MDEVAL: Massively Multilingual Code Debugging

Anonymous ACL submission

Abstract

Code large language models (LLMs) have made significant progress in code debugging by directly generating the correct code based on the buggy code snippet. Programming benchmarks, typically consisting of buggy code snippets and their associated test cases, are used to assess the debugging capabilities of LLMs. However, many existing benchmarks primarily focus on Python and are often limited in terms of language diversity (e.g., DebugBench and DebugEval). To advance the field of multilingual debugging with LLMs, we propose the first massively multilingual debugging benchmark, which includes 3.9K test samples of 20 programming languages and covers the automated program repair (APR) task, the bug localization(BL) task, and the bug identification (BI) task. In addition, we introduce the debugging instruction corpora MDEVAL-INSTRUCT by injecting bugs into the correct multilingual queries and solutions (xDebugGen). Further, a multilingual debugger xDebugCoder trained on MDEVAL-INSTRUCT as a strong baseline specifically to handle bugs of a wide range of programming languages (e.g. “Missing Mut” in language Rust and “Misused Macro Definition” in language C). Our extensive experiments on MDEVAL reveal a notable performance gap between open-source models and closed-source LLMs (e.g., GPT and Claude series), highlighting huge room for improvement in multilingual code debugging scenarios.

1 Introduction

Large language models (LLMs) (OpenAI, 2023; Touvron et al., 2023a; Yang et al., 2024a) designed for code, such as CodeLlama (Rozière et al., 2023), DeepSeekCoder (Guo et al., 2024a), and Qwen-Coder (Hui et al., 2024), are highly effective in code understanding and generation. These capabilities make them particularly useful for debugging, where deep comprehension of code structure and logic is essential. Automated program repair (APR)

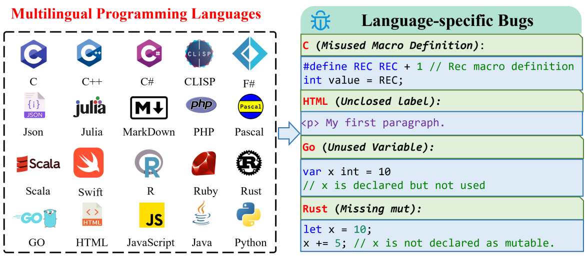


Figure 1: Massively multilingual evaluation task comprised of three tasks, including code generation, code completion, and code explanation.

(Wen et al., 2024) aims to automatically fix bugs without human involvement, significantly reducing time and costs in development processes.

LLMs have recently shown considerable potential in this area. For instance, CodeX (Chen et al., 2021) and GPT-4 series (OpenAI, 2023) outperforming previous conventional methods have demonstrated promising results on bug benchmarks such as QuixBugs (Lin et al., 2017). The recent work DebugBench (Tian et al., 2024) creates a debugging benchmark including Python, Java, and CPP for LLM evaluation. However, for the diverse programming languages in Figure 1, the multilingual debugging scenario poses more language-specific challenges for APR. Multilingual issues (e.g. “Misused Macro Definition” in programming language C, “Missing mut” in Rust, and “Unused Variable” in Go) highlight the complexities and diversities of locating and fixing bugs in the multilingual debugging scenario. Therefore, there is an urgent need to build a truly massively multilingual debugging code benchmark with a wide variety of generic and language-specific bug types.

To further characterize the debugging performance of LLMs across different programming languages, we introduce MDEVAL, a framework for data construction, evaluation benchmark, and a multilingual debugging baseline xDebugCoder, to advance the development of code debugging. First, we propose MDEVAL, the first massively multilingual evaluation benchmark for code debugging covering 20 programming languages and 3.9K samples to assess the capabilities of LLMs across a wide range of languages. Further, we create MDEVAL-INSTRUCT, a multilingual debugging instruction corpus in 20 languages to help the LLM fix the bug given the buggy code snippet. Besides, we propose xDebugGen to create the buggy and correct code pair for debugging instruction tuning. The bugs are injected into the queries and solutions with our designed three strategies (1) Injecting bugs into query. (2) Injecting bugs into solution. (3) Injecting bugs with the round-trip code translation. Leveraging MDEVAL-INSTRUCT, we develop xDebugCoder as a strong baseline, assessing the transferability of LLMs in multilingual debugging tasks.

