# COAST: Enhancing the Code Debugging Ability of LLMs through Communicative Agent Based Data Synthesis

**Weiqing Yang[1], Hanbin Wang[2], Zhenghao Liu[1]*, Xinze Li[1], Yukun Yan[3], Shuo Wang[3], Yu Gu[1], Minghe Yu[4], Zhiyuan Liu[3] and Ge Yu[1]**

[1]Department of Computer Science and Technology, Northeastern University, China
[2]School of Software and Microelectronics, Peking University, China
[3]Department of Computer Science and Technology, Institute for AI, Tsinghua University, China
[4]Software College, Northeastern University, China

## Abstract

Code debugging is a vital stage of software development, essential for ensuring the reliability and performance of Large Language Models (LLMs) in the code generation task. Human debugging typically follows a multi-stage process, which includes Bug Localization, Bug Identification, Code Repair, and Code Recognition. However, existing code debugging benchmarks predominantly focus on the Code Repair stage, which offers only a limited perspective on evaluating the debugging capabilities of LLMs. In this paper, we introduce DEBUGEVAL, a comprehensive benchmark for evaluating the debugging abilities of LLMs by emulating the multi-stage human debugging process. Through evaluating on DEBUGEVAL, we observe that 7B-scale models consistently underperform compared to their larger counterparts, highlighting their limitations in comprehending code semantics. In this case, we propose the **CO**mmunicative **A**gent-based data **S**yn**T**hesis (COAST) framework, which employs a multi-agent system to generate high-quality training data for supervised fine-tuning (SFT). Experimental results demonstrate that COAST-generated data outperform human-curated and GPT-4-generated data, enabling 7B-scale LLMs to achieve debugging performance comparable to GPT-3.5. All data and codes are available at `https://github.com/NEUIR/COAST`.

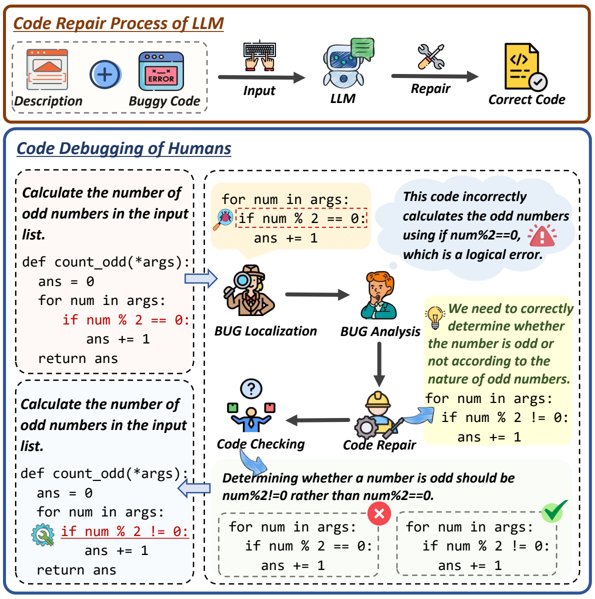



Figure 1: Illustration of the Human Code Debugging Process. Existing studies (Olausson et al., 2023; Chen et al., 2023) typically focus on directly repairing code generated by Large Language Models (LLMs). In contrast to this approach, humans often engage in a multi-stage process to resolve buggy codes.

## 1 Introduction

In software development, code debugging is a crucial stage for ensuring the functionality and reliability of applications (Hailpern and Santhanam, 2002; Kirschner et al., 2020). Traditional code debugging methods often rely on heuristics (Le Goues et al., 2012; Wen et al., 2018) and predefined patterns (Hua et al., 2018; Liu et al., 2019). However, these methods are reaching their limits, as software systems become increasingly complex.

Large Language Models (LLMs) (OpenAI, 2022; Touvron et al., 2023) have opened new avenues for automated code debugging, enabling more flexible and comprehensive methods to identify and resolve code errors (Chen et al., 2023). As illustrated in Figure 1, the debugging process of humans typically involves multiple essential stages: locating buggy code segments, analyzing root causes of bugs, and repairing identified issues (Chen et al., 2024). In addition, a comparison is made between the code before and after repair to check whether the issue is actually solved. Each of these stages is critical for successfully fixing bugs. However, existing debugging benchmarks (Tian et al., 2024; Khan et al., 2023; Huq et al., 2022) primarily focus on assessing LLMs' capability to fix bugs, overlooking their performance in the distinct stages in

*indicates corresponding author.



*Findings of the Association for Computational Linguistics: NAACL 2025*, pages 2570–2585
April 29 - May 4, 2025 ©2025 Association for Computational Linguistics

the human debugging process.

To facilitate more comprehensive evaluations, this paper introduces DEBUGEVAL, a benchmark specifically designed to assess the code debugging capabilities of LLMs. DEBUGEVAL introduces four tasks–BUG Localization, BUG Identification, Code Repair, and Code Recognition–that closely mirror the debugging process of humans in the real world. Each task spans multiple programming languages, including Python, C++, and Java. To better simulate real-world software development, the buggy codes in DEBUGEVAL are from human and GPT-4. We evaluate the debugging performance of various LLMs on DEBUGEVAL and observe that LLMs with 7 billion parameters exhibit significantly weaker debugging capabilities compared to LLMs with 70 billion parameters or more. Thus, improving the debugging proficiency of LLMs themselves remains a critical challenge for building more comprehensive code intelligence.

While Supervised Fine-Tuning (SFT) has been widely adopted to enhance LLMs' performance for specialized tasks using human-labeled or LLM-generated data (Roziere et al., 2023; Yue et al., 2023; Zhang et al., 2023), its effectiveness is often limited by the quality of the labeled data. To improve the quality of synthesized data, we propose the **CO**mmunicative **A**gent-based data **S**yn**T**hesis (COAST) framework. COAST builds three agents for collaboration: `Code Quizzer`, `Code Learner`, and `Code Teacher`. These agents work together to generate code debugging data for finetuning the `Code Learner`. Specifically, the `Code Quizzer` first creates a diverse range of code debugging problems. The `Code Learner` then attempts to answer these questions, serving as a critic to assess their educational value. Problems that the `Code Learner` answers incorrectly are flagged and curated as SFT data. Finally, the `Code Teacher` enriches these problems by providing detailed explanations and guidance. The synthesized data are collected and used to finetune the `Code Learner`, enabling the development of our NeuDebugger model.

Our experiments on DEBUGEVAL demonstrate that the COAST framework significantly enhances the debugging capabilities of 7B-scale LLMs. Notably, further analysis reveals that collecting SFT data from human trials and LLM solely does not improve the debugging performance of LLMs (Gudibande et al., 2024). In the COAST framework, the `Code Teacher` effectively guides the `Code Learner` through three tasks–BUG Localization, BUG Identification, and Code Recognition–by employing Chain-of-Thought (CoT) reasoning (Wei et al., 2022b). However, CoT reasoning negatively affects performance in the Code Repair task, which introduces noise and disrupts the underlying code structure. Additionally, the synthesized data significantly boost the performance of code-focused LLMs, such as DeepSeek-Coder-6.7B-Ins, more effectively than general-purpose LLMs like Llama3-8B-Ins. These findings provide valuable insights for future research that aims to improve LLMs' debugging capabilities.

## 2 Related Work

Debugging is a critical stage to ensure the quality of code generated by Large Language Models (LLMs) (Olausson et al., 2023; Chen et al., 2023). Early approaches primarily relied on feature-based methods, such as templates (Hua et al., 2018; Liu et al., 2019), heuristic rules (Le Goues et al., 2012; Wen et al., 2018), or constraints (Mechtaev et al., 2016; DeMarco et al., 2014), to repair buggy code. However, these methods often fall short in addressing a wide range of bug types or tackling more complex programming challenges.

With the advancement of the pretraining technique, researchers have begun to explore the application of Pretrained Language Models (PLMs) to automated code debugging. For example, Xia et al. (2022) utilize the code-oriented pretrained model CodeX (Chen et al., 2021a) to investigate the debugging potential of PLMs. Their findings reveal that CodeX excels at repairing code, particularly for Python and Java. Similarly, Kolak et al. (2022) employ GPT-2 (Radford et al., 2019) and CodeX to evaluate their ability to generate accurate patches for buggy code lines based on given prefixes.

