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

Anonymous ACL submission

Abstract

Debugging is a crucial aspect of software development, but the debugging capabilities of Large Language Models (LLMs) have not been thoroughly investigated. In this paper, we present DEBUGEVAL, a comprehensive benchmark designed to assess the debugging proficiency of LLMs, comprising four distinct evaluation tasks: BUG Localization, BUG Identification, Code Review, and Code Repair. Then we evaluate the debugging ability of LLMs of different scales on DEBUGEVAL. Our experimental results show that 7B-scale LLMs exhibit nearly random performance in identifying bug types. To address this, we propose a **CO**mmunicative **A**gent based data **S**yn**T**hesis (COAST) model aimed at improving the debugging capabilities of 7B-scale LLMs. COAST establishes a multi-agent system to synthesize data for supervised fine-tuning (SFT) LLMs. Our analyses demonstrate that the SFT data synthesized by COAST is of high-quality and outperforms the data collected from human/GPT-4. All data and codes will be released via GitHub.

1 Introduction

In the realm of software development, code debugging is an indispensable process for ensuring the functionality and reliability of applications (Hailpern and Santhanam, 2002; Kirschner et al., 2020). With the complexity of software systems grows, traditional debugging methods, which often rely on heuristics (Le Goues et al., 2012; Wen et al., 2018) and predefined patterns (Hua et al., 2018; Liu et al., 2019), are reaching their limitations. The emergent ability of Large Language Models (LLMs) (OpenAI, 2022; Touvron et al., 2023) has opened up new horizons in automated debugging, offering a more flexible and comprehensive approach to identifying and rectifying code errors (Chen et al., 2023).

The capabilities of LLMs have been extensively explored for code-related tasks such as code generation and translation (Chen et al., 2021b; Luo et al., 2023; Zheng et al., 2023; Li et al., 2023b; Guo et al., 2024b). However, their debugging capabilities remain relatively underexplored. To better evaluate the code debugging ability, researchers are now building benchmarks to assess the code debugging capabilities of LLMs (Tian et al., 2024; Guo et al., 2024c). Nevertheless, existing code debugging benchmarks face two main issues: 1) They primarily design tasks related to code repair, which is insufficient for a comprehensive evaluation of code debugging ability. 2) Constructing buggy code using GPT-4 (Achiam et al., 2023) fails to capture the complexity and diversity of code errors encountered in real development environments.

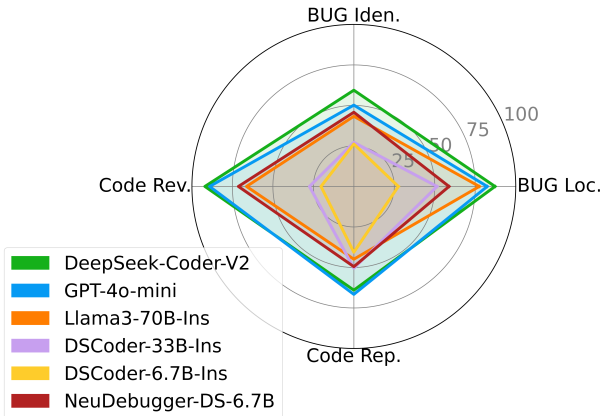


Figure 1: Debugging Ability of Different LLMs. We use the DEBUGEVAL to evaluate the debugging ability of LLMs at different scales.

To address these challenges, this paper introduces DEBUGEVAL, a comprehensive benchmark designed to evaluate the debugging capabilities of LLMs. DEBUGEVAL introduces four tasks: BUG Localization, BUG Identification, Code Review, and Code Repair, which are designed to test the LLMs' ability not only to identify and classify errors but also to provide correct code solutions. Besides, each task in DEBUGEVAL covers differ-

ent program languages, such as Python, C++, and Java. Additionally, DEBUGEVAL incorporates the real user-written buggy codes and GPT-4 generated buggy codes to better simulate real-world software development. While collecting user-written buggy codes, we also implement strict quality control to prevent data leakage in DEBUGEVAL. As shown in Figure 1, we evaluate the debugging effectiveness of different LLMs on DEBUGEVAL. The evaluation results indicate that LLMs with 7 billion parameters exhibit relatively weaker debugging capabilities than LLMs with 70 billion parameters or even larger. To enhance the debugging ability of these 7B-scale models, Supervised Fine-Tuning (SFT) is widely adopted to enhance the performance of LLMs in specific domains using human-labeled datasets or LLM-generated data (Roziere et al., 2023; Yue et al., 2023; Zhang et al., 2023). However, SFT's reliance on the availability and quality of labeled data limits its effectiveness.

To help produce high-quality debugging data for SFT, this paper proposes a **CO**mmunicative **A**gent based data **S**yn**T**hesis (COAST) framework. Specifically, COAST builds three agents: `Code Quizzer`, `Code Learner`, and `Code Teacher`, which collaborate to synthesize code debugging data for SFT the `Code Learner`. In the workflow of multiple agents in our COAST model, the `Code Quizzer` agent first generates a diverse set of code debugging problems and the `Code Learner` then acts as a critic to evaluate the educational value of the synthesized problems. Finally, these synthesized problems that the `Code Learner` answers incorrectly are collected as SFT data, and the `Code Teacher` provides detailed solutions and explanations for these problems. The synthesized data are used to train the `Code Learner` to conduct our NeuDebugger model.