The contributions are summarized as follows: (1) We propose MDEVAL, a comprehensive multilingual code debugging benchmark consisting of 3.9K samples spanning three tasks: automated program repair (APR), code localization (BL), and bug identification (BI). This benchmark covers 20 languages and includes both generic and language-specific bug types. (2) We introduce the massively multilingual code debugging instruction corpora MDEVAL-INSTRUCT created by xDebugGen. By injecting bugs into the correct multilingual query or response, we can create pairs of buggy code and the correct code for instruction tuning. (3) We systematically evaluate the multilingual code debugging capabilities of 40 models on our created MDEVAL and create a leaderboard to evaluate them on 20 programming languages dynamically. Notably, extensive experiments suggest that comprehensive multilingual multitask evaluation can realistically measure the gap between open-source (e.g. DeepSeekCoder and Qwen-Coder) and closed-source models (e.g. Claude series).

2 MDEVAL

2.1 Data Overview

In Table 1, the MDEVAL consists of 3.9K problems. Following Yang et al. (2024c), we design 3 multilingual debugging-related tasks: Automated Program Repair, Bug Localization, and Bug Identification. Each task contains about 1.3K questions, with more than 60 problems in each language. Each problem in MDEVAL includes *question, example test cases, buggy code, correct code, and unit tests.*

We calculate the length of the question and buggy code using the CodeLlama tokenizer (Rozière et al., 2023). The average question length is 83 words, highlighting their detailed descriptive nature. The average buggy code length is 239 tokens, indicating the complexity of the code. In addition, the total number of unit tests for the dataset is 6,838, to ensure the accuracy of the bug-fix judgment.

Statistics	Number
Problems	$3,897$
Automated Program Repair	$1,299$
Bug Localization	$1,299$
Bug Identification	$1,299$
Total Test Cases	$7,133$
#Difficulty Level	
- Easy/Medium/Hard	$1,146/1,407/1362$
Length	
Question	
- *maximum length*	291 tokens
- *minimum length*	7 tokens
- *avg length*	70 tokens
Buggy code	
- *maximum length*	$19,265$ tokens
- *minimum length*	15 tokens
- *avg length*	320.6 tokens

Table 1: MDEVAL dataset statistics.

In Table 2, we compare MDEVAL with other code debugging benchmarks. Our benchmark provides a valuable enhancement to existing ones, significantly expanding the variety of programming languages and introducing language-specific error types, along with a greater number of questions and diverse bug-fixing tasks. The error types in MDEVAL are shown in Figure 2. Figure 3 plots error types distribution. We strive to cover all error types in each language. Due to the inherent differences among languages, we ensure a balanced distribution of difficulty levels, leading to variations in the distribution of error types across languages.

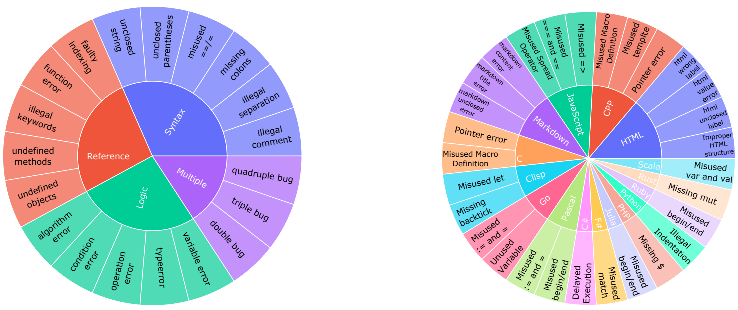
2.2 Data Construction & Quality Control

To curate the massively multilingual code debugging evaluation benchmark MDEVAL, we employ a comprehensive and systematic human annotation process for multilingual code samples. This process is guided by meticulously defined guidelines to guarantee accuracy and consistency.

