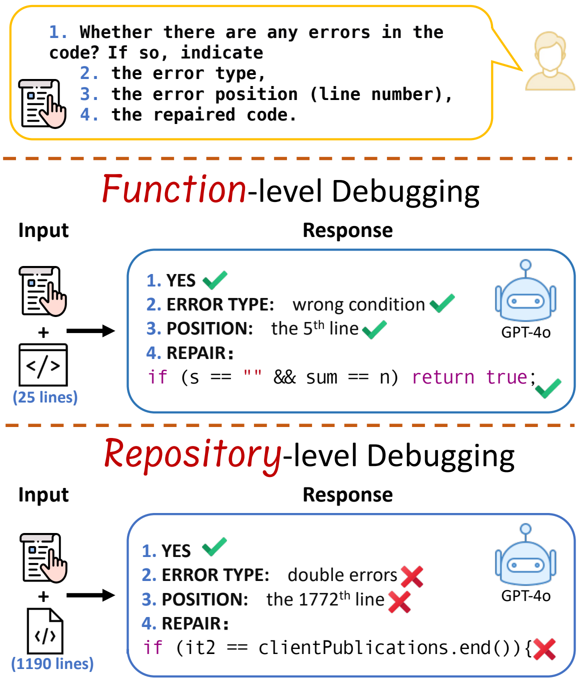

Figure 1: Illustration of code debugging examples with different responses of GPT-4o for function-level and repository-level debugging.

recording their exact locations. To ensure the validity of the bugs, we conduct both automated filtering and manual inspection in the process of collection and construction.

To evaluate the ability of LLMs to debug code errors, this paper utilizes four metrics following Tian et al. (2024a); Jimenez et al. (2024), in which this paper distinguish between the success rate of identifying a single error location and multiple error locations. Following Tian et al. (2024a), the evaluation experiments involve three closed-source models and six open-source models of varying sizes. Finally, the experiment results reveal: (1) Existing large language models exhibit limitations in performance on the RepoDebug dataset. (2) LLMs perform differently in different languages, and the error in Java is easier to detect and repair. (3) Errors of different types vary in difficulty, with multiple errors being the most challenging and syntactic errors being the simplest.

The main contributions are as follows:

- To comprehensively evaluate the repository-level debugging capability of LLMs, we identify a novel code debugging challenge involving multiple tasks, languages, and error types.
- To mitigate this challenge, we construct the first multi-task and multi-language repository-level debugging dataset, **RepoDebug**, which contains 3 tasks, 8 languages, and 22 different subtypes of bugs.
- We evaluate 3 open-source and 7 closed-source models based on RepoDebug. The results demonstrate that even the most advanced models fall short in repository-level debugging, particularly when the number of errors increases and the code length grows.

2 Related Work

This section provides a comprehensive review of the existing benchmarks relevant to our research. Section 2.1 first explores the repository-level benchmarks. Subsequently, section 2.2 delves into the domain of automatic program repair, introducing datasets and benchmarks that assess the ability of LLMs to identify and fix errors in code.

2.1 Repository-Level Benchmark

Repository-level coding tasks have attracted research interest, aiming to assist developers in gathering contextual information within project environments to generate incomplete code(Kondo et al., 2024; Strich et al., 2024; Zhang et al., 2024b; Cheng et al., 2024). These tasks can be categorized into two types based on the nature of the generated content: code completion and code generation. RepoEval(Zhang et al., 2023), RepoBench(Liu et al., 2024b), Stack-Repo(Shrivastava et al., 2023) and ExecRepoBench(Yang et al., 2024a) is an evaluation dataset for code completion tasks. These datasets typically use similarity-based retrieval to query code snippets beneficial for the completion of tasks within a project, to predict the next line of code. EvoCodeBench(Li et al., 2024) and SketchEval(Zan et al., 2024) are collected towards code generation tasks, leveraging the relevant information to achieve function-level or repository-level code generation. Most repository-level benchmarks do not focus on code debugging, while RepoDebug is a dataset specifically designed for debugging completed code.

Furthermore, SWE-Bench(Jimenez et al., 2024), derived from real problems in GitHub software engineering projects, evaluates the model's ability to handle complex problems. However, the source of data in SWE-Bench is limited to 12 widely-used Python libraries. Additionally, it does not categorize error types and evaluate the bug identification and location capacity of LLMs, leaving a challenge

Datasets	RL	Debug Tasks			Languages	Error		#Repos	#Instances
		BI	BL	APR		Types	A.T.		
RepoEval (Zhang et al., 2023)	✓	✗	✗	✗	PY (1)	0	-	14	3,573
RepoBench (Liu et al., 2024b)	✓	✗	✗	✗	JA PY (2)	0	-	1,669	3,636k
Stack-Repo (Shrivastava et al., 2023)	✓	✗	✗	✗	JA (2)	0	-	200	814k
EvoCodeBench (Li et al., 2024)	✓	✗	✗	✗	PY (1)	0	-	25	275
SketchEval (Zan et al., 2024)	✓	✗	✗	✗	PY (1)	0	-	19	1,374
ExecRepoBench (Yang et al., 2024a)	✓	✗	✗	✗	PY (1)	0	-	50	1,200
SWE-Bench (Jimenez et al., 2024)	✓	✗	✗	✓	PY (1)	Un.	-	12	2,294
SWE-PolyBench(Rashid et al., 2025)	✓	✗	✗	✓	PY JA JS TS(4)	Un.	-	21	2,110
DeepFix (Yasunaga and Liang, 2021)	✗	✗	✗	✓	C (1)	4	203	0	6,971
Github-Python (Yasunaga and Liang, 2021)	✗	✗	✗	✓	PY (1)	14	10-128	0	15k
Bug2Fix (Lu et al., 2021)	✗	✗	✗	✓	JA (1)	Un.	$\leq 50/\leq 100$	0	123k
CodeError (Wang et al., 2024)	✗	✗	✗	✓	PY (1)	6	49+31+9	0	4,463
Review4Repair (Huq et al., 2022)	✗	✗	✗	✓	JA (1)	Un.	320+37	0	2,961
FixEval (Anjum Haque et al., 2023)	✗	✗	✗	✓	JA PY (2)	Un.	331/236	0	286k
DebugBench (Tian et al., 2024a)	✗	✗	✗	✓	C++ JA PY (3)	18	468	0	4,253
CodeEditorBench (Guo et al., 2024)	✗	✗	✗	✓	C++ JA PY (3)	14	<1000	0	1,906
DebugEval (Yang et al., 2024b)	✗	✓	✓	✓	C++ JA PY (3)	18	-	0	5,712
MdEval (Liu et al., 2024a)	✗	✓	✗	✓	C C++ C# ... (18)	39	239	0	3,513
FeedbackEval (Dai et al., 2025)	✗	✗	✗	✓	PY(1)	12	-	0	3,736
RepoDebug (Ours)	✓	✓	✓	✓	C C# ...(8)	22	2,124/1,555	63	30,696

Table 1: Comparison between RepoDebug and other datasets. RL indicates repository-level datasets, and BI, BL, and APR indicate three debugging tasks, including Bug Identification, Bug Location, and Automatic Program Repair. Un. means the number of error types is unknown. A.T. refers to the average length of the token, and "-" indicates that there is no exact information available regarding the token length.

in this task. To mitigate this challenge, RepoDebug includes data in 8 languages and supports 3 tasks of code debugging.

2.2 Automatic Program Repair

Recently, automatic program repair (APR) based on LLMs has gained significant attention for its effectiveness and competitiveness(Le Goues et al., 2019). The errors in existing datasets primarily originate from real-world scenarios and large language models.

DeepFix(Yasunaga and Liang, 2021) primarily comprises four types of syntax errors derived from C code compilation errors submitted by students in an introductory programming course. GitHub-Python(Yasunaga and Liang, 2021) is a dataset comprising 3 million Python code snippets, containing 14 types of syntax errors detected by an abstract syntax tree parser. Bug2Fix(Lu et al., 2021) is a Java dataset sourced from GitHub events between March 2011 and October 2017, with variable names normalized at the function level. CodeError(Wang et al., 2024) contains 4,463 Python instances with detailed error information. Review4Repair(Huq et al., 2022) is a Java dataset focusing on code snippets related to code review, with approximately 25.3% of the modifications concerning code repair, categorized into 14 subclasses.

Furthermore, DebugBench(Tian et al., 2024a), CodeEditorBench(Guo et al., 2024), and DEBUGEVAL(Yang et al., 2024b) contain bugs generated from large language models. DebugBench(Tian et al., 2024a) uses code from LeetCode[3] and introduces bugs of 18 types from GPT-4(OpenAI, 2024). CodeEditorBench(Guo et al., 2024) from five sources focus on code editing of errors and other tasks. DebugEval(Yang et al., 2024b) collects data from DebugBench(Tian et al., 2024a) for bug location and bug identification tasks, LiveCodeBench(Jain et al., 2024) for code review tasks, and AtCoder[4] website for code repair. MdEval(Liu et al., 2024a) introduces general and language-specific errors on similar tasks through manual annotation.

However, most existing evaluation datasets focus primarily on assessing the code repair capabilities of models at the function-level, neglecting the repository-level. To fill this gap, we build RepoDebug, a **multi-language and multi-task** benchmark focusing on evaluating the **repository-level** code debugging ability of LLMs.

3 Dataset Construction

In this section, the construction process of RepoDebug is described. As shown in Figure 2, the process begins with collecting project information from GitHub and selecting repositories and

[3] https://leetcode.com/
[4] https://atcoder.jp

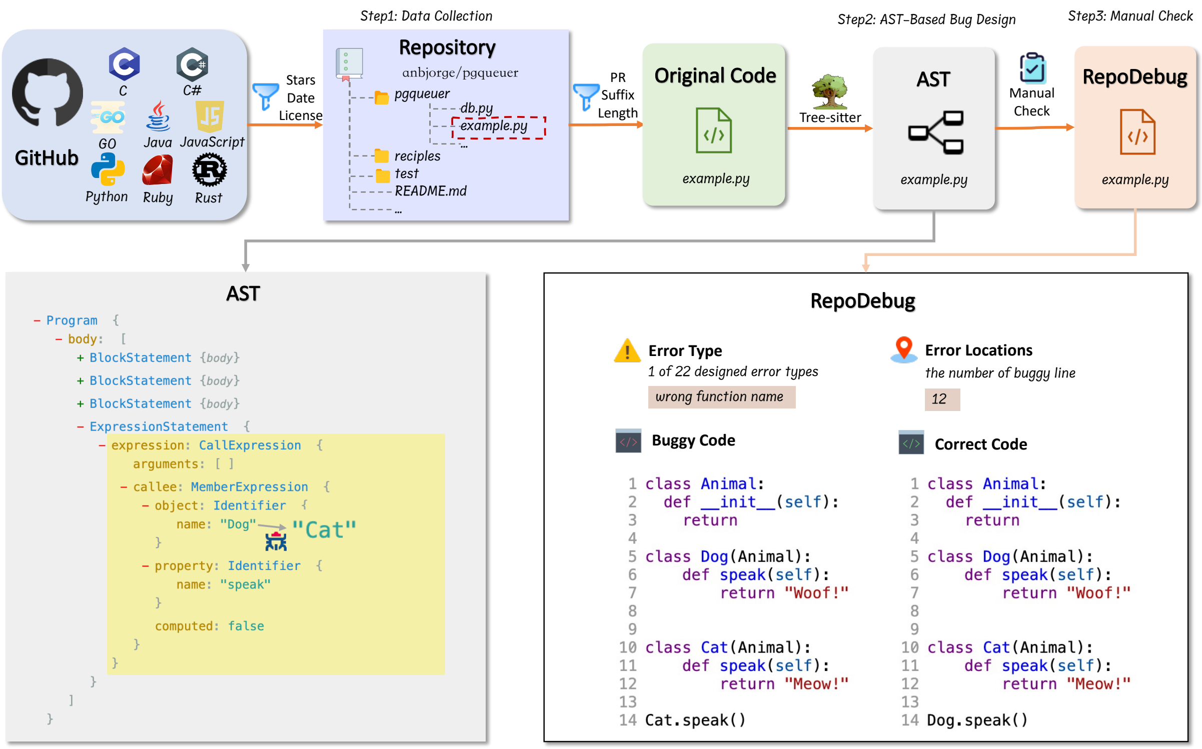

Figure 2: Data construction process of RepoDebug. Firstly, the source data of RepoDebug is collected and filtered from GitHub. Then buggy code is constructed and implanted based on the Abstract Syntax Tree (AST). Additionally, we conduct manual checks to ensure that RepoDebug contains error types, error locations, and pairs of buggy and correct code.

files that meet the specified requirements. The code files are then edited using an AST parser to introduce bugs. Lastly, manual checks are performed to ensure the validity and quality of the data.