Thrived on the emergent ability of Large Language Models (LLMs) (Wei et al., 2022a), recent research has increasingly focused on leveraging LLMs for automated debugging. Self-Debug (Chen et al., 2023) guides LLMs to generate code reviews, enabling them to refine their own buggy outputs. Self-Repair (Olausson et al., 2023) incorporates human feedback to address errors in generated code. Moreover, Self-Edit (Zhang et al., 2023) introduces a fault-aware editor that utilizes both error messages from test case evaluations and corresponding code snippets to repair bugs. Wang et al. (2024) further explore the interactive Chain-of-Repair (CoR), which asks LLMs to iteratively refine code based

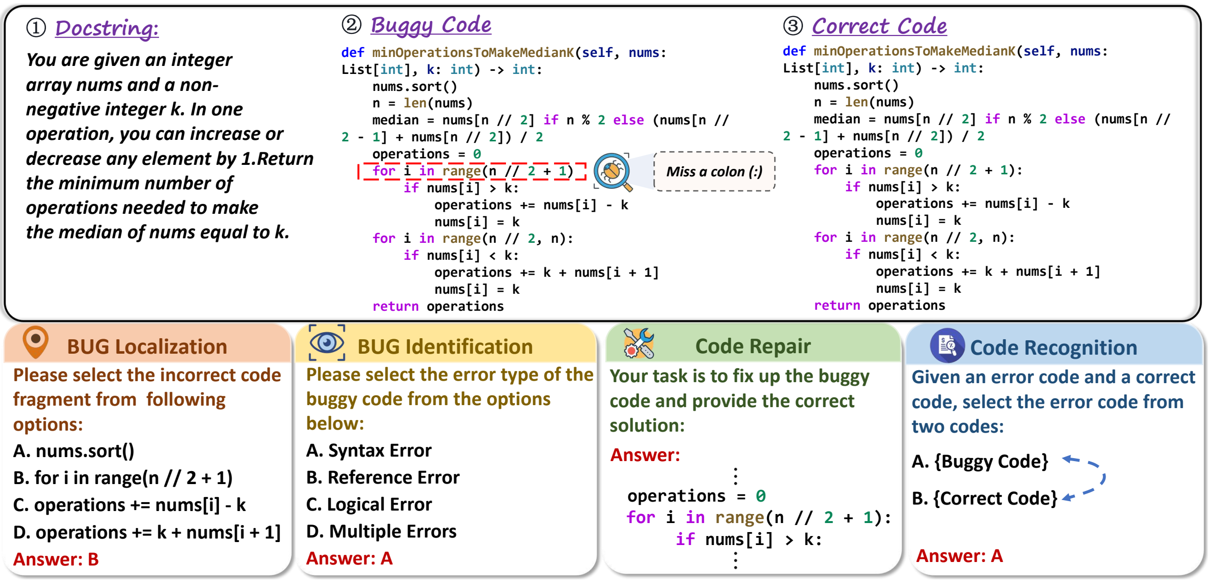



Figure 2: Illustration of DEBUGEVAL Benchmark. The DEBUGEVAL includes four key tasks: BUG Localization, BUG Identification, Code Repair, and Code Recognition.

on compiler error messages and generated repair guidelines. Despite these advances, the success of these methods heavily depends on the inherent debugging proficiency of LLMs.

To enhance the inherent debugging capabilities of LLMs, recent studies have emphasized generating data for supervised fine-tuning. InstructCoder (Li et al., 2023) adopts the Self-Instruct methodology (Wang et al., 2022) to construct an instruction-tuning dataset, which is used to improve LLMs' performance in debugging tasks. Similarly, Li et al. (2024) build the APR-INSTRUCTION dataset and finetune LLMs using four Parameter-Efficient Fine-Tuning (PEFT) techniques, including LoRA (Hu et al., 2022a), p-tuning (Li and Liang, 2021), prefix-tuning (Liu et al., 2021), and $(IA)^3$ (Liu et al., 2022).

To evaluate the debugging capabilities of LLMs, several studies have developed comprehensive benchmarks. For example, Wang et al. (2024) curate buggy code submissions from the Atcoder platform to create CodeError, a benchmark for evaluating the ability of LLMs to fix Python code. Similarly, Tian et al. (2024) introduce DebugBench, which evaluates the debugging performance of LLMs in Python, C++, and Java by synthesizing buggy code using GPT-4 (Achiam et al., 2023). Lastly, Guo et al. (2024b) also propose CodeEditorBench, a benchmark designed to evaluate the debugging capabilities of buggy code of varying difficulty levels across Python, Java, and C++. However, these benchmarks focus primarily on the Code Repair task and fail to provide a more holistic evaluation of LLMs' debugging abilities.

## 3 DEBUGEVAL: Benchmarking the Debugging Capabilities of LLMs

In this section, we first describe task definitions of designed tasks in DEBUGEVAL (Sec. 3.1). Then we detail the process of constructing the DEBUGEVAL benchmark (Sec. 3.2). Finally, we show the characteristics of DEBUGEVAL by comparing with other different debugging benchmarks (Sec. 3.3).

### 3.1 Task Definition

As shown in Figure 2, DEBUGEVAL introduces four distinct tasks to evaluate the debugging capabilities of LLMs. The tasks–BUG Localization, BUG Identification, and Code Recognition–are all single-choice problems, while the Code Repair task focuses on correcting buggy code.

**BUG Localization.** The BUG Localization task aims to identify the specific line of code that contains the bug. In this task, we prompt LLMs to select the line containing the bug from four given choices. Instead of asking LLMs to directly generate the buggy code line, we frame the Bug Localization task as a single-choice problem to ensure a more accurate evaluation.

**BUG Identification.** After pinpointing the code line that contains the bug, the software developers typically analyze the type of bug to facilitate more effective correction. For the BUG Identification

Table 1: A Comparison between DEBUGEVAL and Other Code Debugging Benchmarks.

| Benchmark | Number of Languages | Task | Testing Set Scale | Error Types | Source of Bugs |
|---|---|---|---|---|---|
| DeepFix (2021) | 1 | Code Repair | 6,971 | 4 | User |
| Review4Repair (2022) | 1 | Code Repair | 2,961 | - | User |
| Bug2Fix (2021) | 1 | Code Repair | 5,835 | - | User |
| Github-Python (2021) | 1 | Code Repair | 15,000 | 14 | User |
| FixEval (2023) | 2 | Code Repair | 286,000 | - | User |
| CodeError (2024) | 1 | Code Repair | 4,463 | 6 | User |
| xCodeEval (2023) | 11 | Code Repair | 17,699 | 6 | User |
| DebugBench (2024) | 3 | Code Repair | 4,253 | 18 | GPT4 |
| CodeEditorBench (2024b) | 3 | Code Repair | 1,907 | 14 | GPT4 |
| DEBUGEVAL | 3 | BUG Localization<br>BUG Identification<br>Code Repair<br>Code Recognition | 5,712 | 18 | User&GPT4 |

task, LLMs are responsible for classifying the error type based on the provided buggy code. The possible error types include Syntax Error, Reference Error, Logical Error, and Multiple Errors.

**Code Repair.** The Code Repair (Wang et al., 2024; Tian et al., 2024) task requires LLMs to generate a corrected version of the given buggy code. The performance of LLMs in code repair is evaluated according to the correctness of the repaired code, and the correctness of the repaired code is evaluated by running a set of predefined test cases.

**Code Recognition.** The Code Recognition task provides two code segments and asks LLMs to identify which one contains the bug. Both correct and buggy codes differ in only a few lines.

### 3.2 Details of Data Construction

To ensure the quality of DEBUGEVAL, we collect data from existing benchmarks (DebugBench (Tian et al., 2024) and LiveCodeBench (Jain et al., 2024)) and human trials (AtCoder website[1]).

**Bug Localization.** In the Bug Localization task, we sample test instances from DebugBench. Then we construct the dataset by selecting no more than 20 instances for each of the 15 distinct single code error types across various programming languages. For each instance, we compare the buggy code with the corresponding correct code, identifying the line containing the bug as the golden reference. Additionally, we randomly select three other lines from the buggy code as distractors. Instances where fixing the bug requires inserting or deleting lines are excluded to streamline the construction of evaluation options for the LLMs. Each data entry is manually reviewed to eliminate weak or duplicate candidates, ensuring high quality and relevance.