We initially recruite 13 computer science graduates as multilingual debugging annotators, all proficient in their respective programming languages. After completing a comprehensive training course on annotation methods, the annotators are tasked with defining problems, providing corresponding solutions, and buggy code. Annotators adhere to the following principles: (1) Write a clear problem question and design test cases to ensure that bugs

Benchmark	#Languages	#Task	Size (Easy/Middle/Hard)	#Error Types	Source of Bugs	Language-specific Bugs
DeepFix (Yasunaga and Liang, 2021)	1	1	6,971	4	Collection	✗
Github-Python (Yasunaga and Liang, 2021)	1	1	15K	14	Collection	✗
Bug2Fix (Lu et al., 2021)	1	1	5,835	-	Collection	✗
FixEval (Haque et al., 2023)	2	1	43K/243K	-	Collection	✗
CodeError (Wang et al., 2023)	1	1	4,463	6	Collection	✗
CodeEditorBench (Guo et al., 2024b)	3	1	676/515/716	14	GPT-4 Generation	✗
DebugBench (Tian et al., 2024)	3	1	1,438/1,401/1,414	18	GPT-4 Generation	✗
DebugEval (Yang et al., 2024c)	3	4	1,933/1,903/1,876	18	Collection & GPT-4 Generation	✗
MDEVAL (Ours)	20	3	1,692/1,209/612	47	Human Annotation	✓

Table 2: Comparison between MDEVAL and other code debugging benchmarks. MDEVAL provides a comprehensive multilingual view by expanding the variety of programming languages and language-specific error types.



(a) Generic Errors

(b) Language-specific Errors

Figure 2: Error types in MDEVAL. Part (a) shows generic error types, and Part (b) lists language-specific error types.

can be effectively identified; (2) Categorize bugs into multiple difficulty levels (easy/medium/hard) based on the complexity of fixing these code.

Figure 4 illustrates the overall process of dataset construction. We begin by collecting code snippets from GitHub, which are then extracted and filtered following StarCoder (Li et al., 2023). Prior to the annotation phase, we summarize generic error types and language-specific error types. The three task definitions and corresponding annotation methods are explained in detail. The annotators proceede to annotate the code according to the identified error types and specified annotation methods. To ensure annotation quality, they evaluate the annotated code based on four criteria: problem difficulty, ambiguity, error type, and solvability. Furthermore, after completing their annotations, each annotator exchanges data with another annotator for cross-refining, aiming to minimize subjective bias and errors. Any discrepancies between annotators are resolved through consensus or with input from senior annotators. Finally, we engage three volunteers to assess the accuracy of the benchmark (targeting > 90%) and correct errors.

2.3 Instruction Corpora for Code Debugging

To create the instruction corpora, we need to create the pair of the correct code snippet and the buggy code. First, we select the proper code snippet from 20 languages and prompt the code LLM to generate a new question q^{L_k} of programming language L_k. Then, we use the LLM to generate the correct code c^{L_k} and filter the low-quality response with an LLM filter and the generated test cases. Therefore, we can regard the (q^{L_k}, c^{L_k}) as the correct sample by ensuring the correctness of c^{L_k} as much as possible. We propose xDebugGen comprised of the following three strategies to create the code debugging instruction corpora MDEVAL-INSTRUCT to obtain the fine-tuned LLM xDebugCoder.

Injecting Bugs into Query. We can prompt the LLM to modify the original question to another similar question with minor differences, where the similar question q'^{L_k} is used to generate the answer $\mathcal{M}_w(q'^{L_k})$ by a weak LLM $\mathcal{M}_w$ with small size (e.g. Qwen2.5-1.5B). Since there exist differences between the original question q^{L_k} and the modified question q'^{L_k}, $(\mathcal{M}_w(q'^{L_k}), c^{L_k})$ can be fed into the LLM as source input and target prediction.

Injecting Bugs into Solution. Another more intuitive method is to directly inject the bugs into the correct code c^{L_k}. Given the bug type and the correct code snippet, we prompt the LLM to generate the buggy code $\mathcal{M}(c^{L_k})$. The pair $(\mathcal{M}(c^{L_k}), c^{L_k})$ can be used for the instruction tuning.