Our experiments on DEBUGEVAL demonstrate that NeuDebugger models significantly enhance the debugging capabilities of 7B-scale LLMs by providing higher-quality SFT data. Notably, further analysis shows that collecting data solely for SFT fails to improve LLMs' debugging abilities (Gudibande et al., 2024). It illustrates that the quality of SFT data still challenges the debugging capability of LLMs, even if we collect the real-world data from human trials. In our COAST model, the `Code Teacher` effectively guides the `Code Learner` across three tasks—BUG Localization, BUG Identification, and Code Review—through the generation of Chain-of-Thought (CoT) reasoning (Wei et al., 2022a). However, CoT reasoning negatively impacts performance in the Code Repair task by introducing noise and disrupting the underlying code structure. During SFT, different LLMs exhibit distinct learning behaviors. In detail, the synthesized data notably improves the performance of code-oriented LLMs, such as DeepSeek-Coder-6.7B-Ins, more than some general LLMs like Llama3-8B-Ins. These findings offer valuable insights for facilitating future research to improve LLMs' debugging capabilities.

2 Related Work

Early debugging models primarily rely on feature-based methods, such as using 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 correct the buggy codes. However, the effectiveness of these feature-based debugging methods is limited in correcting different bug errors and dealing with more complex code bugs, because of the limited patterns or rules that are predefined.

With the development of Pretrained Language Models (PLMs), the work also follows the pretraining and finetuning strategy to build the debugging model and deal with various code bugs occurring in real life. For example, Xia et al. (2022) use the code-oriented pretrained model, CodeX (Chen et al., 2021a), to explore the capabilities of PLMs in debugging. It shows that CodeX (Chen et al., 2021a) achieves convincing code repair performance, especially on both Python and Java program languages. Kolak et al. (2022) also use GPT-2 (Radford et al., 2019) and CodeX (Chen et al., 2021a) to evaluate their effectiveness in generating the correct patch line when given corresponding code prefix. All the studies have proven that the effectiveness of PLMs based debugging models mainly thrives on the code understanding ability obtained during pretraining.

Different from previous debugging work, recent research focuses more on correcting the bugs generated by LLMs. Self-Debug (Chen et al., 2023) prompts LLMs to generate the code reviews to aid themselves in refining generated codes, while Self-Repair (Olausson et al., 2023) incorporates human-provided feedback for repairing the buggy code. Furthermore, Self-Edit (Zhang et al., 2023) trains an additional fault-aware editor to repair codes by leveraging the error messages from the test case

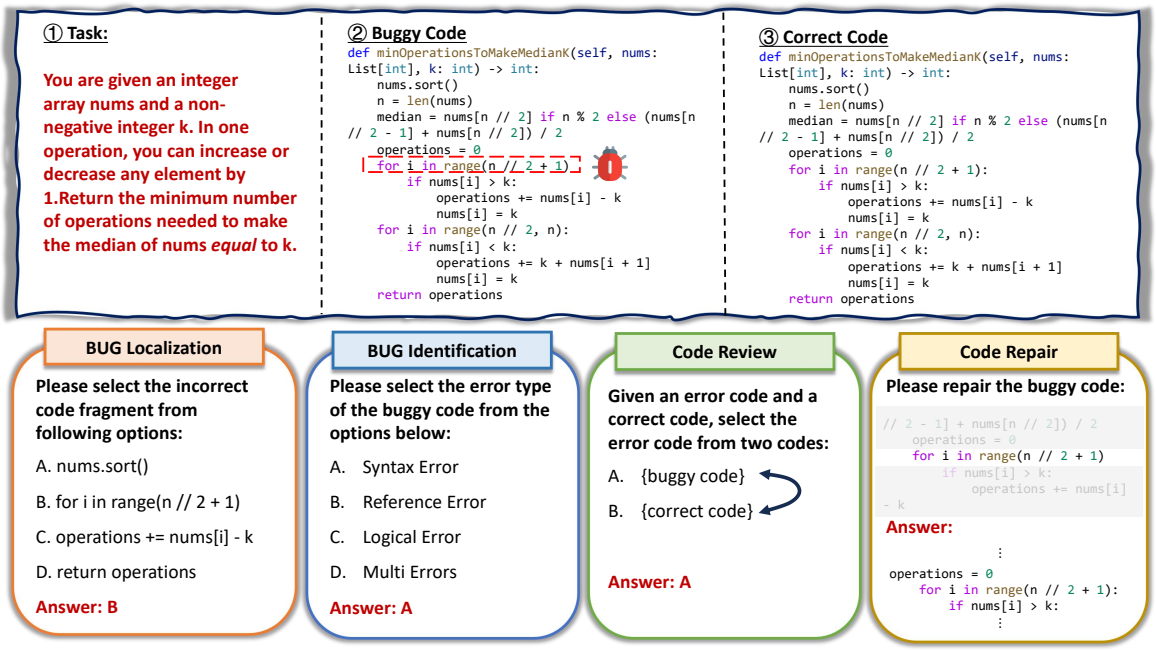


Figure 2: DEBUGEVAL Benchmark. DEBUGEVAL includes four tasks: BUG Localization, BUG Identification, Code Review, and Code Repair.

evaluation and generated code segments. Wang et al. (2024) further explore the effectiveness of interactive Chain-of-Repair (CoR), which uses LLMs to generate the guidelines for repairing codes by incorporating the generated codes and error messages from the code compiler. It is evident that the effectiveness of these code repair models mainly relies on the debugging ability of LLMs.