**Bug Identification.** For the Bug Identification task, we sample test instances from DebugBench. The choices of this task consist of four error types: Syntax Error, Reference Error, Logical Error, and Multiple Errors. We compare the sizes of the test sample sets for each error type and select the smallest set size as the sampling number. The test instances are then sampled from the various code error sets based on this number.

**Code Repair.** For the Code Repair task, we first collect 138 programming contest problems newly published on AtCoder website between September 1, 2023, and April 1, 2024. We then gather buggy code submissions from users, specifically selecting examples in Python, C++, and Java. Additionally, we collect test cases for each problem to validate the correctness of the LLM-modified code.

**Code Recognition.** We build a mixed dataset by collecting test instances from both DebugBench and LiveCodeBench. For each programming language, we randomly select 800 test instances from this dataset. Each test instance consists of both a buggy code snippet and a correct version.

### 3.3 Comparison of Different Debugging Benchmarks

Finally, we present a comprehensive comparison between DEBUGEVAL and other debugging benchmarks in Table 1, highlighting the characteristics and strengths of DEBUGEVAL.

Unlike existing debugging benchmarks, DEBUGEVAL offers a more comprehensive evaluation of debugging capabilities of LLMs by designing testing scenarios related to human-driven code debugging. To better simulate real-world software development, DEBUGEVAL includes buggy code samples generated both by humans and GPT-4. Furthermore, it spans a wide range of error types and different programming languages, more in line with the complexity of the real-world code debugging task. All these aspects ensure that DEBUGEVAL

[1] https://atcoder.jp

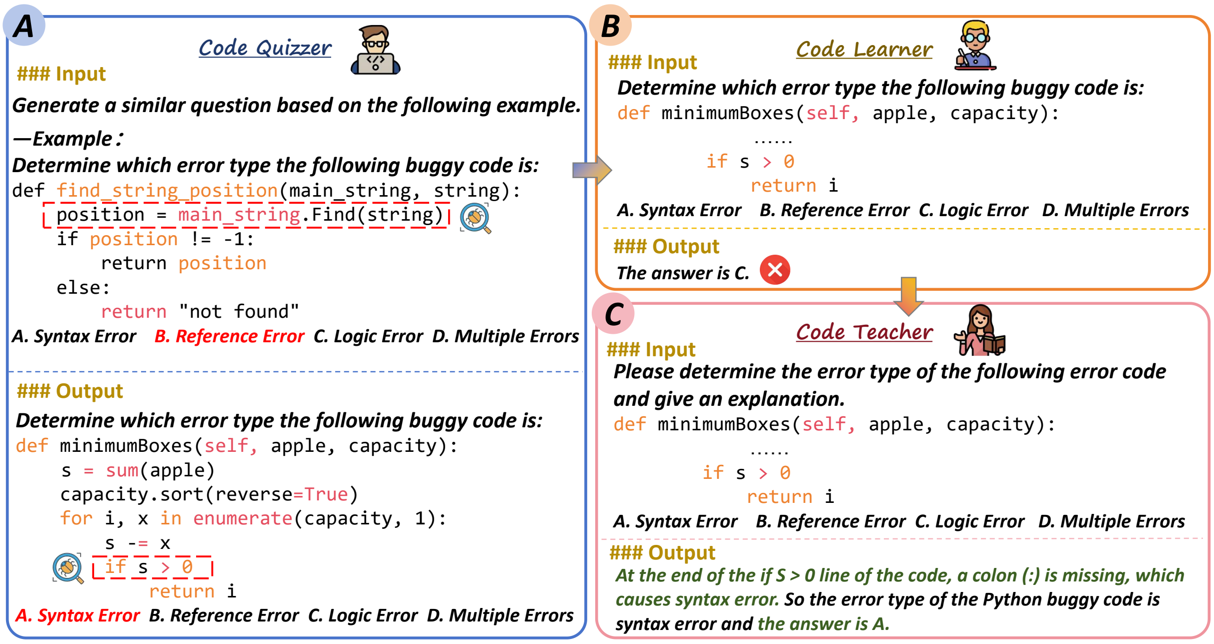



Figure 3: Illustration of **CO**mmunicative **A**gent Based Data **S**yn**T**hesis (COAST) Framework.

can provide more reliable evaluation results.

## 4 COAST: Communicative Agent Based Data Synthesis Framework

This section introduces the **CO**mmunicative **A**gent based data **S**yn**T**hesis (COAST) framework. As shown in Figure 3, COAST automatically synthesizes high-quality debugging data through multi-agent interactions. We first configure the agents to play different roles in the COAST framework (Sec. 4.1). Then COAST employs different agents to collaboratively synthesize high-quality training data for Supervised Fine-Tuning. (Sec. 4.2).

### 4.1 Agent Building

COAST employs three agents that collaboratively generate, resolve, and explain code debugging problems to synthesize high-quality SFT data. More details on the design of the prompts for different agents are provided in the Appendix A.2.

**Code Quizzer.** The `Code Quizzer` is designed to generate a set of code debugging problems. A stronger LLM is utilized as the `Code Quizzer`, which is configured using the instruction: "You are a code debugging expert, skilled in generating code debugging problems to challenge programmers". The `Code Quizzer` leverages examples from the DEBUGEVAL benchmark that span diverse debugging tasks and programming languages. These examples are then used to guide the `Code Quizzer` in generating corresponding task problems in various programming languages, which are subsequently solved by the `Code Learner`.

**Code Learner.** COAST framework sets `Code Learner` (Lee et al., 2024) to improve the quality of training data. The `Code Learner` uses the same backbone model as the SFT model and serves as a critic to evaluate the educational value of the problems generated by `Code Quizzer`. The `Code Learner` is instructed using the prompt: "You are a code debugger". Thus, `Code Learner` attempts to solve problems based on memorized knowledge and judge the educational value of each problem according to whether `Code Learner` successfully solves this problem. These educational instances can help improve the performance of `Code Learner` during Supervised Fine-Tuning.

**Code Teacher.** Inspired by Wang et al. (2024), we also develop a `Code Teacher` by prompting the same LLM used for the `Code Quizzer` with the instruction: "You are an experienced and insightful code debugger". This prompt directs the LLM to act as a proficient code debugger, generating detailed explanations in the form of Chain-of-Thought (CoT) (Wei et al., 2022c). These explanations provide valuable guidance to the `Code Learner` during the SFT process.

### 4.2 Synthesizing SFT Data through Multi-Agent Interactions

COAST automatically synthesizes high-quality data through the interaction among `Code Quizzer`,

Table 2: Statistics of Data Used in Different Supervised Fine-Tuning Strategies.

| SFT Data | Data Source | #Instance |
|---|---|---|
| Human/GPT-4 | UltraInteract (2024) | 154,347 |
| | InstructCoder (2023) | 6,913 |
| | RepairLlama (2023) | 64,643 |
| COAST | BUG Localization data | 4,681 |
| | BUG Identification data | 4,474 |
| | Code Repair data | 11,317 |
| | Code Recognition data | 4,420 |

`Code Learner`, and `Coder Teacher`. The data synthesis process is detailed as follows.

In **Step A**, to ensure the diversity of tasks and error types during data synthesis, we instruct the `Code Quizzer` to generate debugging problems using demonstrations of different tasks and error types. Then, in **Step B**, the `Code Learner` acts as a critic, assessing the educational value of each synthesized problem. If the `Code Learner` solves the problem correctly, it indicates that the `Code Learner` already possesses the necessary knowledge to solve the problem and thus the problem can be discarded. On the other hand, if the `Code Learner` produces an incorrect solution, the problem is reserved as SFT data, due to its educational value for guiding the `Code Learner`. Finally, in **Step C**, the `Code Teacher` reviews the reserved problems and generates a detailed explanation for each problem. These explanations, in the form of Chain-of-Thought (CoT), include the analysis of errors and correct answers, which are essential for the `Code Learner` to understand the problems and refine their solutions. The responses generated by the `Code Teacher` are treated as the final outputs of the synthesized problems, forming the SFT data used to finetune `Code Learner`.