Injecting Bugs with Round-trip Code Translation. Under the multilingual scenario, we can translate the correct c^{L_k} into the $\mathcal{M}_w(c^{L_k}; L_k \rightarrow$

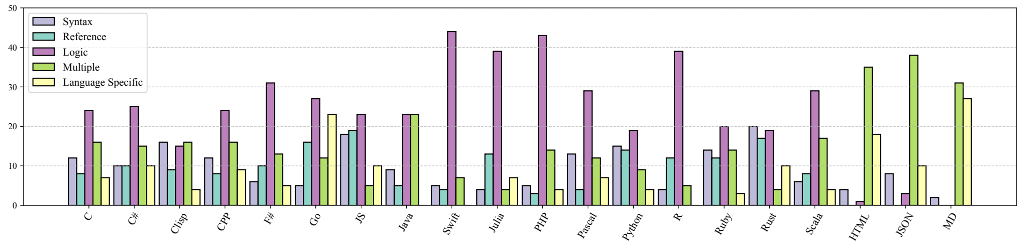


Figure 3: Error types distribution in 20 programming Languages from the MDEVAL.

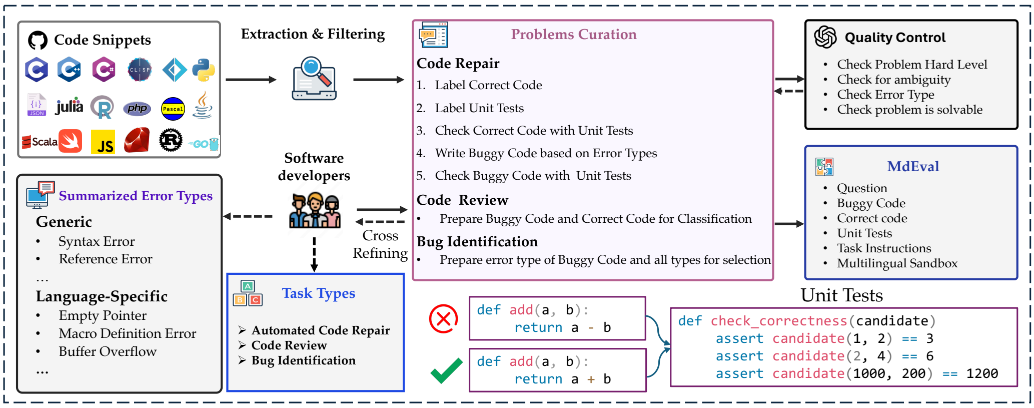


Figure 4: Overview of the MDEVAL construction process. We collect and filter code snippets from GitHub. Before annotation, we summarize error types. Annotators then label the code based on these types. To ensure quality, they use GPT-4o to evaluate the annotations on four criteria: difficulty, ambiguity, error type, and solvable. Finally, they exchange data with each other to minimize bias and errors.

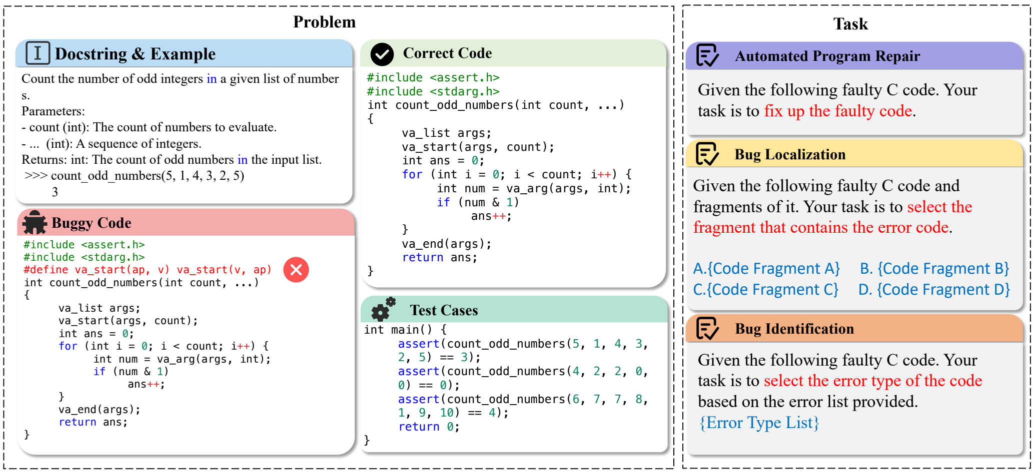


Figure 5: Examples of multilingual automated program repair, bug localization, and bug identification.