To improve the debugging ability of LLMs, recent work focuses more on generating data for supervised fine-tuning. InstructCoder (Li et al., 2023a) employs the Self-Instruct method (Wang et al., 2022) to build an instruction-tuning dataset to improve the effectiveness of LLMs in code debugging. Li et al. (2024) further construct the APR-INSTRUCTION dataset and utilize the dataset to fine-tune LLMs using four different Parameter-Efficient Fine-Tuning (PEFT) methods, such as 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 further evaluate the debugging capabilities of LLMs, some researchers focus on building benchmarks to evaluate LLMs. Wang et al. (2024) collect buggy code submitted by users from the Atcoder website to construct CodeError, which is used to assess the ability of LLMs to repair Python code. Tian et al. (2024) further propose DebugBench to explore the debugging capabilities of LLMs in Python, C++, and Java. They use GPT-4 (Achiam et al., 2023) to insert some code errors into code segments to synthesize the buggy codes and ask LLMs to repair codes. Nevertheless, only the code repair task is insufficient to comprehensively evaluate the debugging ability of LLMs.

3 Evaluating the Debugging Ability of LLMs with DEBUGEVAL

In this section, we first describe the task definition of the designed tasks in DEBUGEVAL (Sec. 3.1). Then we detail the process of constructing the DEBUGEVAL benchmark (Sec. 3.2).

3.1 Task Definition

DEBUGEVAL introduces four different tasks to evaluate LLMs' debugging capability including BUG Location, BUG Identification, Code Review, and Code Repair. We present the illustrations of these four evaluation tasks in Figure 2. As shown in Table 1, the comparison details of DEBUGEVAL and different debugging benchmarks are presented. Different from other debugging benchmarks, DEBUGEVAL is a multi-lingual and multi-task debugging benchmark, which conducts more complete evaluations on the debugging ability of LLMs. Then we delve deeper to describe each task.

BUG Localization. The BUG Localization task focuses on identifying the specific line(s) of code that contains the error. It evaluates the ability of

Table 1: A Comparison between DEBUGEVAL and Other Code Repair Benchmarks. The data statistics only on the testing set of each benchmark. CE indicates compilation errors (e.g. SyntaxError). The Source indicates where the buggy codes are collected.

Dataset	Language	Task	Size	Type	Source	Anti-Data Leakage
DeepFix (2021)	C	Code Repair	6,971	CE	User	✗
Review4Repair (2022)	Java	Code Repair	2,961	All	User	✗
Bug2Fix (2021)	Java	Code Repair	5,835	All	User	✗
Github-Python (2021)	Python	Code Repair	15K	CE	User	✗
FixEval (2023)	Java/Python	Code Repair	43K/243K	All	User	✗
CodeError (2024)	Python	Code Repair	4,463	All	User	✗
DebugBench (2024)	C++/Java/Python	Code Repair	1,438/1,401/1,414	All	GPT-4	✓
CodeEditorBench (2024c)	C++/Java/Python	Code Repair	676/515/716	All	GPT-4	✓
DEBUGEVAL	C++/Java/Python	BUG Localization	195/205/178	All	User GPT-4	✓
		BUG Identification	800/760/760			
		Code Review	800/800/800			
		Code Repair	138/138/138			

LLMs to point out the exact location where the error occurs within a code snippet. For each test instance of the BUG Localization task, we give a buggy code, extract four different code snippets from it that contain a faulty code snippet, and ask LLMs to identify which is the faulty code snippet.

BUG Identification. In this task, for the given program that contains code error(s), we ask LLMs to choose the error type from four choices. The potential choices include SyntaxError, ReferenceError, LogicalError, and MultipleErrors.

Code Review. For the Code Review task, we give the correct code and the buggy code to ask LLMs to distinguish the one that contained bugs. Specifically, there are only a few differences between correct code and incorrect code.

Code Repair. The Code Repair (Wang et al., 2024; Tian et al., 2024) task requires LLMs to generate a correct version of the given buggy code. It not only involves code error detection/identification evaluations, but also needs to correct the error of the code. After generating the corrected code, we evaluate the correctness of the corrected code by using the test cases.

3.2 Details of Data Construction

To ensure the quality of DEBUGEVAL, we collect high-quality data from existing benchmarks and reliable data sources, such as DebugBench (Tian et al., 2024), LiveCodeBench (Jain et al., 2024), and the AtCoder website[1].