## 5 Experimental Methodology

In this section, we describe the SFT dataset, evaluation metrics, evaluation models, and implementation details of our experiments.

**SFT Dataset.** For Vanilla SFT, we collect training data from UltraInteract (Yuan et al., 2024), InstructCoder (Hu et al., 2023), and RepairLlama (Silva et al., 2023) to finetune LLMs. These datasets are generated by GPT-4 or manually annotated by humans. For COAST, we construct training data for different tasks defined by DEBUGEVAL and the training data are mixed for training models. The data statistics are presented in Table 2.

**Evaluation Metrics.** For the BUG Localization, BUG Identification, and Code Recognition tasks, LLMs are required to select an answer from multiple choices. Following prior work (Suzgun et al., 2022), we use Accuracy as the evaluation metric for these tasks. In particular, for the Code Recognition task, we swap the order of the two options, considering the problem is correctly solved only if LLMs answer correctly in both orders. For the Code Repair task, we adopt Pass@1 (Chen et al., 2021b) to assess the effectiveness of various LLMs.

**Evaluation Models.** We evaluate 13 LLMs on DEBUGEVAL, including both closed-source and open-source LLMs. Closed-source LLMs comprise the GPT series (GPT-4o-mini and GPT-3.5-Turbo). Open-source LLMs include the DeepSeek series (DeepSeek-Coder-V2, DeepSeek-V2, Deepseek-Coder-33B-Ins, DeepSeek-Coder-6.7B-Ins, and DeepSeek-LLM-7B-Ins), the Llama series (Llama3-70B-Ins, Llama3-8B-Ins, Llama2-7B-Ins, and CodeLlama-7B-Ins), and the Qwen series (Qwen2-72B-Ins and CodeQwen1.5-7B-Ins). More detailed descriptions of the evaluation models are shown in Appendix A.3.

**Implementation Details.** For closed-source LLMs, we utilize the APIs provided by their respective vendors. For open-source LLMs, we employ the vLLM framework (Kwon et al., 2023) for inference. During inference, we set the temperature to 0.2 and limit the maximum generation length to 1024 tokens. Our `Code Quizzer` and `Code Teacher` models are based on DeepSeek-Coder-V2, while DeepSeek-Coder-6.7B-Ins and Llama3-8B-Ins serve as our `Code Learner` models. All LLMs are trained using the Llama-Factory framework (hiyouga, 2023) using LoRA (Hu et al., 2022b) for efficient fine-tuning. For SFT, we configure the learning rate to 2e-5, set the number of training epochs to 1, use a batch size of 8, and employ 4 gradient accumulation steps.

## 6 Evaluation Results

In this section, we first show the performance of various LLMs on DEBUGEVAL and also conduct ablation studies to investigate the effectiveness of NeuDebugger models. Then the effectiveness of NeuDebugger models is evaluated in addressing different types of code errors. Finally, some case studies are provided in the Appendix A.7.

Table 3: Evaluation Results for Different LLMs on DEBUGEVAL, with DS representing the DeepSeek Models.

| Model | BUG Localization | | | | BUG Identification | | | | Code Repair | | | | Code Recognition | | | | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | PY | C++ | JAVA | Avg. | PY | C++ | JAVA | Avg. | PY | C++ | JAVA | Avg. | PY | C++ | JAVA | Avg. | |
| GPT-4o-mini (2024) | 84.8 | 81.0 | 81.5 | 82.4 | 53.3 | 48.5 | 48.9 | 50.2 | 65.2 | 67.2 | 67.4 | 66.6 | 85.4 | 90.9 | 91.0 | 89.1 | 72.1 |
| GPT-3.5-Turbo (2022) | 40.4 | 47.2 | 52.2 | 46.9 | 35.5 | 33.3 | 34.1 | 34.3 | 57.2 | 52.9 | 61.6 | 57.2 | 79.4 | 82.4 | 84.0 | 81.9 | 55.1 |
| DeepSeek-V2 (2024b) | 82.0 | 81.0 | 85.9 | 83.0 | 62.0 | 61.0 | 61.3 | 61.4 | 65.2 | 63.0 | 63.5 | 63.9 | 77.4 | 83.9 | 80.5 | 80.6 | 72.2 |
| DeepSeek-Coder-V2 (2024) | 88.8 | 83.1 | 89.8 | 87.2 | 58.7 | 58.9 | 60.8 | 59.4 | 66.7 | 63.1 | 62.3 | 64.0 | 87.9 | 94.9 | 93.3 | 92.0 | 75.7 |
| Llama3-70B-Ins (2024) | 74.2 | 75.9 | 82.0 | 77.5 | 42.8 | 42.3 | 44.9 | 43.3 | 44.9 | 44.2 | 45.7 | 44.9 | 73.9 | 61.6 | 63.3 | 66.3 | 58.0 |
| Qwen2-72B-Ins (2024) | 79.8 | 69.2 | 74.6 | 74.4 | 45.8 | 45.0 | 41.3 | 44.1 | 43.5 | 42.0 | 42.8 | 42.8 | 61.5 | 75.8 | 70.4 | 69.2 | 57.6 |
| DSCoder-33B-Ins (2024a) | 52.2 | 50.3 | 51.7 | 51.4 | 24.9 | 26.0 | 30.9 | 27.2 | 46.4 | 50.7 | 54.3 | 50.5 | 24.8 | 27.0 | 30.5 | 27.4 | 39.1 |
| Llama2-7B-Ins (2023) | 18.0 | 20.0 | 22.4 | 20.2 | 24.9 | 27.0 | 25.8 | 25.9 | 4.3 | 11.7 | 19.6 | 11.9 | 2.3 | 0.6 | 2.0 | 1.6 | 14.9 |
| CodeLlama-7B-Ins (2023) | 27.0 | 20.0 | 23.9 | 23.5 | 26.1 | 23.0 | 23.8 | 24.3 | 18.8 | 23.2 | 23.2 | 21.7 | 48.1 | 60.5 | 65.6 | 58.1 | 31.9 |
| CodeQwen1.5-7B-Ins (2023) | 29.2 | 30.8 | 38.0 | 32.9 | 27.6 | 25.9 | 28.8 | 27.4 | 39.1 | 49.3 | 52.9 | 47.1 | 26.9 | 34.4 | 37.1 | 32.8 | 35.1 |
| DeepSeek-LLM-7B-Ins (2024a) | 27.0 | 19.0 | 25.9 | 23.9 | 30.5 | 28.5 | 30.9 | 29.9 | 21.0 | 24.1 | 14.5 | 19.9 | 35.9 | 36.6 | 46.0 | 39.5 | 28.3 |
| DSCoder-6.7B-Ins (2024a) | 22.5 | 25.6 | 33.7 | 27.5 | 26.6 | 26.0 | 25.9 | 26.2 | 31.9 | 43.5 | 46.4 | 40.6 | 15.5 | 17.8 | 27.8 | 20.3 | 28.7 |
| NeuDebugger-DS-6.7B | 62.4 | 55.4 | 59.0 | 58.8 | 42.6 | 46.9 | 47.8 | 45.8 | 43.5 | 48.6 | 56.5 | 49.5 | 71.0 | 71.4 | 71.4 | 71.3 | 56.4 |
| Llama3-8B-Ins (2024) | 55.6 | 55.9 | 61.0 | 57.6 | 36.8 | 38.1 | 34.6 | 36.6 | 26.1 | 34.3 | 28.3 | 29.6 | 69.1 | 77.4 | 78.1 | 74.9 | 49.7 |
| NeuDebugger-Llama3-8B | 64.6 | 57.9 | 61.0 | 61.1 | 38.6 | 29.9 | 33.3 | 33.8 | 38.4 | 41.3 | 45.7 | 41.8 | 75.3 | 78.0 | 82.4 | 78.5 | 53.8 |

Table 4: Effectiveness of Different SFT Strategies.

| Method | BUG Loc. | BUG Iden. | Code Rep. | Code Rec. | Avg. |
|---|---|---|---|---|---|
| *DSCoder-6.7B-Ins* | | | | | |
| Zero-Shot | 27.5 | 26.2 | 40.6 | 20.3 | 28.7 |
| w/ Vanilla SFT | 21.8 | 23.1 | 40.1 | 9.4 | 23.6 |
| w/ COAST (Answer) | 43.8 | 35.8 | 43.5 | 32.7 | 39.0 |
| w/ COAST (CoT) | **60.7** | 45.0 | 38.7 | 34.7 | 44.8 |
| NeuDebugger | 58.8 | **45.8** | **49.5** | **71.3** | **56.4** |
| *Llama3-8B-Ins* | | | | | |
| Zero-Shot | 57.6 | **36.6** | 29.6 | 74.9 | 49.7 |
| w/ Vanilla SFT | 53.6 | 34.0 | 28.7 | 26.0 | 35.6 |
| w/ COAST (Answer) | 58.1 | 34.8 | **42.5** | 32.1 | 41.9 |
| w/ COAST (CoT) | **64.4** | 34.6 | 32.6 | 78.1 | 52.4 |
| NeuDebugger | 61.1 | 33.8 | 41.8 | **78.5** | **53.8** |

## 6.1 Overall Performance

As shown in Table 3, we show the debugging performance of different LLMs on DEBUGEVAL.