3.1 Task Description

In real-world scenarios, developers may not know whether there are errors in the code or the detailed information about the errors (their locations and types). Therefore, models actually lack this information during debugging. To simulate this scenario, we defined a complex task that requires the model to accurately identify errors in the code and provide correct fixes.

Each error instance (C_i, B_i, T_i, L_i) includes the original correct code C_i, buggy code B_i, the error subtype T_i and the error location list $L_i = \{l_{ij} | 1 \leq j \leq 4\}$ containing 1 to 4 line numbers. We provide the model with the buggy code and all possible error subtypes, the model is required to answer whether there are any errors in the buggy code and complete the following three tasks:

(1) predict the error subtype T_i^* of the buggy code;

(2) illustrate the error location list $L_i^* = \{l_{ij}^* | 1 \leq j \leq 4\}$;

(3) repair the buggy code with appropriate edit code $R_i^* = \{r_{ik}^* | k \in L_i^*\}$.

3.2 Data Collection

To obtain high-quality repository-level code data, GitHub is chosen as the data source, and projects are retrieved through the GitHub API. To ensure the accuracy and quality of the original code, the scope of selectable repositories is restricted. Specifically, the repositories in the RepoDebug gain more than 100 stars on GitHub. They are all created after January 1, 2022, to mitigate data leakage. Additionally, we retrieve all the Pull Requests (PR) from these repositories and extract code files with valid modifications and fixed suffixes as the original correct code.

In the dataset construction process of RepoDebug, these standards are followed to insert errors: (1) for reference error and logic error, select instances around the position of the modification in the Pull Requests (PR); (2) sample a maximum of 5 times for each error subtype in one code file; (3) inspired by Karampatsis and

Sutton (2020), the error content involves only one line of code except multiple errors.

A total of 63 projects using the MIT license across 8 specific programming languages are collected, with 46 projects allocated to the training set and 17 to the test set. Additionally, 842 code files are selected to construct instances, of which 727 code files are assigned to the training set and 115 code files to the test set. Based on these code files, there are 34,457 and 5,438 instances in the train and test sets of RepoDebug.

3.3 AST-Based Bug Design

Unlike previous work(Tian et al., 2024a), the buggy code in RepoDebug is not constructed through collection or model generation but using abstract syntax trees (AST). This approach allows precise control over the location of error codes and ensures that the errors affect the project.

As shown in Figure 2, the process begins with using Tree-Sitter to parse the code and construct an abstract syntax tree (AST). The AST's regular structure enables the selection of specific nodes to introduce errors. To achieve this, the corresponding nodes in the syntax tree are queried and randomly selected to determine the positions where the errors will be inserted. Notably, different errors may require different nodes. After collecting these nodes, the corresponding code is analyzed, and modifications are made through string manipulation based on the error subtype. Each error subtype has specific goals for modifying the nodes, with a detailed example provided in the appendix A.

In constructing our error injection dataset, we prioritized five key dimensions to ensure its robustness and applicability: **diversity**, **realism**, **controllability**, **observability**, and **representativeness**. Inspired by DebugBench(Tian et al., 2024a), the dataset encompasses 4 primary types of errors, further divided into 22 subtypes, capturing a wide array of common programming mistakes and is representative of actual developer mistakes. Some of these bug injection patterns are similar with iBiR(Khanfir et al., 2023), which proves that AST-based bugs injection can couple with real ones. This approach allows for precise control over the injection process, ensuring that the modified code remains compilable, which speaks to the dataset's controllability. Moreover, the errors are designed to be detectable by existing test suites, facilitating their observability during testing phases. Compared to errors based on AI, AST-based error preserves the original structural and semantic boundaries of the code, enabling the generation of typical developer mistakes such as mismatched function parameters, or wrong control structures.

The four major types are **syntax**, **reference**, **logic**, and **multiple** errors. Each is designed to capture a different aspect of model robustness, from token-level precision to higher-level semantic reasoning. All errors are injected using AST-guided methods to ensure code remains structurally valid while simulating realistic developer mistakes. More details about the error subtypes are provided in Appendix B.

Syntax errors are token-level faults that violate programming language grammar rules, and RepoDebug includes nine such subtypes. Reference errors occur when an identifier (variable, function, class, or module) is incorrectly used or replaced, and they are categorized into five subtypes in RepoDebug involving minor lexical variations, substitutions with semantically similar or dissimilar identifiers. Logical errors refer to faults involving arithmetic or logical expressions that cause functional anomalies or semantic deviations without triggering compilation failures or syntax violations. Multiple Errors contain 2 to 4 errors, which may occur in either related or unrelated code segments, increasing the diversity and complexity of the fault scenarios.

Our dataset explicitly includes bugs with cross-file effects, such as incorrect import statements or class/function misreferences, where even a single-line modification can affect multiple files. These errors require reasoning beyond the current file, reflecting realistic repository-level debugging challenges (see Appendix C). Moreover, not all injected bugs fail immediately at compilation: we observe that a few buggy instances still compile successfully but introduce hidden logical errors. These errors often corrupt internal states or computations, producing misleading outputs during later execution stages (see Appendix D).

3.4 Manual Check

Following Tian et al. (2024b); Liu et al. (2020); He et al. (2025), manual sampling checks are also conducted on the constructed data. Qualified data must meet the following requirements: (1) The data information must be complete, covering the original correct code, buggy code, error subtype, and the number of error lines; (2) The error modification content must match the error subtype; (3) The buggy code should have an adverse effect during execution. We randomly sample 20 instances from each of the 8 different languages and evaluate them based on three criteria, achieving a pass rate of 100.00%, 100.00% and 98.85%.

3.5 Data Analysis

According to the statistics, syntax errors account for 53.38%, reference errors for 24.37%, logical errors for 6.14%, and multiple errors for 16.08%. In addition, we analyze the length of the token and line in different languages. More details about repositories are in Appendix E.

4 Experiments and Results

This section presents the experiments and results on RepoDebug, including evaluation metrics for LLMs (Section 4.1), baseline models (Section 4.2), the experimental setting (Section 4.3), and experimental results (Section 4.4).

4.1 Metrics

Four key metrics are used to evaluate the performance of LLMs on RepoDebug. Following Yang et al. (2024b), Bug Identification(BI), One Bug's Location(OBL), and All Bugs' Location(ABL) are based on the accuracy to evaluate the effectiveness of existing models in code debugging. Following Liu et al. (2024b); Jimenez et al. (2024), Automatic Program Repair (APR) is typically evaluated using syntactic-level metrics (e.g., Edit Similarity and Exact Match), , and functionality-based semantic metrics (e.g., Pass Rate), providing a comprehensive assessment of both the textual similarity and behavioral correctness of code repairs. More details about the metrics are provided in Appendix F.1.

4.2 Baselines

Following Yang et al. (2024b); Tian et al. (2024a), we conducted evaluation experiments on RepoDebug across three closed-source models and seven open-source models of varying sizes. The closed-source models included GPTs(OpenAI, 2024) (GPT-4o and GPT-4o-mini) and Claude 3.5 Sonnet[5], while the open-source models included Qwen2.5 Coder(Hui et al., 2024) (Qwen2.5 Coder-14b-instruct and Qwen2.5 Coder-7b-instruct), StarCoder2(loz, 2024) (StarCoder2-15b-instruct and StarCoder2-7b) and Deepseek-Coder-V2-16b-lite-instruct(DeepSeek-AI et al., 2024), Code Llama-7b(Rozière et al., 2024) and Deepseek-R1(Guo et al., 2025). More details about the models are provided in Appendix F.2.

4.3 Experimental Setting

The evaluation of proprietary models and DeepSeek R1 is conducted via their official APIs. For other models, we utilize the 4-bit quantized versions provided by the Ollama framework. All experiments are run on a computing cluster equipped with Intel Xeon E5-2620 v4 CPUs, one NVIDIA A100 GPU, and multiple NVIDIA 4090 GPUs, with Ubuntu as the operating system. Notably, Table 2 summarizes the context lengths of models provided by the Ollama framework, ensuring that no input truncation occurred during the evaluation process. Prompt templates are provided in Appendix F.3.

Model	Context Length
Deepseek Coder	163,840
Qwen2.5 Coder	32,768
StarCoder2	16,384
Code Llama	16,384

Table 2: The context length of models.

4.4 Overall Results

We evaluated 10 models on the RepoDebug dataset, and the results shown in Table 3 indicate that current models fall short in repository-level code debugging.

Experimental results reveal substantial performance differences among models. Claude 3.5 Sonnect demonstrates the strongest and most consistent performance across both BI and APR tasks, while DeepSeek R1 also shows competitive results, particularly in bug type classification. The GPT-4o series performs moderately well but exhibits a slight decline across metrics. In contrast, open-source models such as Qwen2.5 Coder, StarCoder2, and Code Llama struggle with syntactically complex or lower-level languages, highlighting limitations in their fine-grained code understanding capabilities.

The results in Table 3 highlight the significant differences in model performance across programming languages. Models generally perform better on high-level languages like Java and JavaScript,