Bug Localization. For the Bug Localization task, we sample test instances from the DebugBench. In more detail, we build the dataset for the Bug Localization task by sampling up to 20 instances of each of the 15 single code error types for each programming language. We compare the buggy code to the correct code, find the line of code that contains the error as the golden choice, and randomly pick up another three lines from the buggy code segment to construct the other three confusing choices.

Bug Identification. To conduct the evaluation on the Bug Identification task, we also sample test instances from DebugBench. Specifically, the choices of the task are divided into four types: SyntaxError, ReferenceError, LogicalError, and MultipleErrors. We compare the sizes of the test sample sets for different types of code errors and then select the smallest set size as the sampling number. Finally, we sample test instances from the different code error test sets according to this sampling number.

Code Review. We first mix the test instances from both DebugBench and LiveCodeBench. Then we collect the test pairs from the mixed dataset by randomly sampling 800 test instances from each programming language. Each test problem consists of both buggy and correct code segments and asks LLMs to choose the buggy one.

Code Repair. For the Code Repair task, we first collect 138 newly released programming competition problems from Atcoder from September 1, 2023, to April 1, 2024. Then we collect buggy code submissions from users and respectively choose one buggy code from Python, C++, and Java languages. Lots of buggy codes (82.7%) have the following submissions that correct the buggy codes and pass all test cases, which are regarded as the golden answer of the buggy code. For the rest of these buggy codes (17.3%), we use different LLMs to correct the buggy codes until the corrected code passes all test cases.

[1] https://atcoder.jp

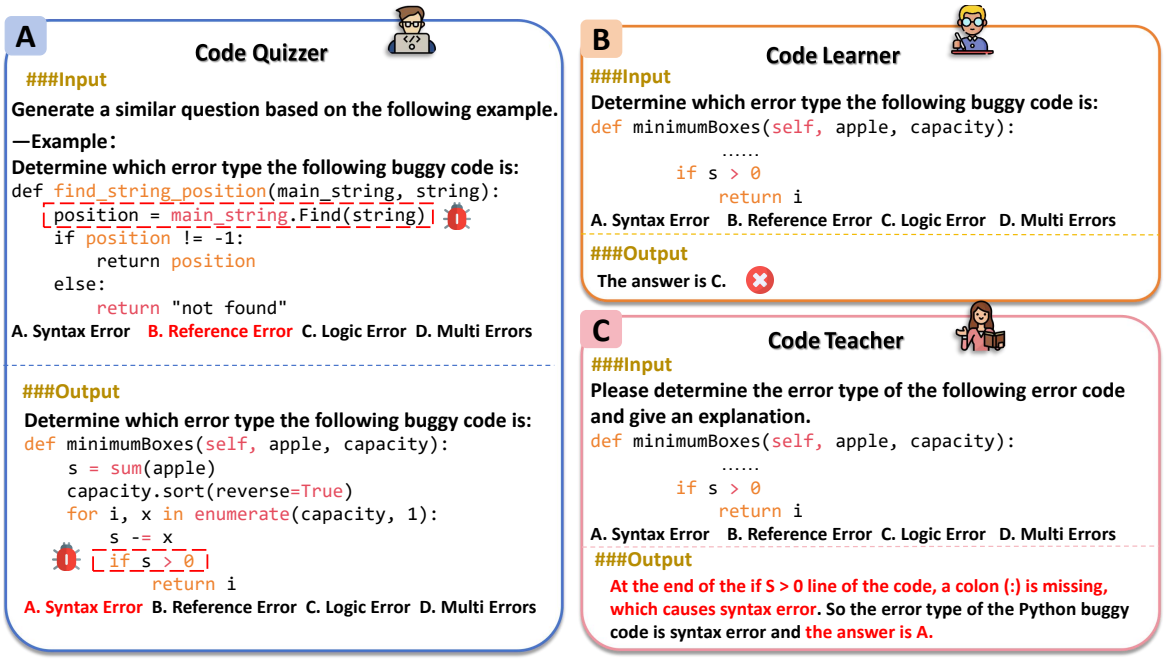


Figure 3: Illustration of **CO**mmunicative **A**gent based data **S**yn**T**hesis (COAST) Framework.

4 Communicative Agent Based Data Synthesis Framework

This section introduces the **CO**mmunicative **A**gent based data **S**yn**T**hesis (COAST) framework, which automatically produces code debugging data by agent interactions. As shown in Figure 3, we first configure the agents to play different roles in the COAST (Sec. 4.1). Then COAST employs different agents to synthesize high-quality training data for SFT (Sec. 4.2).

4.1 Agent Building

COAST constructs three agents that collaboratively generate and resolve the debugging problem data to synthesize high-quality SFT datasets. More details of the prompt designs of different agents are shown in Appendix A.2.

Code Quizzer. The `Code Quizzer` is designed to generate high-quality problems for synthesizing SFT data by using the instruction: "You are a code debugging expert, skilled in generating code debugging problems to challenge programmers". We use a stronger LLM as the `Coder Quizzer` and select examples from the DEBUGEVAL task in different languages to prompt `Coder Quizzer` to generate the corresponding task problems in different languages for `Coder Learner` to answer.