Among the four tasks defined in DEBUGEVAL, LLMs typically excel in Bug Localization and Code Recognition tasks compared to other tasks, which demonstrates that LLMs have strong capabilities in code differentiation and bug detection. In contrast, these LLMs demonstrate lower effectiveness in both BUG Identification and Code Repair tasks. The poor performance of LLMs in the Bug Identification task highlights that current LLMs are weak at analyzing errors and cannot understand them effectively. The suboptimal performance of these 70B-scale LLMs in the Code Repair task further highlights the challenges they face in self-debugging (Chen et al., 2023). This phenomenon has also been observed by Wang et al. (2024).

The experimental results demonstrate that larger-scale LLMs generally exhibit superior code debugging capabilities compared to their smaller-scale counterparts. This illustrates the critical role of model scale in preserving emergent ability (Wei et al., 2022a) and highlights the effectiveness of training on code-related data for acquiring specialized knowledge (Yuan et al., 2024; Gudibande et al., 2024). Furthermore, we evaluate the performance of LLMs across various programming languages to investigate their effectiveness and robustness in addressing debugging challenges. Notably, some LLMs, such as GPT-4o-mini and DeepSeek-Coder-V2, show consistent performance across all programming languages, whereas some 7B-scale LLMs show significant variability in their performance across different languages. These observations highlight the necessity of establishing a comprehensive benchmark encompassing multiple programming languages to rigorously assess the debugging capabilities of LLMs.

Finally, we present the performance of NeuDebugger, which are finetuned using the synthesized data of COAST. These NeuDebugger models demonstrate superior code debugging capabilities compared to other 7B-scale LLMs, achieving performance on par with GPT-3.5-Turbo. Notably, both NeuDebugger-DS-6.7B and NeuDebugger-Llama3-8B outperform their corresponding foundation models across the four tasks outlined in DEBUGEVAL, with improvements of 27.7% and 4.1%, respectively. The phenomenon highlights the effectiveness of COAST in generating high-quality data to improve the debugging performance of LLMs.

## 6.2 Ablation Studies

As shown in Table 4, we conduct ablation studies to evaluate the effectiveness of various SFT strategies, including Vanilla SFT, COAST (Answer), COAST (CoT), and NeuDebugger. Additionally, the impact of data quantity on the performance of the NeuDebugger strategy across different LLMs is further discussed in the Appendix A.5.

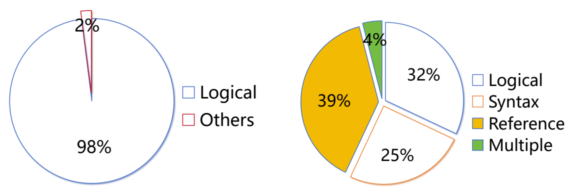
The Vanilla SFT strategy collects SFT data from UltraInteract (Yuan et al., 2024), InstructCoder (Hu et al., 2023), and RepairLlama (Silva et al., 2023) to finetune LLMs. The COAST framework generates SFT data through multi-agent interactions

Table 5: Performance of Vanilla LLMs and NeuDebugger on Different Bug Types. Ref. denotes Reference and Multi. denotes Multiple.

| Task | Model | Python | | | | C++ | | | | Java | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Syntax | Ref. | Logic | Multi. | Syntax | Ref. | Logic | Multi. | Syntax | Ref. | Logic | Multi. |
| Bug Iden. | Llama3-8B-Ins | 0.0 | 3.7 | 97.9 | 45.8 | 0.0 | 3.0 | 87.5 | 62.0 | 0.0 | 3.7 | 87.4 | 47.4 |
| | NeuDebugger-Llama3-8B | 34.2 | 16.8 | 27.9 | 75.3 | 17.0 | 3.0 | 15.0 | 84.5 | 25.3 | 4.2 | 23.2 | 80.5 |
| | DSCoder-6.7B-Ins | 3.2 | 3.2 | 99.5 | 0.5 | 2.0 | 0.0 | 100.0 | 2.0 | 1.6 | 1.6 | 100.0 | 0.5 |
| | NeuDebugger-DS-6.7B | 54.2 | 81.1 | 32.6 | 2.6 | 45.0 | 45.0 | 77.5 | 20.0 | 50.0 | 55.8 | 72.6 | 12.6 |
| Code Rep. | Llama3-8B-Ins | 40.0 | 20.0 | 35.6 | 7.5 | 60.9 | 45.8 | 29.6 | 16.7 | 40.0 | 52.4 | 28.3 | 7.7 |
| | NeuDebugger-Llama3-8B | 90.0 | 46.7 | 39.7 | 20.0 | 65.2 | 66.7 | 40.7 | 10.8 | 80.0 | 76.2 | 39.6 | 15.4 |
| | DSCoder-6.7B-Ins | 40.0 | 33.3 | 38.4 | 17.5 | 69.6 | 58.3 | 40.7 | 21.6 | 60.0 | 71.4 | 47.2 | 23.1 |
| | NeuDebugger-DS-6.7B | 90.0 | 46.7 | 45.2 | 27.5 | 87.0 | 62.5 | 40.7 | 27.0 | 76.0 | 85.7 | 54.7 | 30.8 |

and we conduct three different SFT strategies to show the effectiveness of different agents, including COAST (Answer), COAST (CoT), and NeuDebugger. Specifically, the COAST (Answer) model removes `Code Teacher` from COAST and directly takes correct answers as outputs for the synthesized data. COAST (CoT) asks the `Code Teacher` to generate reasoning process (thoughts) for problem-solving as the output. The NeuDebugger strategy combines both COAST (Answer) and COAST (CoT) strategies by merging training data from both strategies to finetune the `Code Learner`. Specifically, the data for the BUG Localization, BUG Identification, and Code Recognition tasks is sourced from COAST (CoT) strategy, while the Code Repair data come from COAST (Answer) strategy.

The evaluation results reveal that the Vanilla SFT strategy performs significantly worse than the baseline models, illustrating the challenge of ensuring high-quality data for finetuning LLMs. In contrast, using synthesized data generated by COAST markedly enhances the code debugging capabilities of these models. This suggests that both DSCoder-6.7B-Ins and Llama3-8B-Ins struggle to effectively acquire debugging knowledge from human or GPT-4 annotated data (Gudibande et al., 2024). Moreover, the COAST (CoT) strategy consistently outperforms COAST (Answer) in most tasks, except for the Code Repair task. This exception might be due to the Chain-of-Thought (CoT) explanations produced by `Code Teacher`, which offer clearer reasoning but potentially introduce noise specific to Code Repair scenarios. By integrating SFT data from both COAST (CoT) and COAST (Answer), the NeuDebugger strategy achieves the best overall performance among all SFT methods. Collectively, these experimental results highlight the effectiveness of the COAST framework, which leverages multiple agents to synthesize high-quality SFT data for fine-tuning LLMs.




(a) DSCoder-6.7B-Ins. (b) NeuDebugger-DS-6.7B.

Figure 4: Response Distributions of DSCoder-6.7B-Ins and NeuDebugger-DS-6.7B in BUG Identification Task.