[5]https://www.anthropic.com/news/introducing-claude

Language		GPT 4o	GPT 4o-mini	Claude 3.5 Sonnet	DeepSeek R1	Qwen2.5 Coder 14b	Qwen2.5 Coder 7b	StarCoder2 15b	StarCoder2 7b	Deepseek Coder 16b	Code Llama 7b
C	ACC_{BI}	18.24	16.98	26.42	**54.09**	6.29	4.40	0.63	0.63	5.03	0.00
	ACC_{OBL}	0.00	0.63	5.03	2.52	0.00	0.00	0.00	0.00	**5.66**	0.63
	ACC_{ABL}	0.00	0.00	**4.40**	1.89	0.00	0.00	0.00	0.00	**4.40**	0.63
	$Pass@1$	0.00	0.02	**5.03**	1.72	0.00	0.00	0.00	0.00	0.00	0.00
	ES	0.00	0.09	**5.03**	2.10	0.00	0.00	0.00	0.00	0.31	0.05
	EM	0.00	0.00	**5.03**	1.57	0.00	0.00	0.00	0.00	0.00	0.00
C#	ACC_{BI}	34.00	25.00	37.00	**50.00**	17.00	10.00	0.00	1.00	4.00	0.00
	ACC_{OBL}	2.00	3.00	**22.00**	18.00	3.00	3.00	0.00	0.00	10.00	3.00
	ACC_{ABL}	0.00	2.00	**17.00**	12.00	3.00	3.00	0.00	0.00	4.00	1.00
	$Pass@1$	0.00	1.31	**17.10**	14.05	0.69	0.00	0.02	0.00	0.91	0.01
	ES	0.14	2.16	**19.86**	17.28	2.52	0.20	0.26	0.00	3.24	0.14
	EM	0.00	1.00	**16.00**	12.67	0.00	0.00	0.00	0.00	0.00	0.00
GO	ACC_{BI}	22.53	17.81	**40.34**	33.26	20.17	6.22	0.21	0.21	4.72	0.21
	ACC_{OBL}	1.50	2.58	**9.66**	0.86	2.58	2.36	0.21	0.00	7.30	1.29
	ACC_{ABL}	0.43	1.50	**7.30**	0.43	1.72	1.29	0.21	0.00	4.72	1.29
	$Pass@1$	0.10	1.06	**4.08**	0.46	0.62	0.23	0.00	0.00	0.36	0.10
	ES	0.34	1.58	**7.33**	0.56	1.63	0.96	0.07	0.00	1.23	0.34
	EM	0.11	0.97	**2.79**	0.43	0.21	0.00	0.00	0.00	0.00	0.00
Java	ACC_{BI}	40.66	30.20	**55.78**	47.46	24.99	10.46	0.41	0.23	8.54	1.28
	ACC_{OBL}	9.32	3.02	**16.63**	5.21	4.11	2.15	0.09	0.09	10.83	1.96
	ACC_{ABL}	5.85	1.46	**11.01**	3.75	2.19	1.14	0.05	0.05	5.89	0.91
	$Pass@1$	4.47	1.25	**13.58**	3.69	0.86	0.21	0.00	0.00	0.75	0.07
	ES	6.47	1.93	**14.68**	4.77	2.16	0.65	0.02	0.01	2.09	0.29
	EM	3.61	1.03	**13.16**	3.36	0.39	0.05	0.00	0.00	0.23	0.00
JavaScript	ACC_{BI}	27.47	20.51	43.22	**55.68**	11.36	8.06	1.47	0.00	4.76	0.00
	ACC_{OBL}	12.45	6.59	**18.68**	12.09	4.76	5.49	0.00	0.73	13.55	1.10
	ACC_{ABL}	10.62	4.76	**15.02**	10.62	3.66	4.03	0.00	0.73	10.62	0.73
	$Pass@1$	5.28	2.40	**15.75**	8.68	1.94	1.02	0.00	0.00	1.40	0.12
	ES	10.58	4.66	**17.84**	11.04	4.15	2.70	0.00	0.07	3.22	0.43
	EM	3.05	1.47	**14.90**	7.69	1.10	0.37	0.00	0.00	0.73	0.00
Python	ACC_{BI}	23.54	20.43	36.77	**42.61**	14.01	3.89	0.19	0.39	8.95	0.19
	ACC_{OBL}	6.23	3.11	**8.17**	6.81	3.11	1.56	0.00	0.39	7.59	0.78
	ACC_{ABL}	4.28	1.95	**6.42**	5.25	2.53	0.78	0.00	0.39	5.25	0.58
	$Pass@1$	3.02	1.06	**6.15**	5.51	1.13	0.09	0.00	0.00	0.51	0.00
	ES	4.31	2.00	**6.63**	5.90	2.16	0.46	0.00	0.03	1.78	0.07
	EM	2.53	0.78	**6.03**	5.45	0.78	0.00	0.00	0.00	0.00	0.00
Ruby	ACC_{BI}	26.51	21.53	33.63	**43.06**	20.11	5.52	0.00	0.36	10.32	0.00
	ACC_{OBL}	9.25	5.34	**16.37**	14.59	6.58	2.67	0.00	0.18	12.63	1.07
	ACC_{ABL}	4.80	3.02	**11.21**	10.32	4.27	1.42	0.00	0.00	8.90	0.89
	$Pass@1$	1.80	0.81	**13.77**	8.89	1.37	0.33	0.00	0.00	0.59	0.01
	ES	6.13	3.03	**15.06**	12.65	4.88	1.40	0.00	0.00	2.07	0.17
	EM	0.09	0.00	**13.35**	7.41	0.00	0.00	0.00	0.00	0.00	0.00
Rust	ACC_{BI}	20.17	11.83	27.23	**36.00**	7.15	2.98	0.26	0.00	6.21	0.00
	ACC_{OBL}	4.68	1.96	**8.09**	4.34	2.30	1.45	0.00	0.09	7.40	0.77
	ACC_{ABL}	3.40	1.28	**5.70**	3.23	1.28	0.85	0.00	0.09	5.28	0.43
	$Pass@1$	2.67	0.77	**6.99**	3.93	0.79	0.14	0.00	0.00	0.56	0.00
	ES	3.46	1.44	**7.60**	4.18	1.30	0.47	0.00	0.00	1.39	0.14
	EM	2.38	0.60	**6.84**	3.83	0.68	0.09	0.00	0.00	0.23	0.00

Table 3: Results for different languages. **Bold** indicates the best, underline indicates the second best.

but struggle with lower-level or statically typed languages such as C and Rust. This can be attributed to two main factors: the inherent complexity of low-level languages, including strict syntax and manual memory management, and the training data bias that models are typically trained on corpora with greater representation of high-level languages, leading to uneven generalization across language types.

4.5 Data Leakage

To further analyze the impact of data leakage, we have separately evaluated the performance on instances before and after April 2024. The results in Table 5 show that Claude 3.5 Sonnet performs worse on instances after April 2024 than before. Therefore, data leakage could be a factor that influences the performance of Claude 3.5 Sonnet.

4.6 Qualitative Analysis

We selected several instances from the models' generated outputs to help assess their ability to under-

Type		GPT 4o	GPT 4o-mini	Claude 3.5 Sonnect	DeepSeek R1	Qwen2.5 Coder 14b	Qwen2.5 Coder 7b	StarCoder2 15b	StarCoder2 7b	Deepseek Coder 16b	Code Llama 7b
Syntax	ACC_{BI}	39.24	34.55	<u>54.15</u>	**60.18**	26.42	10.13	0.45	0.24	12.61	0.96
	ACC_{OBL}	5.03	2.38	**11.16**	5.86	3.03	1.41	0.03	0.17	<u>7.03</u>	1.03
	ACC_{ABL}	5.03	2.38	**11.16**	5.86	3.03	1.41	0.03	0.17	<u>7.03</u>	1.03
	$Pass@1$	2.36	1.15	**9.76**	<u>4.65</u>	0.83	0.11	0.00	0.00	0.51	0.00
	ES	4.01	2.05	**10.56**	<u>5.56</u>	2.00	0.52	0.01	0.01	1.40	0.14
	EM	1.69	0.90	**9.51**	<u>4.36</u>	0.38	0.03	0.00	0.00	0.24	0.00
Reference	ACC_{BI}	30.24	12.67	**39.89**	<u>32.96</u>	11.61	3.62	0.30	0.23	1.21	0.08
	ACC_{OBL}	7.24	1.58	**11.69**	4.68	1.89	1.51	0.08	0.08	<u>7.32</u>	0.83
	ACC_{ABL}	7.24	1.58	**11.69**	4.68	1.89	1.51	0.08	0.08	<u>7.32</u>	0.83
	$Pass@1$	<u>4.52</u>	0.80	**9.49**	3.26	0.59	0.25	0.00	0.00	0.43	0.02
	ES	<u>5.96</u>	1.20	**10.75**	4.11	1.20	0.71	0.03	0.01	1.36	0.24
	EM	<u>3.92</u>	0.68	**8.97**	3.02	0.45	0.15	0.00	0.00	0.08	0.00
Logic	ACC_{BI}	24.55	13.17	<u>51.20</u>	**51.50**	11.68	2.10	0.60	0.00	4.49	0.00
	ACC_{OBL}	0.90	1.20	**6.89**	2.99	1.20	1.50	0.00	0.00	<u>4.79</u>	0.30
	ACC_{ABL}	0.90	1.20	**6.89**	2.99	1.20	1.50	0.00	0.00	<u>4.79</u>	0.30
	$Pass@1$	0.43	0.36	**4.84**	<u>1.63</u>	0.21	0.11	0.00	0.00	0.26	0.00
	ES	0.79	0.84	**5.68**	<u>2.79</u>	1.00	0.47	0.00	0.00	0.97	0.04
	EM	0.30	0.30	**4.49**	<u>1.20</u>	0.00	0.00	0.00	0.00	0.00	0.00
Multiple	ACC_{BI}	2.06	0.23	<u>3.66</u>	1.14	0.91	**3.89**	0.00	0.23	1.60	0.11
	ACC_{OBL}	16.11	8.57	**24.80**	11.31	9.26	5.71	0.11	0.23	<u>23.66</u>	3.77
	ACC_{ABL}	0.34	0.23	0.11	<u>1.03</u>	0.46	0.11	0.00	0.00	**1.49**	0.11
	$Pass@1$	0.05	0.00	**0.11**	<u>0.09</u>	0.00	0.00	0.00	0.00	0.00	0.00
	ES	8.19	3.51	**19.65**	<u>9.49</u>	5.15	1.82	0.04	0.00	4.48	0.54
	EM	3.12	0.80	**16.23**	<u>6.38</u>	0.86	0.00	0.00	0.00	0.19	0.00

Table 4: Results for different error types. **Bold** indicates the best, <u>underline</u> indicates the second best.

Metric	**Before**	**After**
ACC_{BI}	41.88	43.15
ACC_{OBL}	13.73	12.39
ACC_{ABL}	9.39	9.02
$Pass@1$	4.48	4.45
ES	12.38	10.78
EM	11.28	8.33

Table 5: Comparison of error distributions before and after April 2024.

stand and debug code. We discuss a case with extreme responses from a Go project to illustrate the models' performance, along with our overall findings. More details are in Appendix G.1.

The experimental results reveal several consistent patterns in model behavior. Specifically, the models demonstrate a stronger capacity for generating plausible fixes than for accurately localizing the underlying errors. Furthermore, the model outputs frequently exhibit a tendency to over-identify errors, often introducing additional spurious corrections beyond those present in the original code.

5 Analysis

To comprehensively investigate the factors influencing the repository-level code debugging capabilities of large language models, we employ three research questions (RQs) for further analysis: (1) examining the influence of various error types on repository-level code debugging (**RQ1**); and (2) investigating the effect of the number of errors on repository-level code debugging (**RQ2**); and (3) analyzing the impact of different token length on repository-level code debugging (**RQ3**).

5.1 RQ1: How does error type influence repository-level code debugging?

We compared the performance of different models on four error types: syntax errors, reference errors, logic errors, and multiple errors. As shown in Table 4, the results highlight that models' performance is highly dependent on error type, with syntax errors being the easiest and multiple errors the most challenging. For example, Claude 3.5 Sonnect achieves 54.15% for syntax errors and only 3.66% for multiple errors in the BI task. This result may be due to the models' limited ability to understand code semantics and handle multiple error types simultaneously.

5.2 RQ2: How does the number of errors affect repository-level code debugging?

Compared to single errors, multiple errors are more challenging to fully repair due to their complexity and interdependencies. However, large language models can play a more significant role in such cases. As shown in Figure 4, the OBL results for multiple errors are significantly better than those for single errors. For example, Claude 3.5 Sonnect achieves 24.80% for multiple errors, while only 11.16%, 11.69%, and 6.89% for three single errors. This indicates that models can incrementally address multiple errors, increasing the likelihood of partial but effective repairs. More details can be found in Appendix G.3.

5.3 RQ3: How does the length of code influence LLMs' performance?

An increase in code length adversely affects the performance of the model. We analyze the debugging performance of models on code with different lengths, ranging from below 500 to 10,000 tokens. The performance of most models is significantly lower for long tokens (larger than 500) compared to short tokens (less than 500). For example, Claude 3.5 Sonnect achieves 51.48%, 20.66%, 13.06%, 11.42%, 18.18% and 16.53% for short tokens (less than 500), while dropping to 43.07%, 13.47%, 9.43%, 7.68%, 11.99% and 10.34% for tokens with lengths between 500 and 1000. This demonstrates that the length of the code limits the model's performance in code debugging. More details can be found in Appendix G.2.