Code Learner. The `Code Learner` shares the same backbone model as the SFT model and serves as the critic to evaluate the educational value of the problems generated by the `Code Quizzer`. Using the prompt: "You are a student, please provide an answer to the following code debugging question using your own knowledge", the `Code Learner` is tasked with solving the problem based on its memorized knowledge. The educational value of the problem is then assessed by determining whether the `Code Learner` can correctly solve it, thereby contributing to the finetuning process.

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 debugger". This prompt directs the LLM to act as a proficient code debugger to produce the Chain-of-Thought (Wei et al., 2022b) as detailed solutions to the problems, which can better guide LLM during the SFT process.

4.2 Synthesizing SFT Data Using Communicative Agents

COAST automatically synthesizes high-quality data through the multi-agent collaboration, leveraging the expertise of stronger models to enhance the capabilities of weaker ones. The data synthesizing process is presented as follows.

In the initial step (**Step A**), the `Code Quizzer` synthesizes various code debugging problems based on examples from the DEBUGEVAL task. To ensure the task diversity and error diversity in

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 Review data	4,420
	Code Repair data	11,317

the synthesized data, we instruct the `Code Quizzer` to generate debugging problems with different error types, which align with the problems of the tasks defined in DEBUGEVAL. After synthesizing the debugging problems, 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 thereby the problem can be discarded. On the other hand, if the `Code Learner` produces an incorrect solution, the problem is reserved as the 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 detailed explanations and solutions. These Chain-of-Thought based explanations may include error explanations and the correct solution to solve the problems, which are essential for the `Code Learner` to comprehend the problems and refine their solutions. The responses generated by the `Code Teacher` are treated as the final outputs for the synthesized problems, forming the SFT data to fine-tune our NeuDebugger model.

5 Experimental Methodology

In this section, we describe the Supervised Fine-Tuning (SFT) strategies, evaluation metrics, details of evaluated foundation models, and implementation details of our experiments.

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

Evaluation Metrics. For the BUG Localization, BUG Identification, and Code Review tasks, LLMs are required to select one answer from multiple choices. Following prior work (Suzgun et al., 2022), we use accuracy as the evaluation metric for these tasks. For the Code Repair task, in line with previous studies (Chen et al., 2021c; Zheng et al., 2023; Luo et al., 2023; Guo et al., 2024a), we adopt Pass@1 (Chen et al., 2021c) to measure the effectiveness of various LLMs.

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

Implementation Details. For all LLMs, we set the generation temperature to 0.2 and the maximum generation length to 1024. For the closed-source models, we use the API provided by the respective vendors and for the open-source models, we use the vLLM (Kwon et al., 2023) framework for inference. To fine-tune the NeuDebugger model, we use DeepSeek-Coder-6.7B-Ins (Guo et al., 2024a) and Llama3-8B-Ins (AI@Meta, 2024) as backbone models and leverage the data synthesized by COAST for SFT. During SFT, we train all models using Llama-Factory (hiyouga, 2023) and utilize LoRA (Hu et al., 2022b) for efficient fine-tuning. In our experiments, we set the learning rate to 2e-5 and the training epoch to 1. We optimize models using the AdamW optimizer, with the batch size of 8 and the gradient accumulation steps of 4.

6 Evaluation Results

In this section, we show the debugging performance of different LLMs and NeuDebugger on DEBUGEVAL. Then we conduct ablation studies and explore the effectiveness of NeuDebugger in handling the problems of different code error types. Case studies are shown in Appendix A.6.

6.1 Overall Performance

As shown in Table 3, we evaluate the debugging effectiveness of different LLMs on DEBUGEVAL.

Overall, larger-scale LLMs exhibit stronger code debugging capability than small-scale LLMs. This phenomenon underscores the importance of model

Table 3: Evaluation Results of Different LLMs on DEBUGEVAL. DS denotes the DeepSeek Models.

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

Table 4: Effectiveness of Different Modules of COAST in Synthesizing SFT Data. NeuDebugger collects the SFT data from COAST (Anwser) and COAST (CoT).

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

scale in maintaining emergent abilities and acquiring critical knowledge through supervised fine-tuning (SFT) on code data (Yuan et al., 2024; Gudibande et al., 2024). Compared with other 7B-scale LLMs, NeuDebugger significantly enhances the code debugging effectiveness of foundation models and achieves competitive performance comparable to the 70B models. Besides, both NeuDebugger-DS-6.7B and NeuDebugger-Llama3-8B perform better than corresponding foundation models, DSCoder-6.7B-Ins and Llama3-8B-Ins, on the four tasks of DEBUGEVAL, bringing improvements of 27.7% and 4.1%, respectively. These improvements demonstrate that COAST can produce high-quality data for SFT and significantly improve model performance on debugging tasks.

Among the four tasks defined in DEBUGEVAL, LLMs typically produce better results in both BUG Localization and Code Review tasks. This indicates that these LLMs have strong code understanding capabilities by finetuning with code generation tasks, allowing them to effectively identify buggy code snippets. On the contrary, these LLMs demonstrate less effectiveness in both BUG Identification and Code Repair tasks. For the BUG Identification task, LLMs are required to identify the cause of bugs. The degenerated performance of LLMs in this task illustrates the difficulty current LLMs have deriving bug causes. The Code Repair task is even more complex, requiring LLMs to locate buggy snippets, determine the error type, and fix the code. The suboptimal performance of these 70B LLMs further indicates the challenges they face in self-debugging (Chen et al., 2023). Wang et al. (2024) have also confirmed this phenomenon.