### 6.3 Effectiveness of NeuDebugger on Different Bug Types

In Table 5, we evaluate the effectiveness of NeuDebugger models in different bug types.

For the Bug Identification task, the evaluation results reveal that existing LLMs remain suboptimal in analyzing the root causes of bugs. Figure 4 further shows the response distributions of these models, which illustrates a notable limitation of DSCoder-6.7B-Ins. DSCoder-6.7B-Ins often defaults to selecting "Logical Error", leading to a relatively high accuracy for this category but a lack of nuanced identification among different bug types. In contrast, NeuDebugger-DS-6.7B showcases its effectiveness by achieving a more balanced distribution in option selection, resulting in significant improvements in overall Bug Identification performance. Additional results of other LLMs on this task are provided in the Appendix A.6.

For the Code Repair task, NeuDebugger models achieve improvements on all bug types, particularly for Syntax Errors, showing the effectiveness of NeuDebugger models. This finding also suggests that Syntax Errors are relatively simpler and easier for LLMs to learn compared to the other bug types. However, NeuDebugger models still face challenges when addressing more complex issues such as Logic Errors and Multiple Errors.

## 7 Conclusion

This paper presents DEBUGEVAL, a comprehensive benchmark designed to evaluate the debugging capabilities of LLMs from multiple aspects. Our experiments reveal that 7B-scale LLMs consistently underperform in a wide range of debugging tasks. To address this limitation, we introduce COAST, a multi-agent based framework for synthesizing high-quality data to improve the debugging abilities of LLMs.

## Limitations

While DEBUGEVAL includes four evaluation tasks, incorporating additional tasks could enhance its ability to better evaluate the debugging capabilities of LLMs. In our experiments, the COAST framework demonstrates strong potential in generating high-quality SFT data for LLMs. However, its effectiveness is constrained by the performance of the `Code Quizzer` and `Code Teacher` models. In particular, the quality of the generated data heavily relies on the capabilities and diversity of the foundation model of `Code Quizzer` and `Code Teacher`.


## Acknowledgments

This work is partly supported by the Natural Science Foundation of China under Grant (No. 62206042, No. 62137001, and No. 62272093), CCF-Zhipu Large Model Innovation Fund (No. 202403), the Joint Funds of Natural Science Foundation of Liaoning Province (No. 2023-MSBA-081), and the Fundamental Research Funds for the Central Universities under Grant (No. N2416012).


## References


2024. Qwen2 technical report.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *ArXiv preprint*, abs/2303.08774.

AI@Meta. 2024. Llama 3 model card.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *ArXiv preprint*, abs/2309.16609.

Weilin Cai, Juyong Jiang, Fan Wang, Jing Tang, Sunghun Kim, and Jiayi Huang. 2024. A survey on mixture of experts.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. 2024. Coder: Issue resolving with multi-agent and task graphs, 2024. *URL https://arxiv. org/abs/2406.01304*.

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

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

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *ArXiv preprint*, abs/2304.05128.

DeepSeek-AI. 2024a. Deepseek llm: Scaling open-source language models with longtermism. *ArXiv preprint*, abs/2401.02954.

DeepSeek-AI. 2024b. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model.

Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*.

Arnav Gudibande, Eric Wallace, Charlie Victor Snell, Xinyang Geng, Hao Liu, Pieter Abbeel, Sergey Levine, and Dawn Song. 2024. The false promise of imitating proprietary language models. In *The Twelfth International Conference on Learning Representations*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024a. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

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. 2024b. Codeeditorbench: Evaluating code editing capability of large language models.

Brent Hailpern and Padmanabhan Santhanam. 2002. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12.

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. IEEE.

hiyouga. 2023. Llama factory.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022a. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022b. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Qisheng Hu, Kaixin Li, Xu Zhao, Yuxi Xie, Tiedong Liu, Hui Chen, Qizhe Xie, and Junxian He. 2023. Instructcoder: Empowering language models for code editing. *ArXiv preprint*, abs/2310.20329.

Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Sketchfix: a tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

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*, page 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*, abs/2403.07974.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*.

Lukas Kirschner, Ezekiel Soremekun, and Andreas Zeller. 2020. Debugging inputs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 75–86.

Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch generation with language models: Feasibility and scaling behavior. In *Deep Learning for Code Workshop*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, page 54–72.

Nicholas Lee, Thanakul Wattanawong, Sehoon Kim, Karttikeya Mangalam, Sheng Shen, Gopala Anumanchipali, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. 2024. Llm2llm: Boosting llms with novel iterative data enhancement. *ArXiv preprint*, abs/2403.15042.

Guochang Li, Chen Zhi, Jialiang Chen, Junxiao Han, and Shuiguang Deng. 2024. A comprehensive evaluation of parameter-efficient fine-tuning on automated program repair.

Kaixin Li, Qisheng Hu, Xu Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Qizhe Xie, and Junxian He. 2023. Instructcoder: Instruction tuning large language models for code editing.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online. Association for Computational Linguistics.

Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. Tbar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Xiao Liu, Kaixuan Ji, Yicheng Fu, WengLam Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2021. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *Cornell University - arXiv,Cornell University - arXiv*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv preprint*, abs/2102.04664.

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*.

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue.

OpenAI. 2024. Gpt-4o mini: advancing cost-efficient intelligence.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *ArXiv preprint*, abs/2308.12950.

André Silva, Sen Fang, and Martin Monperrus. 2023. Repairllama: Efficient representations and fine-tuned adapters for program repair. Technical report.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them.

Qwen Team. 2024. Introducing qwen1.5.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. Debugbench: Evaluating debugging capability of large language models.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *ArXiv preprint*, abs/2307.09288.

Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2024. Intervenor: Prompt the coding ability of large language models with the interactive chain of repair. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022a. Emergent abilities of large language models. *Transactions on Machine Learning Research*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022b. Chain-of-thought prompting elicits reasoning in large language models.

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

Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical program repair in the era of large pre-trained language models. *ArXiv preprint*, abs/2210.14179.

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, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 11941–11952. PMLR.

Lifan Yuan, Ganqu Cui, Hanbin Wang, Ning Ding, Xingyao Wang, Jia Deng, Boji Shan, Huimin Chen, Ruobing Xie, Yankai Lin, et al. 2024. Advancing llm reasoning generalists with preference trees. *ArXiv preprint*, abs/2404.02078.

Shengbin Yue, Wei Chen, Siyuan Wang, Bingxuan Li, Chenchen Shen, Shujun Liu, Yuxuan Zhou, Yao Xiao, Song Yun, Wei Lin, et al. 2023. Disc-lawllm: Fine-tuning large language models for intelligent legal services. *ArXiv preprint*, abs/2309.11325.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. *ArXiv preprint*, abs/2305.04087.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *ArXiv preprint*, abs/2406.11931.

# A Appendix

## A.1 License

For all datasets in our experiments, DebugBench uses the Apache License 2.0, LiveCodeBench uses the MIT License. All of these licenses allow their data to be used for academic use.

## A.2 Prompt Templates of Different Agents Defined in COAST

As illustrated in Figure 5, we utilize different prompts to guide LLMs to play the roles of `Code Quizzer`, `Code Learner`, and `Code Teacher`.

For each task defined in DEBUGEVAL, we individually select an instance for each programming language and then use them to prompt `Code Quizzer` to generate code debugging problems of different tasks, bug types, and programming languages. These problems are intended to evaluate `Code Learner`'s code debugging abilities. For the questions that `Code Learner` answers incorrectly, we use the following instruction: "You are an experienced and insightful code debugger" to prompt `Code Teacher` to generate explanations and solutions of the questions. Finally, questions, solutions, and explanations are collected to construct the SFT data for fine-tuning `Code Learner`.

**Prompts of COAST**

You are a code debugging expert, skilled in generating code debugging problems to challenge programmers. Please construct a similar question based on the example provided that is related to {question type} and contains the correct answer, and the buggy code in the question contains { error type }.
### Example
{Example}
### Problem

*Code Quizzer*

You are a code debugger. Please provide an answer to the following code debugging problem using your own knowledge.
### Problem
{Problem}
### Answer

*Code Learner*

You are an experienced and insightful code debugger. We present you with a question that student answered incorrectly and the corresponding correct answer. Please provide a detailed explanation for it.
### Problem
{Problem}
### Explanation
Let's think step by step.