6 Conclusion

This work identifies the significant challenge of repository-level code debugging across multiple tasks and languages. To mitigate this challenge, we introduce a novel task focused on error identification, localization, and repair for repository-level code. We also present the first multi-task and multi-language dataset for repository-level code debugging, `RepoDebug`. To facilitate further research, we conducted comprehensive benchmarking experiments on `RepoDebug`, and the results highlight the limitations of LLMs on repository-level code debugging. We hope that `RepoDebug` will serve as a valuable dataset to promote future research in repository-level code debugging.

Acknowledgements

Thanks for the insightful comments and feedback-from the reviewers. This work was supported by the National Natural Science Foundation of China(No. 62406015), and CCF-Baidu Open Fund (No.CCF-BAIDU202411).

Limitations

Repository-level code evaluation requires models to understand and process long-range contextual information, which presents significant challenges. Consequently, the complexity and scale of repository-level tasks might demand both advanced model architectures and high-performance computing environments.

Ethical Statement

The code incorporated within `RepoDebug` is exclusively sourced from publicly accessible repositories, and each repository is associated with an MIT license that explicitly permits its utilization. Throughout the process of collection and evaluation, our methodology was strictly confined to the utilization of the original code and fundamental project metadata. No user-specific or private information was accessed or utilized in any capacity. The data selection and screening methodologies employed in this study are devoid of any discriminatory or biased practices. The data construction process has been meticulously designed to ensure equitable treatment of each repository and code file, thereby maintaining objectivity and fairness. This research is conducted in strict adherence to the ethical guidelines governing AI development. The primary objective of this endeavor is to contribute positively to the advancement of the field by providing a comprehensive and unbiased dataset for further exploration and analysis.

References

2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2023. Fixeval: Execution-based evaluation of program fixes for programming problems. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 11–18, Melbourne, Australia. IEEE.

Nghi Bui, Yue Wang, and Steven C.H. Hoi. 2022. Detect-localize-repair: A unified framework for learning to debug with codet5. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 812–823, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflow-guided retrieval augmentation for repository-level code completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7957–7977, Bangkok, Thailand. Association for Computational Linguistics.

Dekun Dai, MingWei Liu, Anji Li, Jialun Cao, Yanlin Wang, Chong Wang, Xin Peng, and Zibin Zheng. 2025. Feedbackeval: A benchmark for evaluating large language models in feedback-driven code repair tasks. *Preprint*, arXiv:2504.06939.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *Preprint*, arXiv:2406.11931.

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*.

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. 2024. Codeeditorbench: Evaluating code editing capability of large language models. *arXiv preprint*.

Mengliang He, Jiayi Zeng, Yankai Jiang, Wei Zhang, Zeming Liu, Xiaoming Shi, and Aimin Zhou. 2025. Flow2Code: Evaluating large language models for flowchart-based code generation capability. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 8124–8146, Vienna, Austria. Association for Computational Linguistics.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Faria Huq, Masum Hasan, Md Mahim Anjum Haque, Sazan Mahbub, Anindya Iqbal, and Toufique Ahmed. 2022. Review4repair: Code review aided automatic program repairing. *Information and Software Technology*, 143:106765.

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. *arXiv preprint arXiv:2403.07974*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *Preprint*, arXiv:2406.00515.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Rafael-Michael Karampatsis and Charles Sutton. 2020. How often do single-statement bugs occur?: The manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 573–577. ACM.

Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawende F. Bissyandé, Jacques Klein, and Yves Le Traon. 2023. ibir: Bug-report-driven fault injection. *ACM Trans. Softw. Eng. Methodol.*, 32(2).

Mizuki Kondo, Daisuke Kawahara, and Toshiyuki Kurabayashi. 2024. Improving repository-level code search with text conversion. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 4: Student Research Workshop)*, pages 130–137, Mexico City, Mexico. Association for Computational Linguistics.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM*, 62(12):56–65.

Jia Li, Ge Li, Xuanming Zhang, YunFei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. 2024. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. In *Advances in Neural Information Processing Systems*, volume 37, pages 57619–57641. Curran Associates, Inc.

Shukai Liu, Linzheng Chai, Jian Yang, Jiajun Shi, He Zhu, Liran Wang, Ke Jin, Wei Zhang, Hualei Zhu, Shuyue Guo, Tao Sun, Jiaheng Liu, Yunlong Duan, Yu Hao, Liqun Yang, Guanglin Niu, Ge Zhang, and Zhoujun Li. 2024a. Mdeval: Massively multilingual code debugging. *arXiv preprint*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024b. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*.

Zeming Liu, Haifeng Wang, Zheng-Yu Niu, Hua Wu, Wanxiang Che, and Ting Liu. 2020. Towards conversational recommendation over multi-type dialogs. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1036–1049, Online. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *Preprint*, arXiv:2102.04664.

OpenAI. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. 2025. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *Preprint*, arXiv:2504.08703.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

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

Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. Repofusion: Training code models to understand your repository. *Preprint*, arXiv:2306.10998.

Jan Strich, Florian Schneider, Irina Nikishina, and Chris Biemann. 2024. On improving repository-level code QA for large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pages 209–244, Bangkok, Thailand. Association for Computational Linguistics.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Hui Haotian, Liu Weichuan, Zhiyuan Liu, and Maosong Sun. 2024a. 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.

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

Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2024. INTERVENOR: Prompting the coding ability of large language models with the interactive chain of repair. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2081–2107, Bangkok, Thailand. Association for Computational Linguistics.

Weishi Wang, Yue Wang, Steven Hoi, and Shafiq Joty. 2023. Towards low-resource automatic program repair with meta-learning and pretrained language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6954–6968, Singapore. Association for Computational Linguistics.

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

Weiqing Yang, Hanbin Wang, Zhenghao Liu, Xinze Li, Yukun Yan, Shuo Wang, Yu Gu, Minghe Yu, Zhiyuan Liu, and Ge Yu. 2024b. Enhancing the code debugging ability of llms via communicative agent based data refinement. *Preprint*, arXiv:2408.05006.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 11941–11952. PMLR.

Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Wei Li, Yafen Yao, Yongshun Gong, Xiaolin Chen, Bei Guan, Zhiguang Yang, Yongji Wang, Qianxiang Wang, and Lizhen Cui. 2024. Codes: Natural language to code repository via multi-layer sketch. *Preprint*, arXiv:2403.16443.

Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2024a. Prompt-enhanced software vulnerability detection using chatgpt. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 276–277, Lisbon Portugal. ACM.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024b. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, Bangkok, Thailand. Association for Computational Linguistics.

Yuze Zhao, Zhenya Huang, Yixiao Ma, Rui Li, Kai Zhang, Hao Jiang, Qi Liu, Linbo Zhu, and Yu Su. 2024. RePair: Automated program repair with process-based feedback. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 16415–16429, Bangkok, Thailand. Association for Computational Linguistics.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024a. 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.

Zhiyuan Zhong, Sinan Wang, Hailong Wang, Shaojin Wen, Hao Guan, Yida Tao, and Yepang Liu. 2024b. Advancing bug detection in fastjson2 with large language models driven unit test generation. *arXiv preprint*.

A Example for introducing a bug.

This section shows an example to introduce a bug involving a wrong function call in correct code using an abstract syntax tree. There are four main steps to do this: (1) filter a code file; (2) build an abstract syntax tree; (3) find the required nodes; (4) generate buggy code.

A.1 Filter a Code File.

The repository, janbjorge/pgqueuer[6], is created on 2024-04-19 and has 1,204 stars in GitHub, which meets the screening criteria of RepoDebug. Then we filter a specific code file (pgqueuer/types.py) serving as the source code from the pull requests of this repository.

```python
from __future__ import annotations
from typing import Literal, NewType
Channel = NewType("Channel",  str)
PGChannel = Channel
OPERATIONS = Literal["insert", "update", "delete", "truncate"]
EVENT_TYPES = Literal["table_changed_event", "requests_per_second_event", "cancellation_event"]
JobId = NewType("JobId",  int)
JOB_STATUS = Literal[
    "queued",
    "picked",
    "successful",
    "canceled",
    "deleted",
    "exception",
]
CronEntrypoint = NewType(
    "CronEntrypoint",
     str
)
CronExpression = NewType(
    "CronExpression",
     str
)
ScheduleId = NewType(
    "ScheduleId",
     int
)
```

Listing 1: Original code file (pgqueuer/types.py).

[6] https://github.com/janbjorge/pgqueuer/tree/main

A.2 Build an Abstract Syntax Tree.

We utilize a Tree-sitter to parse the Listing 1, and the structure of the generated abstract syntax tree (AST) is hierarchical and node-based. As shown in Listing 2, the root node (module) represents the entire code file or program. Beneath the root, there are various child nodes representing different constructs such as future_import_statement, import_from_statement, and expression_statement.

```
module [0, 0] - [27, 0]
  future_import_statement [0, 0] - [0, 34]
    name: dotted_name [0, 23] - [0, 34]
      identifier [0, 23] - [0, 34]
  import_from_statement [1, 0] - [1, 35]
    module_name: dotted_name [1, 5] - [1, 11]
      identifier [1, 5] - [1, 11]
    name: dotted_name [1, 19] - [1, 26]
      identifier [1, 19] - [1, 26]
    name: dotted_name [1, 28] - [1, 35]
      identifier [1, 28] - [1, 35]
  expression_statement [2, 0] - [2, 33]
    assignment [2, 0] - [2, 33]
      left: identifier [2, 0] - [2, 7]
      right: call [2, 10] - [2, 33]
        function: identifier [2, 10] - [2, 17]
        arguments: argument_list [2, 17] - [2, 33]
          string [2, 18] - [2, 27]
            string_start [2, 18] - [2, 19]
            string_content [2, 19] - [2, 26]
            string_end [2, 26] - [2, 27]
          identifier [2, 29] - [2, 32]
  expression_statement [3, 0] - [3, 19]
    assignment [3, 0] - [3, 19]
```

```
    left: identifier [3, 0] - [3, 9]
    right: identifier [3, 12] - [3, 19]
expression_statement [4, 0] - [4, 62]
  assignment [4, 0] - [4, 62]
    left: identifier [4, 0] - [4, 10]
    right: subscript [4, 13] - [4, 62]
      value: identifier [4, 13] - [4, 20]
      subscript: string [4, 21] - [4, 29]
        string_start [4, 21] - [4, 22]
        string_content [4, 22] - [4, 28]
        string_end [4, 28] - [4, 29]
      subscript: string [4, 31] - [4, 39]
        string_start [4, 31] - [4, 32]
        string_content [4, 32] - [4, 38]
        string_end [4, 38] - [4, 39]
      subscript: string [4, 41] - [4, 49]
        string_start [4, 41] - [4, 42]
        string_content [4, 42] - [4, 48]
        string_end [4, 48] - [4, 49]
      subscript: string [4, 51] - [4, 61]
        string_start [4, 51] - [4, 52]
        string_content [4, 52] - [4, 60]
        string_end [4, 60] - [4, 61]
expression_statement [5, 0] - [5, 95]
  assignment [5, 0] - [5, 95]
    left: identifier [5, 0] - [5, 11]
    right: subscript [5, 14] - [5, 95]
      value: identifier [5, 14] - [5, 21]
      subscript: string [5, 22] - [5, 43]
        string_start [5, 22] - [5, 23]
        string_content [5, 23] - [5, 42]
        string_end [5, 42] - [5, 43]
      subscript: string [5, 45] - [5, 72]
        string_start [5, 45] - [5, 46]
        string_content [5, 46] - [5, 71]
        string_end [5, 71] - [5, 72]
      subscript: string [5, 74] - [5, 94]
        string_start [5, 74] - [5, 75]
        string_content [5, 75] - [5, 93]
        string_end [5, 93] - [5, 94]
expression_statement [6, 0] - [6, 29]
  assignment [6, 0] - [6, 29]
    left: identifier [6, 0] - [6, 5]
    right: call [6, 8] - [6, 29]
      function: identifier [6, 8] - [6, 15]
      arguments: argument_list [6, 15] - [6,
          29]
        string [6, 16] - [6, 23]
          string_start [6, 16] - [6, 17]
          string_content [6, 17] - [6, 22]
          string_end [6, 22] - [6, 23]
        identifier [6, 25] - [6, 28]
expression_statement [7, 0] - [14, 1]
  assignment [7, 0] - [14, 1]
    left: identifier [7, 0] - [7, 10]
    right: subscript [7, 13] - [14, 1]
      value: identifier [7, 13] - [7, 20]
      subscript: string [8, 4] - [8, 12]
        string_start [8, 4] - [8, 5]
        string_content [8, 5] - [8, 11]
        string_end [8, 11] - [8, 12]
      subscript: string [9, 4] - [9, 12]
        string_start [9, 4] - [9, 5]
        string_content [9, 5] - [9, 11]
        string_end [9, 11] - [9, 12]
      subscript: string [10, 4] - [10, 16]
        string_start [10, 4] - [10, 5]
        string_content [10, 5] - [10, 15]
        string_end [10, 15] - [10, 16]
      subscript: string [11, 4] - [11, 14]
        string_start [11, 4] - [11, 5]
        string_content [11, 5] - [11, 13]
        string_end [11, 13] - [11, 14]
      subscript: string [12, 4] - [12, 13]
        string_start [12, 4] - [12, 5]
        string_content [12, 5] - [12, 12]
        string_end [12, 12] - [12, 13]
      subscript: string [13, 4] - [13, 15]
        string_start [13, 4] - [13, 5]
        string_content [13, 5] - [13, 14]
        string_end [13, 14] - [13, 15]
expression_statement [15, 0] - [18, 1]
  assignment [15, 0] - [18, 1]
    left: identifier [15, 0] - [15, 14]
    right: call [15, 17] - [18, 1]
      function: identifier [15, 17] - [15, 24]
      arguments: argument_list [15, 24] - [18,
          1]
        string [16, 4] - [16, 20]
          string_start [16, 4] - [16, 5]
          string_content [16, 5] - [16, 19]
          string_end [16, 19] - [16, 20]
        identifier [17, 4] - [17, 7]
expression_statement [19, 0] - [22, 1]
  assignment [19, 0] - [22, 1]
    left: identifier [19, 0] - [19, 14]
    right: call [19, 17] - [22, 1]
      function: identifier [19, 17] - [19, 24]
      arguments: argument_list [19, 24] - [22,
          1]
        string [20, 4] - [20, 20]
          string_start [20, 4] - [20, 5]
          string_content [20, 5] - [20, 19]
          string_end [20, 19] - [20, 20]
        identifier [21, 4] - [21, 7]
expression_statement [23, 0] - [26, 1]
  assignment [23, 0] - [26, 1]
    left: identifier [23, 0] - [23, 10]
    right: call [23, 13] - [26, 1]
      function: identifier [23, 13] - [23, 20]
      arguments: argument_list [23, 20] - [26,
          1]
        string [24, 4] - [24, 16]
          string_start [24, 4] - [24, 5]
          string_content [24, 5] - [24, 15]
          string_end [24, 15] - [24, 16]
        identifier [25, 4] - [25, 7]
```