The model performance across various programming languages reveals the effectiveness and robustness of different LLMs. For instance, the GPT-4o-mini and DeepSeek-Coder-V2 models exhibit consistent performance across all languages. In contrast, some LLMs demonstrate inconsistent performance across different programming languages. These findings underscore the necessity of developing a benchmark to evaluate debugging effectiveness across various programming languages, further supporting the motivation of our paper in building the DEBUGEVAL benchmark.

6.2 Ablation Studies

As shown in Table 4, ablation studies are conducted to explore the effectiveness of different SFT strategies, including Vanilla SFT, COAST (Answer), COAST (CoT), and NeuDebugger. The impact of data quantity on the SFT of different LLMs is shown in Appendix A.5.

The Vanilla SFT strategy gathers high-quality SFT data from UltraInteract (Yuan et al., 2024), InstructCoder (Hu et al., 2023), and RepairLlama (Silva et al., 2023) for fine-tuning LLMs. Then, we use the COAST framework to construct the SFT data and explore three different SFT strategies: COAST (Answer), COAST (CoT), and NeuDebugger. COAST (Answer) indicates that we remove the `Code Teacher` in COAST and ask

Table 5: Performance of Vanilla LLMs and NeuDebugger on Different Bug Types. Syntax, Ref, Logic, and Multi represent SyntaxError, ReferenceError, LogicalError, and MultipleErrors, respectively.

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**

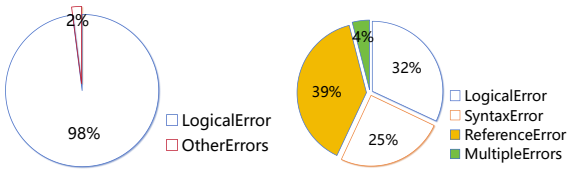
LLMs to directly give the correct choice during SFT. COAST (CoT) asks the `Code Learner` to mimic the thought of problem-solving of the `Code Teacher`. The NeuDebugger method combines the SFT strategies from both COAST (Answer) and COAST (CoT) by mixing the datasets from both SFT strategies for finetuning backbone models, except for the code repair data of COAST (CoT).

The evaluation results show that both DSCoder-6.7B-Ins and Llama3-8B-Ins perform worse with the Vanilla SFT method compared to the baseline, indicating that the data quality for SFT remains a challenge in fine-tuning LLMs. When the COAST synthesized data is used for SFT, the code debugging ability of these models is significantly enhanced. This illustrates that both DSCoder-6.7B-Ins and Llama3-8B-Ins are less effective at learning debugging knowledge from human/GPT-4 annotated data (Gudibande et al., 2024). Furthermore, the COAST (CoT) method generally achieves much better performance than COAST (Answer), except for the code repair task. The reason may lie in that the Chain-of-Thought outcomes generated by `Code Teacher` can better explain the reasons behind answer choices but might incorporate additional noise in code repair tasks. By combining SFT data from both COAST (CoT) and COAST (Answer), NeuDebugger achieves the best performance among all SFT strategies. All these experimental results demonstrate the effectiveness of the COAST model, which employs multiple agents to synthesize high-quality SFT data.

6.3 Effectiveness of NeuDebugger on Different Bug Types

As shown in Table 5, we show the effectiveness of NeuDebugger on different tasks.

For the BUG Identification task, the evaluation results demonstrate that existing LLMs are still sub-optimal for debugging tasks. Both DSCoder-6.7B-Ins and Llama3-8B-Ins achieve over 97% accuracy on LogicalError, but perform poorly on other coder error types. Furthermore, Figure 4 illustrates the choice distribution that is selected by LLMs. Such a distribution indicates that DSCoder-6.7B-Ins lacks the ability to identify bugs and defaults to selecting LogicalError, resulting in a high accuracy. NeuDebugger-DS-6.7B shows its effectiveness by conducting a more balanced answer choice distribution and achieving significant improvements in the Bug Identification task.



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

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

For the Code Repair task, we observe that NeuDebugger achieves improvements across almost all code error types, especially for SyntaxError, showing its effectiveness for code debugging. This also illustrates that SyntaxError is relatively simpler and easier for LLMs to learn when we compare it to the other three types of code errors. Besides, NeuDebugger also struggles with repairing code containing LogicalError or MultipleErrors.

7 Conclusion

This paper introduces DEBUGEVAL, a comprehensive benchmark designed to evaluate the debugging capabilities of LLMs from multiple dimensions. Through DEBUGEVAL, we observe that 7B-scale LLMs exhibit significantly weaker performance across various debugging tasks. To address this, we propose COAST, a novel approach that leverages communicative agents to synthesize high-quality debugging data for SFT. COAST demonstrates its effectiveness by significantly improving the debugging abilities of these LLMs on all evaluation tasks.