*Code Teacher*

Figure 5: Illustrations of Prompts Used in COAST to Configure Different Agents. Within COAST, there are three LLM-based agents, including `Code Quizzer`, `Code Learner`, and `Code Teacher`. We utilize specific instructions to ensure they play the correct roles and carry out the intended tasks.

## A.3 Details of LLMs Used for Evaluation

In this subsection, we present a detailed description of the models evaluated on DEBUGEVAL.

*OpenAI GPTs.* `GPT-4o-mini` (OpenAI, 2024) and `GPT-3.5-Turbo` (OpenAI, 2022) are two popular and powerful LLMs, which belong to different variants of the GPT family, developed by OpenAI. `GPT-4o-mini` is a lightweight version of the `GPT-4o` model, but still inherits the core advantages of the `GPT-4o`, including powerful text generation, logical reasoning, and code generation. Both models are black-box models, which supply commercial APIs for usage.

*Meta Llama.* `Llama2-7B-Ins` (Touvron et al., 2023) is an open-sourced LLM. It is trained with up to 1.4 trillion tokens, where 4.5% of them are code tokens from Github. `CodeLlama-7B-Ins` (Roziere et al., 2023) conducts an additional instruction-tuning stage to adapt `Llama2` (Touvron et al., 2023) to improve the effectiveness in code-related tasks. Recently, `Llama3` (AI@Meta, 2024) models have been released, which is a major leap over `Llama2` models and establishes a new state-of-the-art.

*Aliyun Qwen.* `Qwen2-72B-Ins` is a 72 billion parameter scale LLM. `Qwen2-72B` employs a variety of automated methods for obtaining high-quality instruction and preference data, making it perform well on code and maths tasks. `CodeQwen1.5-7B-Ins` (Bai et al., 2023) is the code-oriented version of `Qwen1.5-7B` (Team, 2024). `CodeQwen1.5-7B-Ins` has been tuned with around 3 trillion tokens of code-related data. It supports 92 programming languages and supports long context understanding and generation with a context length of 64K tokens.

*DeepSeek.* DeepSeek series models are released by High-Flyer. `DeepSeek-LLM-7B` (DeepSeek-AI, 2024a) is trained from scratch with 2 trillion tokens in both English and Chinese. `DeepSeek-LLM-7B-Ins` (DeepSeek-AI, 2024a) is initialized by `DeepSeek-LLM-7B` and tuned with an additional 1 million instruction data. `DSCoder-6.7B-Ins` (Guo et al., 2024a) and `DSCoder-33B-Ins` (Guo et al., 2024a) are trained from scratch on 2T tokens, which consist of 87% code and 13% natural language. `DeepSeek-V2` (DeepSeek-AI, 2024b) contains 236B parameters and employs the Mixture-of-

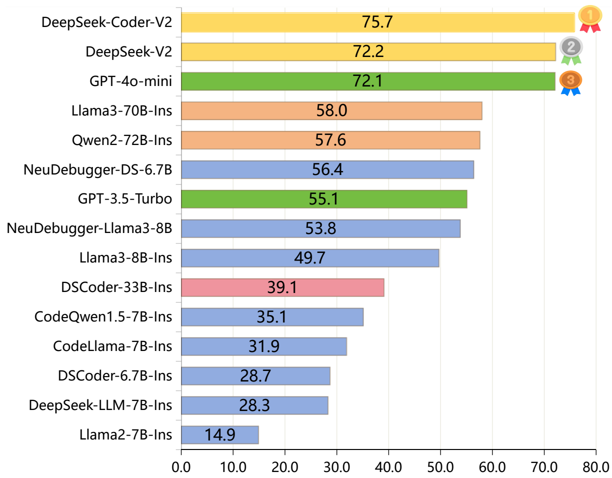
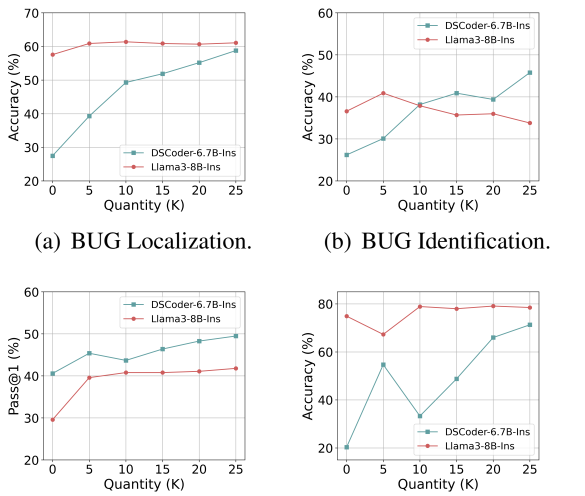


Figure 6: Debugging Performance of Different LLMs on DEBUGEVAL. Different LLMs are ranked according to their average performance. Different colors represent different LLM scales.



(a) BUG Localization. (b) BUG Identification.

(c) Code Repair. (d) Code Recognition.

Figure 7: Impact of the Amount of Training Data on Model Performance.

Experts (MoE) (Cai et al., 2024) architecture to conduct efficient training and inference. It is trained on a high-quality corpus comprising 8.1 trillion tokens. `DeepSeek-Coder-V2` (Zhu et al., 2024) is also an open-sourced MoE-based LLM, which achieves comparable performance with `GPT4-Turbo` in code-related tasks. `DeepSeek-Coder-V2` starts from an intermediate checkpoint of `DeepSeek-V2` and is tuned using 6 trillion tokens.

## A.4 Performance Ranking on DEBUGEVAL

As shown in Figure 6, we show the debugging performance of different LLMs on DEBUGEVAL.

The evaluation results show that larger-scale LLMs consistently outperform smaller-scale LLMs on DEBUGEVAL, highlighting the superior capacity of larger-scale LLMs to handle code debugging tasks. The DeepSeek series models exhibit particularly strong performance, with the open-source DeepSeek-Coder-V2 surpassing the closed-source GPT-4o-mini. NeuDebugger-DS-6.7B and NeuDebugger-Llama3-8B, based on DeepSeek-Coder-6.7B-Ins and Llama3-8B-Ins, respectively, demonstrate improvements of 27.7% and 4.1% when trained using the data synthesized by the COAST framework. It shows the effectiveness of COAST in generating high-quality training data to improve the debugging performance of LLMs.

## A.5 The Impact of Data Quantity

This subsection explores the impact of data quantity when finetuning LLMs using the NeuDebugger strategy. As shown in Figure 7, we finetune the DSCoder-6.7B-Ins and Llama3-8B-Ins models using varying amounts of SFT data points. We then assess their performance on the DEBUGEVAL and visualize the results.

Compared to Llama3-8B-Ins, DSCoder-6.7B-Ins shows a significant performance increase when more SFT data are fed. This indicates that code-oriented LLMs are better at learning from debugging data, whereas a standard language model struggles to enhance its debugging capabilities without an essential understanding of code. Across all debugging tasks defined by DEBUGEVAL, DSCoder-6.7B-Ins exhibits significant improvements in BUG Localization, BUG Identification, and Code Recognition, while only showing slight improvements in the code repair task. This suggests that these debugging data do indeed contribute to the better code-repair ability of LLMs, though the task remains challenging to improve significantly.

Table 6: Response Distributions of LLMs in BUG Identification Task. A.Syntax, B.Ref, C.Logic and D.Multi are four choices of the task.

| Model | Response Distributions (%) | | | |
|---|---|---|---|---|
| | A.Syntax | B.Ref | C.Logic | D.Multi |
| *Zero-Shot* | | | | |
| Llama3-8B-Ins | 0.0 | 0.8 | 71.0 | 28.2 |
| NeuDebugger-Llama3-8B | 11.4 | 3.9 | 8.6 | 76.1 |
| DeepSeek-Coder-V2 | 28.0 | 7.2 | 36.8 | 28.0 |

## A.6 Analysis of the Responses of LLMs in the Bug Identification Task

In Table 6, we show the response distributions of different models in the Bug Identification task.