Listing 2: Example for an abstract syntax tree. Blue refers to call nodes, and red refers to function nodes.

A.3 Find the Required nodes.

To locate specific types of nodes, RepoDebug employs particular queries to interrogate the abstract syntax tree. The following Listing 3 is an example for querying the abstract syntax tree to find the function call statement. As the parser detects in Listing 4, there are 5 matching statements in the Listing 2 whose function name is NewType, located on lines 3, 7, 16, 20 and 24 respectively.

```
(call
 function:(identifier)@function
)@call
```

Listing 3: Query to extract call nodes. “@” stands for the nodes required.

```
3 NewType("Channel", str)
7 NewType("ScheduleId", int)
16 NewType("CronEntrypoint", str)
20 NewType("CronExpression", str)
24 NewType("JobId", int)
```

Listing 4: Query results to extact call nodes from abstract syntax tree.

A.4 Generate Buggy Code.

To introduce the wrong function call to the Listing 1, it is critical to locate the exact position of the function name. For example, the first function call node detected is in the third line and it starts at `[2, 10]` and ends at `[2, 33]`, with the function name specifically located between `[2, 10]` and `[2, 17]`. Then, `NewType` within this range is replaced by another identifier, `Literal`. Listing 5 presents the full modified code file, which includes a wrong function call error.

```
1 from __future__ import annotations
2 from typing import Literal, NewType
3 Channel = Literal ("Channel",  str)
4 PGChannel = Channel
5 OPERATIONS = Literal["insert", "
    update", "delete", "truncate"]
6 EVENT_TYPES = Literal["
    table_changed_event", "
    requests_per_second_event", "
    cancellation_event"]
7 JobId = NewType("JobId",  int)
8 JOB_STATUS = Literal[
9     "queued",
10     "picked",
11     "successful",
12     "canceled",
13     "deleted",
14     "exception",
15 ]
16 CronEntrypoint = NewType(
17     "CronEntrypoint",
18      str
19 )
20 CronExpression = NewType(
21     "CronExpression",
22      str
23 )
24 ScheduleId = NewType(
25     "ScheduleId",
26      int
27 )
```

Listing 5: Buggy code file (pgqueuer/types.py). **Red background** refers to buggy context.

B Details for Different Subtypes

As shown in Table 6, we construct 22 distinct error subtypes and provide their detailed definitions and examples in this section.

B.1 Syntax Error

Misuse equal sign. We design two formats to misuse equal signs. The first one queries a single equal sign in the assignment statement and replaces it with a double equal sign, such as Figure 3a . The second one queries a double equals sign in a conditional statement and replaces it with a single equal sign.

Open parenthesis, open bracket, and open brace. Parentheses, square brackets, and braces are widely used in programming for structuring code and representing data, where they play a role in distinguishing code blocks and data structures. We use an abstract syntax tree parser to identify them in the code and remove them from the original code. Finally, the buggy code misses a closing “]” like Figure 3b.

Missing colon, missing comma, and missing semicolon. We query the specific colon, comma, or semicolon in the AST and remove them, such as Figure 3c. These errors may occur in reference statements, conditional statements, or loop statements. Their absence may lead to compilation errors or two parallel variables being merged into one.

Invalid annotation. We query diverse annotations because of the different annotation definitions of program languages. For example, “#” is used in Python and Ruby, but C, C++, Java, JavaScript, and C# use “//” and “/* */” for comments. As shown in Figure 3d, we make modifications to the annotations in RepoDebug.

B.2 Reference Error

Reference errors in source code can be categorized into several distinct types based on the nature of the token mismatch, each reflecting different underlying causes and exhibiting varying impacts on program behavior.

Error Types	Index	Subclass	Description
Syntax	1	misuse equal sign1	Using = instead of == in comparisons.
	2	misuse equal sign2	Using == instead of = in assignments.
	3	open parenthesis	Missing closing) in code.
	4	open bracket	Missing closing] in lists.
	5	open brace	Missing closing } in dictionaries or blocks.
	6	missing colon	Missing colon: at the end of a statement.
	7	missing comma	Missing comma, between elements.
	8	missing semicolon	Missing semicolon ; at the end of a line.
	9	invalid annotation	Using invalid symbols or no symbols in type annotations.
Reference	10	wrong return statement	Incorrect return statement.
	11	wrong import statement	Incorrect module/class/function name in import.
	12	wrong class call	Incorrect class name when calling a class.
	13	wrong function call	Incorrect function name when calling a function.
	14	wrong parameters	Incorrect or missing function parameters.
Logic	15	divide-by-zero	Division by zero in binary operations.
	16	opposite binary operator	Using wrong binary operator.
	17	missing operand	Missing operand in binary operation.
	18	opposite condition	Using opposite condition in if statement.
	19	constant condition	Using constant value in if condition.
Multiple	20	double bugs	Two separate bugs in the same code segment.
	21	triple bugs	Three separate bugs in the same code segment.
	22	quadruple bugs	Four separate bugs in the same code segment.

Table 6: Illustration of different error subtypes in RepoDebug.

The first type involves **minor lexical variations**, such as incorrect capitalization (e.g., Schedule vs. schedule in Figure 4a) or misuse of singular and plural forms (e.g., row vs. rows in Figure 4b). These errors are typically syntactically valid yet semantically incorrect, often arising from carelessness, inconsistent naming conventions, or case sensitivity issues in programming languages like Python and JavaScript.

The second type involves substitution with **semantically or visually similar identifiers**, such as replacing dataList with dataSet or index with idx. These errors frequently occur in contexts where multiple identifiers share similar naming patterns, and are commonly introduced through cognitive confusion, autocomplete suggestions, or code reuse. As shown in Figure 4c, fetch is related with fetch_schedule, but they have vary different influences. While these substitutions may pass static checks, they often result in subtle logic errors that are difficult to detect and debug.

The third type comprises substitution with **unrelated or dissimilar identifiers**, such as mistakenly using settings in place of register in Figure 4d. These errors usually reflect a deeper misunderstanding of the code context or an incorrect assumption about variable roles, and they can lead to more severe runtime failures or completely unintended behaviors.

Wrong return statement. To introduce bugs in return statements, return statements with values are queried and modified. Another identifier in the block of a function or outside the block of a function replaces the return value.

Wrong import statement. The import statements vary significantly across different programming languages. So we use quite different queries in these languages and modify the import statements for functions, modules, and classes.

Wrong class call and wrong function call. In real-world scenarios, some classes and functions in the same repository may share similar names and they are more easily confused and misused. Thus, RepoDebug prioritizes using other classes or functions to replace the selected calls instead of using identifiers.

Wrong parameters. We apply different modification strategies to parameters of number, identifier, and string. Another random number or string of the same length is used to replace the original number or string. And the identifier is replaced by the same strategy with other reference bugs.

B.3 Logic Error

As illustrated in Figure 5, we present three representative subtypes of logical errors. In contrast to syntax and reference errors, these errors typically do not lead to compilation failures and are largely insensitive to data type mismatches. However, they can severely compromise the intended functionality of the code.

Divide-by-zero. A division operation performed with a divisor of zero can lead to a runtime error or undefined behavior, as division by zero is not a valid mathematical operation. To introduce this bug, we first get the original binary code like "a+b". Then, `a+b` is transformed into "`(a+b)/0`" and put back into the code surrounding it.

Opposite binary operator. We modify the operators in the arithmetic operations within the code, including "+", "-", "*", "/", "+=", "-=", "*=", and "/=". For example, we might change "`a + b`" to "`a - b`" or "`a * b`" to "`a / b`". This can alter the intended logic and produce incorrect results.

Missing operand. We remove one of the operands in a binary operation. For example, "`a + b`" might be changed to "a" or "b". This can cause syntax errors or unexpected behavior, depending on the context.

Opposite condition. For example, using "&&" instead of "||" can lead to incorrect logic. We might change "`if (a || b)`" to "`if (a && b)`", which can significantly alter the flow of the program.

Constant condition We replace a conditional expression with a constant value, such as "`true`" or "`false`". For example, "`if (a > b)`" might be changed to "`if (true)`", causing the condition to always evaluate to true regardless of the actual values of "a" and "b".

C Error Spanning across Files

Our dataset explicitly includes cross-file bugs, such as functional or class-level reference errors that span multiple files. A common example is an incorrect import statement that prevents access to functions or classes defined in other modules.

As shown in Figure 7, the original code contains the correct statement "`import com.microsoft.semantickernel.data.vectorstorage.VectorStoreRecordcollection;`", and it is modified to "`import com.microsoft.semantickernel.data.vectorstorage.VolatileVectorStoreRecordcollectionOptions;`" in the buggy code. While the injected bug is indeed localized to a single line, its semantic impact often extends beyond that line propagating across multiple files. This makes it challenging for models to accurately localize it, which in turn leads to incorrect or ineffective repairs.