Limitations

DEBUGEVAL comprises four evaluation tasks, but the dataset for each remains relatively small. It is necessary to further increase the quantity and diversity of the data to ensure it can test LLMs' debugging capabilities across a wider range of error types and programming scenarios. Additionally, although the COAST framework can generate high-quality SFT data for LLMs, it is limited by the capabilities of the `Code Teacher` model. This highlights that the quality and diversity of the data generated by COAST are directly influenced by the performance of `Code Teacher`.

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.

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.

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. 2021c. 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.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024b. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *ArXiv preprint*, abs/2401.14196.

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. 2024c. 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.

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.

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. 2023a. Instructcoder: Instruction tuning large language models for code editing.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023b. Starcoder: May the source be with you! *ArXiv preprint*, abs/2305.06161.

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.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *ArXiv preprint*, abs/2306.08568.

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, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022a. 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. 2022b. 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.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

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 for academic use.

A.2 Prompt Templates Used in COAST

As illustrated in Figure 5, we employ 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 three instances for each programming language and then use them to prompt `Code Quizzer` to generate corresponding code debugging problems of different tasks and programming languages. These problems are intended to evaluate the `Code Learner`'s ability to solve the corresponding debugging tasks. For the questions that `Code Learner` answers incorrectly, we use the instruction: "You are an experienced and insightful debugger" to prompt the `Code Teacher` to generate the correct solution and explanation of the questions. Finally, questions, solutions, and explanations are collected to construct the SFT data for fine-tuning `Code Learner`.

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 it 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.

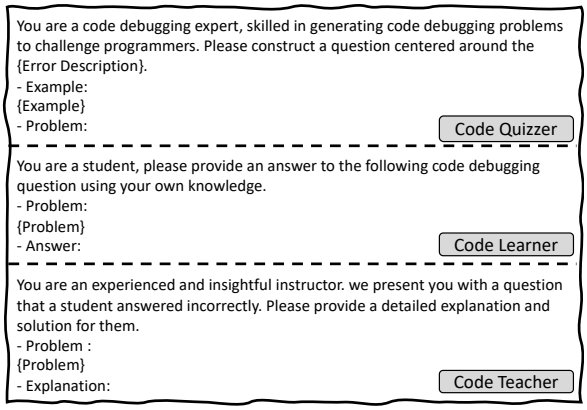


Figure 5: Illustrations of Prompts Used in COAST to Configure the Agents. Within COAST, there are three LLM-based agents, including `Code Quizzer`, `Code Learner`, and `Code Teacher`. We utilize specific instructions to ensure that they play the correct roles and carry out the intended tasks.

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-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

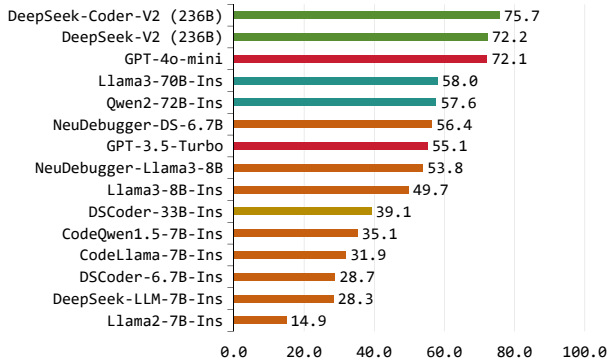
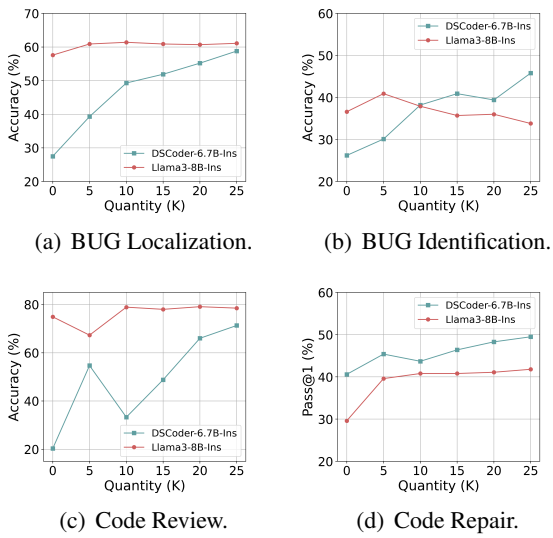


Figure 6: Debugging Performance on DEBUGEVAL. Different LLMs are ranked according to their average performance. Different colors are used to represent different model scales.



(a) BUG Localization. (b) BUG Identification. (c) Code Review. (d) Code Repair.

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

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 models 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 on data synthesized by COAST. This underscores COAST's ability to generate high-quality training data that significantly enhances model performance on debugging tasks.

A.5 The Impact of Data Quantity

This subsection explores the impact of data quantity when fine-tuning LLMs using the SFT data synthesized by COAST. As shown in Figure 7, we fine-tune the DSCoder-6.7B-Ins and Llama3-8B-Ins models using varying amounts of SFT data points. We then assess their debugging abilities by evaluating their performance on the DEBUGEVAL benchmark and visualizing the results.