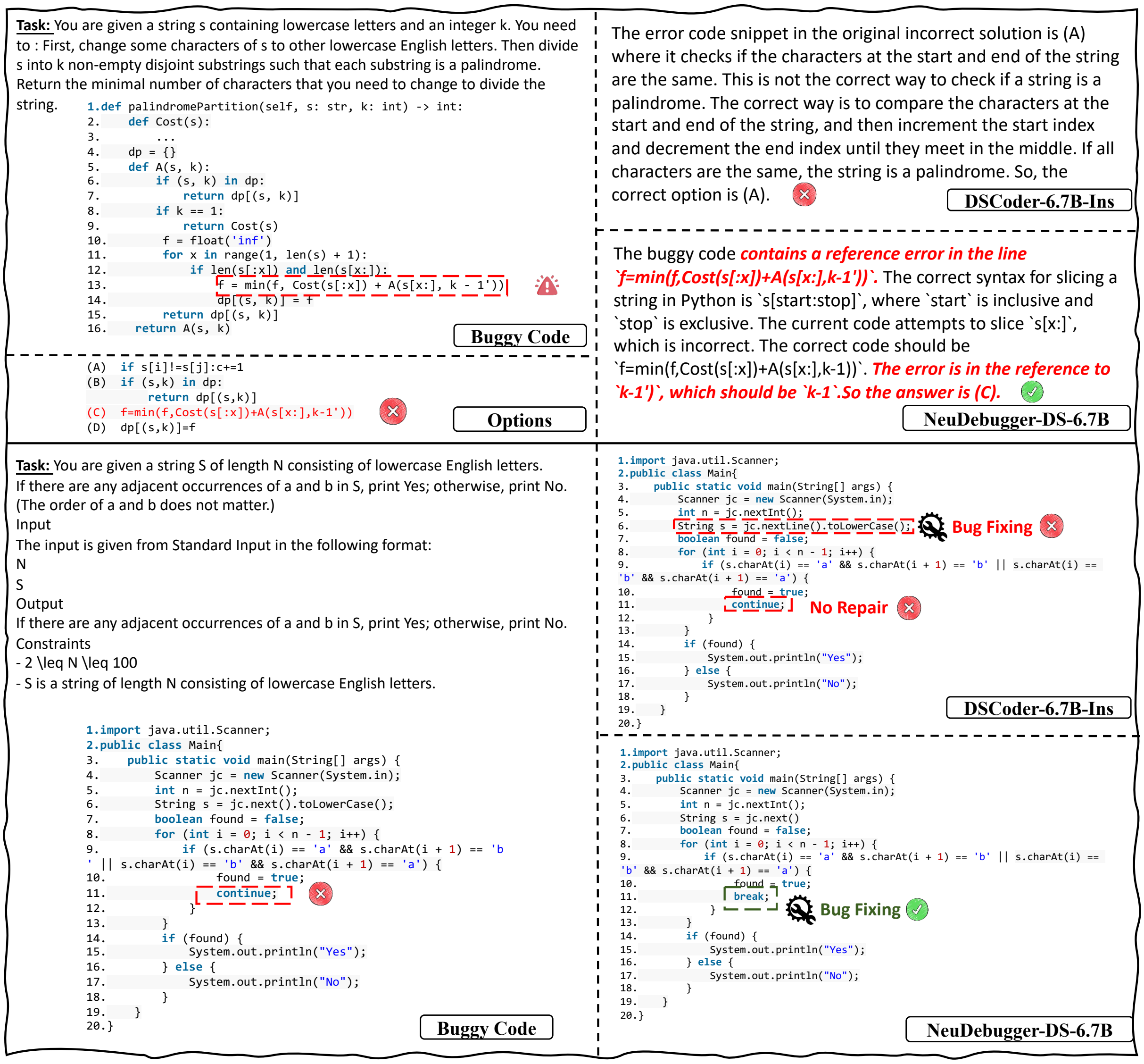



Figure 8: Case Studies. We provide two cases from BUG Localization task and Code Repair task to show the effectiveness of NeuDebugger.

The evaluation results show that LLama3-8B-Ins demonstrates a tendency to favor Logical Error and Multiple Errors. In contrast, DeepSeek-Coder-V2 performs in a more balanced manner compared to 7B-scale models, effectively avoiding random guessing behavior. In addition, NeuDebugger-LLama3-8B has a more even distribution of error types than the base model LLama3-8B-Ins, and is able to identify Syntax Error and Reference Error, which highlights NeuDebugger's ability to better identify error types rather than randomly guessing answers as Logical Error or Multiple Errors.

## A.7 Case Studies

As shown in Figure 8, we compare the performance of the DSCoder-6.7B-Ins model before and after training through two cases to demonstrate the effectiveness of NeuDebugger.

For the first case of the BUG Localization task, the code error is caused by the line `f = min(f, Cost(s[:x]) + A(s[x:], k-1'))`, which has an incorrect index `k-1'`. Thus, the correct answer is (C). DSCoder-6.7B-Ins considers the code fragment `if s[i]!=s[j]: c+=1` as erroneous, stating "This is not the correct way to check if a string is a palindrome". In contrast, NeuDebugger-DS-6.7B accurately analyzes the reason for the bug "contains an error in the line `f = min(f, Cost(s[:x]) + A(s[x:], k-1'))`, the error is in the reference to `k-1'`, which should be `k-1`", demonstrating its effectiveness in BUG Localization.

In the second case of the Code Repair task, the error involves the misuse of `continue`, which leads to a Logical Error. The DSCoder-6.7B-

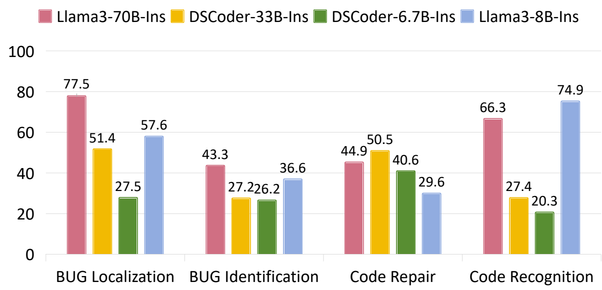



Figure 9: Debugging Performance of Different LLMs on DEBUGEVAL.

Ins model fails to identify this error and instead suggests changing the line `String s = jc.next().toLowerCase()` to `String s = jc.nextLine().toLowerCase()`. This modification introduces a new error, as it does not handle the input correctly. NeuDebugger-DS-6.7B accurately recognizes that the problem lies in the use of `continue` and changes `continue` to `break`, successfully resolving the bug.

### A.8 More Analyses of Tasks Defined in the DEBUGEVAL

This subsection provides an in-depth analysis of the tasks defined in DEBUGEVAL.

**Performance of Different LLMs.** As illustrated in Figure 9, we compare the performance of various LLMs in different tasks. The results reveal that their performance is inconsistent across tasks. For instance, while DSCoder-6.7B-Ins outperforms Llama3-8B-Ins in the Code Repair task, Llama3-8B-Ins achieves better results in Bug Localization, Bug Identification, and Code Recognition tasks. These findings illustrate the limitations of evaluating LLMs' code debugging capabilities based solely on a single Code Repair task, highlighting the importance of incorporating different tasks for evaluation.

**The Feasibility of Different Tasks.** The benchmark includes four task scenarios—Bug Localization, Bug Identification, Code Recognition, and Code Repair—that evaluate the debugging capabilities of LLMs. These tasks closely align with the skills demanded of developers in real-world scenarios. In the software development workflow, a developer typically identifies the buggy code, analyzes its root cause, implements a solution, and verifies the fix to ensure correctness. Thus DEBUGEVAL designs these tasks to mirror the key stages of real-world debugging.

The three tasks defined in DEBUGEVAL —BUG Localization, BUG Identification, and Code Recognition—are structured as single-choice question-answering problems, enhancing the accuracy of evaluation. This design is well-suited to the nature of these tasks. First, in real-world scenarios, code error localization involves pinpointing specific lines within faulty code, a process that naturally aligns with a single-choice problem. Second, BUG Identification requires classifying errors such as Syntax Error, Reference Error, Logical Error, and Multiple Errors, making it intuitive and efficient to adopt a single-choice format where the model selects the correct error type. Lastly, Code Recognition involves identifying the incorrect code fragment between two options, inherently fitting the single-choice question format.