D Compilable Bugs with Hidden Errors

In our full dataset, all buggy instances are executed within a Docker environment tailored to each programming language. We collected all the results and observed that only approximately 5% of the injected bugs compile without immediate execution failure. However, even in these cases, the buggy code could lead to downstream issues. One common example is silent logic errors in Figure 8, where the program appears to run normally, but internal states or computations are corrupted, resulting in incorrect behavior in later execution stages. For instance, a buggy code replaces parameter "isCatalogVisible" with parameter "istalogVisible" and they share same data type. The buggy code compiles successfully but introduces logical errors in later execution stages, resulting in clearly misleading outcomes.

E Details for Data Analysis

Table 7 provides a statistical overview of the dataset, while Tables 8 and 9 enumerate all repositories contained in RepoDebug.

F More Details of Evaluation Settings

F.1 Details of Metrics

Bug Identification (BI): For bug identification, the LLM is required to select one error subtype from the provided pool of error types as its answer. We evaluate the model's ability to identify error types using accuracy. This metric measures whether the predicted bug subtype T_i^* for a given buggy code C_i matches the actual bug subtype T_i. If the prediction is correct, M_i^{BI} is set to 1; otherwise, it is set to 0.

$$ACC_{BI}^{i} = \begin{cases} 1 & T_i^*(C_i) = T_i \\ 0 & T_i^*(C_i) \neq T_i. \end{cases} \tag{1}$$

One Bug's Location (OBL): For bug localization, OBL uses accuracy to assess the model's localization capabilities. OBL evaluates whether the model can accurately locate a single error position. Specifically, this metric checks if there is at least one common line between the predicted bug location list L_i^* and the actual bug location list L_i. If there is any overlap, M_i^{OBL} is set to 1; otherwise, it is set to 0.

$$ACC_{OBL}^{i} = \begin{cases} 1 & L_i^*(C_i) \cap L_i \neq \emptyset \\ 0 & L_i^*(C_i) \cap L_i = \emptyset. \end{cases} \tag{2}$$

```
--- pgqueuer/queries.py
+++ pgqueuer/queries.py
@@ -60,7 +60,7 @@
         default_factory=qb.QueryQueueBuilder,
     )
     qbs: qb.QuerySchedulerBuilder = dataclasses.field(
-        default_factory=qb.QuerySchedulerBuilder,
+        default_factory==qb.QuerySchedulerBuilder,
     )

     async def install(self) -> None:
```

(a) An instance of using == instead of = in assignments.

```
--- pgqueuer/queries.py
+++ pgqueuer/queries.py
@@ -418,7 +418,7 @@
     ) -> None:
         await self.driver.execute(
             self.qbs.create_insert_schedule_query(),
-            [k.expression for k in schedules],
+            [k.expression for k in schedules,
             [k.entrypoint for k in schedules],
             list(schedules.values()),
         )
```

(b) An instance of missing closing].

```
--- pgqueuer/queries.py
+++ pgqueuer/queries.py
@@ -177,7 +177,7 @@
         self,
         entrypoint: str,
         payload: bytes | None,
-        priority: int = 0,
+        priority: int = 0
         execute_after: timedelta | None = None,
     ) -> list[models.JobId]: ...
```

(c) An instance of missing comma, between elements.

```
--- pgqueuer/queries.py
+++ pgqueuer/queries.py
@@ -216,7 +216,7 @@
             ValueError: If the lengths of the lists provided do not match when using
multiple jobs.
         """

-        # If they are not lists, create single-item lists for uniform processing
+         If they are not lists, create single-item lists for uniform processing
         normed_entrypoint = entrypoint if isinstance(entrypoint, list) else [entrypoint]
         normed_payload = payload if isinstance(payload, list) else [payload]
         normed_priority = priority if isinstance(priority, list) else [priority]
```

(d) An instance of using invalid symbols or no symbols in type annotations.

Figure 3: Four instances of syntax errors. Red indicates original code; green indicates injected errors.

```diff
--- pgqueuer/queries.py
+++ pgqueuer/queries.py
@@ -451,7 +451,7 @@

     async def peak_schedule(self) -> list[models.Schedule]:
         return [
-            models.Schedule.model_validate(dict(row))
+            models.schedules.model_validate(dict(row))
             for row in await self.driver.fetch(
                 self.qbs.create_peak_schedule_query(),
             )
```

(a) An instance of reference error with incorrect capitalization.

```diff
--- pgqueuer/queries.py
+++ pgqueuer/queries.py
@@ -129,7 +129,7 @@
         )
         assert len(rows) == 1
         (row,) = rows
-        return row["exists"]
+        return rows["exists"]

     async def dequeue(
         self,
```

(b) An instance of reference error with misuse of singular and plural forms.

```diff
--- pgqueuer/queries.py
+++ pgqueuer/queries.py
@@ -356,7 +356,7 @@
         """
         return [
             models.LogStatistics.model_validate(dict(x))
-            for x in await self.driver.fetch(
+            for x in await self.driver.fetch_schedule(
                 self.qbq.create_log_statistics_query(),
                 tail,
                 None if last is None else last.total_seconds(),
```

(c) An instance of reference error with similar identifiers.

```diff
--- pgqueuer/qm.py
+++ pgqueuer/qm.py
@@ -235,7 +235,7 @@
             )
             return func

-        return register
+        return settings

     def observed_requests_per_second(
         self,
```

(d) An instance of reference error with dissimilar identifiers.

Figure 4: Four instances of reference errors. Red indicates original code; green indicates injected errors.

```diff
--- pgqueuer/qm.py
+++ pgqueuer/qm.py
@@ -255,7 +255,7 @@
         samples = self.entrypoint_statistics[entrypoint].samples
         if not samples:
             return 0.0
-        timespan = helpers.utc_now() - min(t for _, t in samples) + epsilon
+        timespan = (helpers.utc_now() - min(t for _, t in samples) + epsilon)/0
         requests = sum(c for c, _ in samples)
         return requests / timespan.total_seconds()
```

(a) An instance of dividing by zero error.

```diff
--- pgqueuer/qm.py
+++ pgqueuer/qm.py
@@ -255,7 +255,7 @@
         samples = self.entrypoint_statistics[entrypoint].samples
         if not samples:
             return 0.0
-        timespan = helpers.utc_now() - min(t for _, t in samples) + epsilon
+        timespan =  epsilon
         requests = sum(c for c, _ in samples)
         return requests / timespan.total_seconds()
```

(b) An instance of missing operand.

```diff
--- src/scanner.c
+++ src/scanner.c
@@ -80,7 +80,7 @@
     // scan for r#' r##' or more #
     skip_whitespace(lexer);

-    if (lexer->lookahead != 'r') {
+    if (lexer->lookahead == 'r') {
         return false;
     }
     adv;
```

(c) An instance of the opposite condition.

Figure 5: Three instances of logic errors. Red indicates original code; green indicates injected errors.

```diff
--- pgqueuer/queries.py
+++ pgqueuer/queries.py
@@ -117,7 +117,7 @@
         (row,) = rows
         return row["exists"]

-    async def has_user_defined_enum(self, key: str, enum: str) -> bool:
+    async def has_user_defined_enum(self, key str, enum: str) -> bool:
         """Check if a value exists in a user-defined ENUM type."""
         rows = await self.driver.fetch(self.qbe.create_user_types_query())
         return (key, enum) in {(row["enumlabel"], row["typname"]) for row in rows}
@@ -187,7 +187,7 @@
         entrypoint: list[str],
         payload: list[bytes | None],
         priority: list[int],
-        execute_after: list[timedelta | None] | None = None,
+        execute_after: list[timedelta | None | None = None,
     ) -> list[models.JobId]: ...

     async def enqueue(
```

(a) An instance of multiple errors.

```diff
--- pgqueuer/helpers.py
+++ pgqueuer/helpers.py
@@ -163,7 +163,7 @@

     # Check for an existing search_path
     if any(opt.startswith("-c search_path=") for opt in options):
-        raise ValueError("search_path is already set in the options parameter.")
+        raise wait_for_notice_event("search_path is already set in the options
parameter.")

     options.append(f"-c search_path={schema}")
     query["options"] = options
@@ -124,7 +124,7 @@
         The jitter will be in the specified range of the base delay.
     """
     jitter = random.uniform(*jitter_span)
-    return timedelta(seconds=timeout.total_seconds() * delay_multiplier * jitter)
+    return timedelta(seconds=(timeout.total_seconds() * delay_multiplier * jitter)/0)


 def normalize_cron_expression(expression: str) -> str:
```

(b) An instance of multiple errors.

Figure 6: Two instances of multiple errors. Red indicates original code; green indicates injected errors.

```
@@ -7,7 +7,7 @@
  import com.azure.core.credential.keyCredential;
  import com.microsoft.semantickernel.aiservices.openai.textembedding.OpanAITextEmbeddingGenerationService;
  import com.microsoft.semantickernel.data.vectorsearch.VectorSearchResults;
- import com.microsoft.semantickernel,data.vectorstorage.VectorStoreRecordcollection;
+ import com.microsoft.semantickernel.data.vectorstorage.VolatileVectorStoreRecordcollectionOptions;
  import com.microsoft.semantickernel.data.VolatileVectorStore;
  import com.microsoft.semantickernel.data.VolatileVectorStoreRecordCollectionOptions;
  import com.microsoft.semantickernel.data.vectorstorage.annotations.VectorStoreRecordData;
```

Figure 7: An instances of error spanning across files. Red indicates original code; green indicates injected errors.

```
@@ -36,7 +36,7 @@
  }

  public Boolean isCatalogVisible() {
-     return isCatalogVisible;
+     return istalogVisible;

  }
```

Figure 8: An instances of compilable error. Red indicates original code; green indicates injected errors.

Language	Train					Test				
	Instances	A.T.	M.T.	A.L.	M.L.	Instances	A.T.	M.T.	A.L.	M.L.
C	251	841	1,512	120	185	159	5,320	5,813	517	606
C#	397	448	864	47	73	100	1,205	1,350	76	81
Go	1,037	1,378	4,604	156	499	466	1,082	2,911	160	411
Java	25,005	869	19,601	106	2,287	2,189	816	2,647	108	419
Javascript	525	2,397	12,312	300	1,563	273	1,342	1,868	188	246
Python	3,500	3,178	22,720	370	2,359	514	3,275	8,685	412	1,021
Ruby	2,759	724	4,267	81	581	562	628	1,053	83	129
Rust	10,144	2,545	28,134	338	4,006	1,175	3,157	15,407	355	1,517

Table 7: Illustration of statistical details of RepoDebug. A.T. refers to the Average Token Length with prompt, M.T. refers to the Maximum Token Length with prompt, A.L. refers to the Average Line Count, and M.L. refers to the Maximum Line Count.