Compared to Llama3-8B-Ins, DSCoder-6.7B-Ins shows a significant performance decrease when more SFT data points 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. Among all debugging tasks, DSCoder-6.7B-Ins exhibits significant improvements in BUG Localization, BUG Identification, and Code Review, while only showing slight improvements in the code repair task. This suggests that these debugging tasks do indeed contribute to the better code repair ability of LLMs, though the task remains challenging to improve significantly.

A.6 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". On the contrary, NeuDebugger-DS-6.7B accurately analyzes the reason of bugs "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 mistake. The DSCoder-6.7B-Ins model fails to identify this error and in-

Task: You are given a string s containing lowercase letters and an integer k. You need to : First, change some characters of s to other lowercase English letters. Then divide s into k non-empty disjoint substrings such that each substring is a palindrome. Return the minimal number of characters that you need to change to divide the string.

```
1.def palindromePartition(self, s: str, k: int) -> int:
2.    def Cost(s):
3.        ...
4.    dp = {}
5.    def A(s, k):
6.        if (s, k) in dp:
7.            return dp[(s, k)]
8.        if k == 1:
9.            return Cost(s)
10.        f = float('inf')
11.        for x in range(1, len(s) + 1):
12.            if len(s[:x]) and len(s[x:]):
13.                f = min(f, Cost(s[:x]) + A(s[x:], k - 1'))
14.                dp[(s, k)] = f
15.        return dp[(s, k)]
16.    return A(s, k)
```

Buggy Code

```
(A)  if s[i]!=s[j]:c+=1
(B)  if (s,k) in dp:
         return dp[(s,k)]
(C)  f=min(f,Cost(s[:x])+A(s[x:],k-1'))
(D)  dp[(s,k)]=f
```

Options

The error code snippet in the original incorrect solution is (A) where it checks if the characters at the start and end of the string are the same. This is not the correct way to check if a string is a palindrome. The correct way is to compare the characters at the start and end of the string, and then increment the start index and decrement the end index until they meet in the middle. If all characters are the same, the string is a palindrome. So, the correct option is (A).

DSCoder-6.7B-Ins

The buggy code ***contains a reference error in the line `f=min(f,Cost(s[:x])+A(s[x:],k-1'))`***. The correct syntax for slicing a string in Python is `s[start:stop]`, where `start` is inclusive and `stop` is exclusive. The current code attempts to slice `s[x:]`, which is incorrect. The correct code should be `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`.So the answer is (C).***

NeuDebugger-DS-6.7B

Task: You are given a string S of length N consisting of lowercase English letters. If there are any adjacent occurrences of a and b in S, print Yes; otherwise, print No. (The order of a and b does not matter.)
Input
The input is given from Standard Input in the following format:
N
S
Output
If there are any adjacent occurrences of a and b in S, print Yes; otherwise, print No.
Constraints
- 2 \leq N \leq 100
- S is a string of length N consisting of lowercase English letters.

```
1.import java.util.Scanner;
2.public class Main{
3.    public static void main(String[] args) {
4.        Scanner jc = new Scanner(System.in);
5.        int n = jc.nextInt();
6.        String s = jc.next().toLowerCase();
7.        boolean found = false;
8.        for (int i = 0; i < n - 1; i++) {
9.            if (s.charAt(i) == 'a' && s.charAt(i + 1) == 'b
' || s.charAt(i) == 'b' && s.charAt(i + 1) == 'a') {
10.                found = true;
11.                continue;
12.            }
13.        }
14.        if (found) {
15.            System.out.println("Yes");
16.        } else {
17.            System.out.println("No");
18.        }
19.    }
20.}
```

Buggy Code

```
1.import java.util.Scanner;
2.public class Main{
3.    public static void main(String[] args) {
4.        Scanner jc = new Scanner(System.in);
5.        int n = jc.nextInt();
6.        String s = jc.nextLine().toLowerCase();
7.        boolean found = false;
8.        for (int i = 0; i < n - 1; i++) {
9.            if (s.charAt(i) == 'a' && s.charAt(i + 1) == 'b' || s.charAt(i) ==
'b' && s.charAt(i + 1) == 'a') {
10.                found = true;
11.                continue;
12.            }
13.        }
14.        if (found) {
15.            System.out.println("Yes");
16.        } else {
17.            System.out.println("No");
18.        }
19.    }
20.}
```

Bug Fixing

No Repair

DSCoder-6.7B-Ins

```
1.import java.util.Scanner;
2.public class Main{
3.    public static void main(String[] args) {
4.        Scanner jc = new Scanner(System.in);
5.        int n = jc.nextInt();
6.        String s = jc.next()
7.        boolean found = false;
8.        for (int i = 0; i < n - 1; i++) {
9.            if (s.charAt(i) == 'a' && s.charAt(i + 1) == 'b' || s.charAt(i) ==
'b' && s.charAt(i + 1) == 'a') {
10.                found = true;
11.                break;
12.            }
13.        }
14.        if (found) {
15.            System.out.println("Yes");
16.        } else {
17.            System.out.println("No");
18.        }
19.    }
20.}
```

Bug Fixing

NeuDebugger-DS-6.7B

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

stead 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 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.