All Bugs' Location (ABL): ABL uses accuracy to assess the model's localization capabilities from different perspectives with OBL and plays a crucial role in the analysis of multiple error localization, as multiple errors involve several error locations. ABL assesses the ability of the models to accurately locate all error positions. Specifically, this metric evaluates whether all actual bug locations L_i are included in the predicted bug location list L_i^*. If the predicted list fully contains the actual list, M_i^{ABL} is set to 1; otherwise, it is set to 0.

$$ACC_{ABL}^{i} = \begin{cases} 1 & L_i \subseteq L_i^*(C_i) \\ 0 & L_i \nsubseteq L_i^*(C_i). \end{cases} \tag{3}$$

Automatic Program Repair (APR): For the task of automatic program repair, rather than evaluating the ability of LLMs to completely repair the errors, we focus on the model's ability to make effective modifications at the correctly identified error positions. We assess it by Edit Similarity (ES) and Exact Match (EM) between the predicted repair r_{ik} and the actual code $C_i[k]$ for each common line k in both the actual and predicted bug location lists. Additionally, it also calculates the Pass@1 of the repair code r_{ik}. The average similarity score indicates how well the repair matches the actual code.

$$ES^{i} = \frac{1}{|L_i|} \sum_{k \in L_i \cap L_i^*} ES(C_i[k], r_{ik}). \tag{4}$$

$$EM^{i} = \frac{1}{|L_i|} \sum_{k \in L_i \cap L_i^*} EM(C_i[k], r_{ik}). \tag{5}$$

F.2 More Information of Baselines

GPT-4.(OpenAI, 2024) This is a large-scale multi-modal model developed by OpenAI. The version GPT-4o-240806 and GPT-4o-mini-240718 belong to the GPT-4o series. In terms of processing English text and code, the performance of GPT-4o is comparable to that of GPT-4 Turbo.

Language	Repository	Creation Time	#Star	Size	#F.	#L.
C	nushell/tree-sitter-nu	2022-09-30	132	52,572	1	151
	jszczerbinsky/lwp	2022-09-18	916	12,898	2	269
	microsoft/xdp-for-windows	2022-04-12	386	5,508	1	58
C#	Sergio0694/PolySharp	2022-10-22	1,901	270	13	588
Go	failsafe-go/failsafe-go	2023-04-12	1,730	640	7	330
	wind-c/comqtt	2022-09-04	1,062	740	3	610
	Wsine/feishu2md	2022-05-16	1,313	213	8	1,311
	sozercan/kubectl-ai	2023-03-19	1,058	249	3	386
Java	zema1/suo5	2022-11-22	2,247	2,965	1	570
	abhi9720/BankingPortal-API	2023-07-22	142	38,494	11	1,690
	woowacourse-teams/2023-hang-log	2023-06-29	228	160,545	12	1,976
	Futsch1/medTimer	2024-01-19	171	80,172	21	3,357
	woowacourse-teams/2022-dallog	2022-06-28	148	4,638	10	1,255
	maplibre/maplibre-react-native	2022-11-03	313	38,767	29	9,994
	marcushellberg/java-ai-playground	2023-10-11	309	766	2	226
	ollama4j/ollama4j	2023-10-26	312	1,190	2	994
JavaScript	RoleModel/turbo-confirm	2023-02-02	152	682	2	113
	sindresorhus/nano-spawn	2024-08-19	473	465	7	321
	LavaMoat/snow	2022-05-30	107	460	1	1,563
Python	Bunsly/JobSpy	2023-07-06	1,198	734	5	601
	janbjorge/pgqueuer	2024-04-19	1,204	1,072	9	2,497
	noamgat/lm-format-enforcer	2023-09-21	1,696	773	4	1,199
	Textualize/trogon	2023-04-18	2,562	479	7	1,500
	pydantic/pydantic-settings	2022-09-07	787	379	2	2,909
	datadreamer-dev/DataDreamer	2023-06-02	963	916	7	2,004
	farizrahman4u/loopgpt	2023-04-14	1,444	571	1	751
	python-humanize/humanize	2022-03-06	558	852	4	1,426
	tetra-framework/tetra	2022-05-01	577	560	12	2,966
	getludic/ludic	2024-03-08	781	938	1	236
Ruby	joeldrapper/quickdraw	2023-02-20	150	238	1	103
	jhawthorn/vernier	2022-04-26	910	529	1	581
	gbaptista/ollama-ai	2024-01-06	210	126	1	160
	oven-sh/homebrew-bun	2022-10-20	120	137	13	671
	alexandreruban/action-markdown	2022-11-10	146	78	7	269
	hopsoft/universalid	2023-03-31	377	160	42	2,035
	excid3/revise_auth	2023-01-12	405	330	8	304
	trilogy-libraries/activerecord-trilogy-adapter	2022-08-10	174	168	2	469
Rust	guywaldman/magic-cli	2024-06-24	737	427	1	304
	resyncgg/dacquiri	2022-01-06	349	183	3	688
	sophiajt/june	2023-05-19	802	582	6	11,341
	woodruffw/zizmor	2024-08-19	1,995	1,143	14	3,677
	hydro-project/rust-sitter	2022-06-26	624	246	7	2,362
	tokio-rs/toasty	2024-10-22	1,271	433	45	14,180
	vincent-herlemont/native_db	2023-05-09	534	861	1	376
	lunatic-solutions/submillisecond	2022-05-04	913	440	14	3,433
	lnx-search/datacake	2022-09-30	397	517	22	8,507

Table 8: Repositories of train set in RepoDebug. #F. represents the number of files with injected errors in a repository and #L. indicates the total line count of files with injected errors in a repository.

Claude 3.5 Sonnet[7]**.** Claude 3.5 Sonnet 241022 is the latest generation of AI models launched by

[7] https://www.anthropic.com/news/introducing-claude

Language	Repository	Creation Time	#Star	Size	#F.	#L.
C	wmww/gtk4-layer-shell	2023-04-06	176	774	2	1,034
C#	amantinband/error-or	2022-05-31	1,703	732	2	154
Go	destel/rill	2024-02-02	1,583	222	1	72
	charmbracelet/log	2022-12-02	2,514	574	7	1,044
Java	projectdiscovery/nuclei-burp-plugin	2022-01-17	1,214	63,116	1	97
	Bindambc/whatsapp-business-java-api	2022-10-13	185	15,826	44	3,663
	microsoft/semantic-kernel-java	2024-06-12	130	4,000	13	1,432
JavaScript	mcollina/borp	2023-11-24	166	690	4	475
	sindresorhus/make-asynchronous	2022-06-26	248	21	1	215
Python	microsoft/picologging	2022-06-10	689	757	4	1,586
	laike9m/Python-Type-Challenges	2023-10-23	572	771	2	79
Ruby	hynek/stamina	2022-09-30	1,048	908	2	923
	Shopify/autotuner	2023-05-25	548	151	8	673
	skryukov/skooma	2023-08-23	153	88	6	291
Rust	tokio-rs/turmoil	2022-08-03	881	302	3	1,261
	automerge/autosurgeon	2022-11-06	305	195	6	1,473
	rust-cross/cargo-zigbuild	2022-02-16	1,760	813	9	2,309

Table 9: Repositories of test set in RepoDebug. #F. represents the number of files with injected errors in a repository and #L. indicates the total line count of files with injected errors in a repository.

Anthropic. It is an important version of the Claude 3.5 series. It performs exceptionally well in software engineering capabilities, agent coding, tool usage, and many other aspects, and is considered an industry-leading generative AI model.

Code Llama.(Rozière et al., 2024) Code Llama is a series of large code models based on Llama2. We deploy Code-llama-7b for evaluation.

DeepSeek R1.(Guo et al., 2025) DeepSeek R1 is a publicly available large language model developed by DeepSeek, designed to enhance reasoning, mathematical, and programming capabilities.

DeepSeek-Coder-V2.(DeepSeek-AI et al., 2024) It is a Mixture-of-Experts (MoE) code large model that continues pre-training on DeepSeek-V2 to enhance its coding and mathematical reasoning capabilities. We evaluate DeepSeek-Coder-V2-16b-lite-instruct in the experiments.

Qwen2.5 Coder.(Hui et al., 2024) This series of models is based on the Qwen2.5 architecture, with a pre-training dataset exceeding 5.5 trillion tokens, and models of 7B, 14B are used.

StarCoder 2.(loz, 2024) StarCoder 2 is pre-trained on The Stack v2 dataset, which covers 619 programming languages and various data types. The 7B and 15B versions of the StarCoder 2 models are evaluated.

We conduct experiments using both proprietary models and the official API of DeepSeek R1. For other open-source models, we utilize their 4-bit K-M quantized versions provided within the Ollama framework.

F.3 Evaluation Prompt

We provide a detailed prompt template for evaluating the model's performance on the test set of RepoDebug in Figure 9. The model input is divided into three components: code, error type description, and instruction. The code component contains the buggy code. The error type description component includes a comprehensive list of error subtypes along with their brief explanations. The instruction component specifies the task for the model, detailing the problem to be resolved and the required format of the output.

G More Experimental Analysis

G.1 Error Response Analysis

The selected task instance comes from the Go project destel/rill, specifically from the file iter.go, as shown in the Figure 10. The model input included the buggy code, a description of the error type, and explicit instructions for the model output. In this case, the code contained two errors: a make misuse on line 66 and a missing brace on line 68.

Buggy code

```
<code>
{buggy_code}
</code>
```

Error Types

```
<error type>
Type 1 : Using = instead of == in comparisons.
Type 2 : Using == instead of = in assignments.
Type 3 : Missing closing parenthesis in code.
Type 4 : Missing closing bracket in lists.
Type 5 : Missing closing brace in dictionaries or blocks.
Type 6 : Missing colon : at the end of a statement.
Type 7 : Missing comma , between elements.
Type 8 : Missing semicolon ; at the end of a line.
Type 9 : Using invalid symbols or no symbols in type annotations.
Type 10 : Incorrect return statement.
Type 11 : Incorrect module/class/function name in import.
Type 12 : Incorrect class name when calling a class.
Type 13 : Incorrect function name when calling a function.
Type 14 : Incorrect or missing function parameters.
Type 15 : Division by zero in binary operations.
Type 16 : Using wrong binary operator.
Type 17 : Missing operand in binary operation.
Type 18 : Using opposite condition in if statement.
Type 19 : Using constant value in if condition.
Type 20 : Two separate bugs in the same code segment.
Type 21 : Three separate bugs in the same code segment.
Type 22 : Four separate bugs in the same code segment.
</error type>
```

Instructions

```
You need to write the following content:
1. Is there an error or errors in the code ?
2. If there is an error or errors, write the index of the error type.
3. If there is an error or errors, write which line the error or errors are.
4. If there is an error or errors, write the right code of the error line.

If there is an error or errors in the code, your output should follow the
format, and multiple <fix> lines are allowed.:
<error>yes</error>
<type>index of error type</type>
<line>Line number or numbers of the error code, split with ','</line>
<fix>
    <line>a line number</line>
    <code>the correct code for this line </code>
</fix>
If there is no error in the code, Your output should follow the format:
<error>no</error>
Notice that do not write anything else.
```

Figure 9: Prompt for code debugging evaluation of large language models.

According to the predefined error taxonomy, this corresponds to a compound error labeled as type 20.

For this instance, the Claude model successfully resolved the issue. Its response not only correctly identifies the error types and their locations, but also generates a precise fix, replacing make with x on line 66 and adding the missing brace on line 68. In contrast, the Qwen model fails to detect the presence of any errors and does not modify the original buggy code, resulting in its inability to pass certain test cases.

We analyze the common errors frequently observed in the responses generated by different models, as shown in Figure 11 and Figure 12. In the model outputs, we observe several consistent patterns: the ability to explicitly localize the error is generally weaker than the ability to generate plausible fixes; moreover, the model-generated responses tend to identify more errors than actually exist, potentially introducing additional, spurious errors.

There exists a discrepancy between the model's explicit ability to identify the error type and location, and its implicit ability to correct the error through code modification. In some cases, models fail to accurately specify the error category or line number in their textual responses, yet still manage to apply correct fixes in the code. This inconsistency suggests that while the underlying language understanding and contextual reasoning capabilities of the model are relatively strong, the error explanation and localization components of the generation process remain limited in precision.

Models also exhibit a tendency toward overcorrection or producing redundant edits, often modifying parts of the code unrelated to the target error. This behavior may stem from that some models treat code as a holistic structure to be globally improved rather than performing minimal, targeted edits.

G.2 Performance on Token Length

Table 10 shows the results for different lengths of code.

G.3 Performance on the Number of Errors

We analyze the models' performance across different error numbers. As shown in Figure 11, the number of errors has a significant impact on the performance of model debugging. When the number of errors increases from 1 to 2, the accuracy of BI and ABL for most models decreases significantly. In contrast, OBL shows a marked increase. When the number of errors is 3 or 4, the trend in model performance remains unchanged, but the rate of change becomes more stable. This indicates that the increasing number of errors leads to higher difficulty in identifying, locating, and completely fixing them. However, it also stimulates the model's potential to locate one error.

Model Input

Buggy Code

```
62 func ToSeq2[A any](in <-chan Try[A]) iter.Seq2[A, error] {
63     return func(yield func(A, error) bool) {
64         defer DrainNB(in)
65         for x := range in {
66             if !yield(x.Value, make.Error) {
67                 return
68
69         }
70     }
71 }
```

Error Types

<error type>
Type 1 : Using = instead of == in comparisons.
...
</error type>

Instructions

You need to write the following content:
1. Is there an error or errors in the code ?
...

Model Output

Successful Output

<error>yes</error>
<type>20</type>
<line>66, 68</line>
<fix><line>66</line><code>\t\t\tif !yield(x.Value, x.Error) {</code>
<line>68</line><code>\t\t\t}</code></fix>

Failed Output

<error>no</error>

Test

Repaired Code && Test Result

```
62 func ToSeq2[A any](in <-chan Try[A]) iter.Seq2[A, error] {
63     return func(yield func(A, error) bool) {
64         defer DrainNB(in)
65         for x := range in {
66             if !yield(x.Value, x.Error) {
67                 return
68             }
69         }
70     }
71 }
```

```
ok   github.com/destel/rill    (cached)
ok   github.com/destel/rill/internal/core    (cached)
ok   github.com/destel/rill/internal/ringbuffer    (cached)
?    github.com/destel/rill/internal/th    [no test files]
?    github.com/destel/rill/mockapi  [no test files]
```

Buggy Code && Test Result

```
62 func ToSeq2[A any](in <-chan Try[A]) iter.Seq2[A, error] {
63     return func(yield func(A, error) bool) {
64         defer DrainNB(in)
65         for x := range in {
66             if !yield(x.Value, make.Error) {
67                 return
68
69         }
70     }
71 }
```

```
FAIL github.com/destel/rill [build failed]
ok   github.com/destel/rill/internal/core    (cached)
ok   github.com/destel/rill/internal/ringbuffer    (cached)
?    github.com/destel/rill/internal/th    [no test files]
?    github.com/destel/rill/mockapi  [no test files]
```

Figure 10: We provide an example of a buggy instance, accompanied by successful and failed model outputs and their associated test results. Red underlines indicate erroneous code, while green underlines denote the corrected code.

Basic Information

```
language: rust
repo: automerge/autosurgeon
file: autosurgeon-derive/src/attrs.rs
error_index: 2
error_lines: [270]
```

Comparison Patch

```
--- autosurgeon-derive/src/attrs.rs
+++ autosurgeon-derive/src/attrs.rs
@@ -267,7 +267,7 @@

 impl EnumNewtypeAttrs {
     pub(crate) fn from_field(field: &syn::Field) -> Result, syn::parse::Error> {
-        let mut result = None;
+        let mut result == None;
         for attr in &field.attrs {
             if attr.path().is_ident("autosurgeon") && result.is_some() {
                 return Err(syn::parse::Error::new(
```

Model: gpt-4o-2024-08-06

```
<error>yes</error>
<type>1</type>
<line>485</line>
<fix><line>485</line><code>let mut result = None;</code></fix>
```

Figure 11: An example where a model successfully analyzes the error type and suggests a fix, but fails to identify the exact location of the error.

📄 Basic Information

```
language: python
repo: laike9m/Python-Type-Challenges
file: challenges/intermediate-generic/solution.py
error_index: 4
error_lines: [27]
```

🛠️ Comparison Patch

```
--- challenges/intermediate-generic/solution.py
+++ challenges/intermediate-generic/solution.py
@@ -24,5 +24,5 @@

 assert_type(add(1, 2), int)
 assert_type(add("1", "2"), str)
-assert_type(add(["1"], ["2"]), List[str])
+assert_type(add(["1", ["2"]), List[str])
 assert_type(add(1, "2"), int)  # expect-type-error
```

🤖 Model: gpt-4o-2024-08-06

```
<error>yes</error>
<type>3,4,9</type>
<line>16,20</line>
<fix><line>16</line><code>assert_type(add(["1", "2"]), List[str])</code></fix>
<fix><line>20</line><code># assert_type(add(1, "2"), int)  # expect-type-error</code></fix>
```

Figure 12: An example where a model incorrectly identifies multiple errors.

Num		GPT		Claude 3.5	DeepSeek	Qwen2.5 Coder		StarCoder2		Deepseek	Code Llama
		4o	4o-mini	Sonnect	R1	14b	7b	15b	7b	Coder 16b	7b
500	ACC_{BI}	36.34	30.16	**51.48**	46.07	25.85	11.09	0.27	0.38	9.51	0.87
	ACC_{OBL}	13.61	5.85	**20.66**	9.84	7.10	3.72	0.11	0.38	14.64	2.73
	ACC_{ABL}	8.14	3.11	**13.06**	6.83	3.88	2.13	0.11	0.27	8.42	1.48
	APR	3.22	1.31	**11.42**	4.75	0.66	0.16	0.00	0.00	0.38	0.00
	ES	9.30	3.72	**18.18**	8.78	4.48	1.48	0.02	0.02	3.32	0.39
	EM	4.31	1.53	**16.53**	6.77	1.01	0.16	0.00	0.00	0.47	0.00
1,000	ACC_{BI}	34.05	26.28	**47.88**	44.00	23.33	8.76	0.22	0.22	9.31	0.67
	ACC_{OBL}	10.37	4.65	**17.68**	8.57	5.68	3.15	0.06	0.26	11.94	1.93
	ACC_{ABL}	6.35	2.57	**11.75**	6.10	3.37	1.77	0.06	0.19	7.12	1.03
	$Pass@1$	2.70	1.03	**9.95**	4.09	0.51	0.10	0.00	0.00	0.26	0.00
	ES	7.19	2.97	**15.67**	7.72	3.51	1.18	0.02	0.01	2.61	0.26
	EM	3.51	1.16	**13.86**	5.64	0.75	0.10	0.00	0.00	0.31	0.00
2,000	ACC_{BI}	33.63	25.05	**46.39**	45.27	21.26	8.26	0.34	0.22	8.36	0.63
	ACC_{OBL}	8.75	3.91	**15.92**	8.29	4.71	2.67	0.07	0.19	10.98	1.68
	ACC_{ABL}	5.54	2.21	**11.01**	5.95	2.87	1.56	0.05	0.15	6.63	0.90
	$Pass@1$	2.31	0.87	**9.14**	4.02	0.41	0.07	0.00	0.00	0.19	0.00
	ES	6.14	2.51	**14.23**	7.50	2.90	0.98	0.02	0.01	2.31	0.28
	EM	2.92	1.02	**12.44**	5.43	0.60	0.07	0.00	0.00	0.23	0.00
5,000	ACC_{BI}	31.00	23.11	43.76	**45.50**	18.64	7.37	0.33	0.22	7.84	0.57
	ACC_{OBL}	7.53	3.27	**14.01**	7.19	3.86	2.27	0.06	0.16	10.13	1.48
	ACC_{ABL}	4.81	1.85	**9.79**	5.26	2.36	1.32	0.04	0.12	6.34	0.85
	$Pass@1$	1.97	0.71	**7.96**	3.60	0.33	0.06	0.00	0.00	0.16	0.00
	ES	5.25	2.12	**12.48**	6.50	2.38	0.83	0.02	0.01	1.98	0.24
	EM	2.51	0.85	**10.76**	4.76	0.48	0.06	0.00	0.00	0.19	0.00
10,000	ACC_{BI}	30.51	22.77	43.07	**45.50**	18.14	7.12	0.36	0.22	7.70	0.56
	ACC_{OBL}	7.20	3.17	**13.47**	6.93	3.71	2.17	0.06	0.15	9.82	1.41
	ACC_{ABL}	4.61	1.80	**9.43**	5.06	2.27	1.26	0.04	0.11	6.18	0.81
	$Pass@1$	1.89	0.67	**7.68**	3.46	0.32	0.06	0.00	0.00	0.15	0.00
	ES	5.03	2.04	**11.99**	6.27	2.29	0.79	0.02	0.01	1.89	0.23
	EM	2.41	0.81	**10.34**	4.59	0.46	0.06	0.00	0.00	0.18	0.00

Table 10: Results for different lengths of code. **Bold** indicates the best, underline indicates the second best.

Num		GPT		Claude 3.5	DeepSeek	Qwen2.5 Coder		StarCoder2		Deepseek	Code Llama
		4o	4o-mini	Sonnect	R1	14b	7b	15b	7b	Coder 16b	7b
1	ACC_{BI}	35.55	26.63	49.79	**53.65**	21.04	7.65	0.42	0.22	8.70	0.64
	ACC_{OBL}	5.37	2.06	**11.00**	5.70	2.56	1.45	0.04	0.13	6.95	0.92
	ACC_{ABL}	5.37	2.06	**11.00**	5.70	2.56	1.45	0.04	0.13	6.95	0.92
	$Pass@1$	2.24	0.79	**8.99**	4.04	0.37	0.07	0.00	0.00	0.18	0.00
	ES	4.34	1.72	**10.26**	5.34	1.70	0.57	0.01	0.01	1.35	0.16
	EM	2.24	0.79	**8.99**	4.04	0.37	0.07	0.00	0.00	0.18	0.00
2	ACC_{BI}	3.17	0.90	**10.86**	0.45	2.26	9.95	0.00	0.00	5.88	0.00
	ACC_{OBL}	10.86	6.33	18.55	10.86	7.69	5.43	0.00	0.00	**19.91**	4.07
	ACC_{ABL}	0.90	0.90	0.45	**2.71**	1.36	0.45	0.00	0.00	2.26	0.45
	$Pass@1$	0.27	0.19	0.19	**1.23**	0.42	0.09	0.00	0.00	0.42	0.11
	ES	6.60	3.51	**16.52**	9.88	5.10	2.45	0.00	0.00	3.82	0.75
	EM	2.71	1.36	**14.93**	7.69	1.36	0.00	0.00	0.00	0.45	0.00
3	ACC_{BI}	2.65	0.00	2.32	2.65	0.99	**2.98**	0.00	0.33	0.33	0.33
	ACC_{OBL}	17.55	8.28	**24.83**	12.91	9.60	4.64	0.33	0.33	23.84	2.98
	ACC_{ABL}	0.33	0.00	0.00	0.99	0.00	0.00	0.00	0.00	**1.32**	0.00
	$Pass@1$	0.00	0.00	0.00	**0.72**	0.00	0.00	0.00	0.00	0.27	0.00
	ES	10.29	3.42	**21.49**	10.13	5.41	1.26	0.00	0.00	5.11	0.47
	EM	4.30	0.50	**18.21**	6.13	0.99	0.00	0.00	0.00	0.22	0.00
4	ACC_{BI}	**0.85**	0.00	0.28	0.28	0.00	**0.85**	0.00	0.28	0.00	0.00
	ACC_{OBL}	18.18	10.23	**28.69**	13.35	9.94	6.82	0.00	0.28	25.85	4.26
	ACC_{ABL}	0.00	0.00	0.00	0.28	0.28	0.00	0.00	0.00	**1.14**	0.00
	$Pass@1$	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**0.10**	0.00
	ES	7.39	3.59	**20.04**	10.92	4.96	1.91	0.11	0.00	4.35	0.46
	EM	2.37	0.71	**15.34**	7.10	0.43	0.00	0.00	0.00	0.00	0.00

Table 11: Results for different numbers of errors. **Bold** indicates the best, underline indicates the second best.