L_j) and then back-translate into the original language L_k of programming languages using the weak LLM $\mathcal{M}$, where the round-trip translation code snippet can be regarded as the buggy code. The pair $(\mathcal{M}_w(\mathcal{M}_w(c^{L_k}; L_k \rightarrow L_j); L_j \rightarrow L_k), c^{L_k})$ can be used for the instruction tuning.

2.4 Evaluation Task

Automated Program Repair (APR). The automated program repair task forces the LLM to fix the bug in the given code snippet and then generates the correct code. Given the programming language $L_k \in \{L_i\}_{i=1}^{K}$ ($K = 20$ is the number of programming languages), we provide the question q^{L_k}, the corresponding buggy code b^{L_k}, and the examples test cases e^{L_k} for inputs. We can organize the different input settings for evaluation:

$$r^{L_k} = \mathbb{I}(P(c^{L_k}|I; \mathcal{M}); u^{L_k}) \quad (1)$$

where $\mathbb{I}(\cdot)$ is the executor of the multilingual sandbox to verify the correctness of the generated code with the test cases u^{L_k} (If the fixed code c^{L_k} passes all test cases, the evaluation result $r^{L_k} = 1$, else 0). In our work, we provide three settings for evaluation to simulate the realistic user queries: (1) Question with buggy code: $I = \{q^{L_k}, b^{L_k}\}$ (2) Buggy code with example test cases: $I = \{b^{L_k}, e^{L_k}\}$ (3) Only buggy code: $I = \{b^{L_k}\}$.

Bug Localization (BL). The Bug Localization (BL) task aims to identify the specific line(s) of code within a given buggy program c^{L_K} that contains the error. For each test instance in the BL task, a buggy code c^{L_K} is provided, from which four code snippets, S_A, S_B, S_C, S_D, are extracted. The LLMs are then tasked with identifying the golden snippet S_G, which contains the error.

Bug Identification (BI). In this task, LLMs are required to classify the type of error present in a given buggy program c^{L_k} with one error. The LLMs must choose the correct error category from 47 bug types (including generic bug types and language-specific bug types).

2.5 Evaluation Metrics

Automated Program Repair. In the automated program repair task, we evaluate models by executing the generated code against a set of unit tests and assessing performance using the Pass@1 metric (pass rate for just one-time generation). Greedy Pass@1 indicates whether a result produced by the LLM successfully passes corresponding unit tests.

Bug Localization & Bug Identification. In the bug localization and bug identification tasks, we evaluate model performance using accuracy, as both require the model to select from a set of provided options.

3 Experiments

3.1 Experiment Setup

Code LLMs. We evaluate 40 popular models, including GPTs (OpenAI, 2023), Claude-3.5 (Anthropic, 2023), and code-specific models like Qwen2.5-Coder (Hui et al., 2024), DeepSeek-Coder (Guo et al., 2024a), CodeLlama (Rozière et al., 2023), and Codegemma (Gemma Team, 2024). Additionally, we fine-tune Qwen2.5-Coder-7B as our baseline xDebugCoder.

xDebugCoder Training Setup The training data for xDebugCoder comprises our debugging dataset MDEVAL-INSTRUCT and the Magicoder-Instruct code generation dataset (Wei et al., 2023), ensuring fundamental instruction-following capabilities for code-related tasks. xDebugCoder, built on Qwen2.5-Coder-7B, is trained for 3 epochs using a cosine scheduler with an initial learning rate of 5×10^{-5} with a 3% warmup ratio. We employ AdamW (Loshchilov and Hutter, 2017) as the optimizer, with a batch size of 1024 and a maximum sequence length of 2048. (Details can be found in the Appendix).

3.2 Main Results

Automated Program Repair. Table 3 presents the Pass@1 results of different models on MDEVAL for the multilingual automated program repair task (given question with buggy code, request the model to fix buggy code). The results indicate a marked disparity between closed-source state-of-the-art models and the majority of open-source models across nearly all programming languages. Notably, GPT-4o, Claude-3.5-sonnet, and Qwen2.5-Coder-Instruct excel in this task and demonstrate significant performance advantages over other models. Furthermore, our baseline model xDebugCoder, is fine-tuned using only 16K bug-related data MDEVAL-INSTRUCT. Despite the limited size of this dataset, the model demonstrated competitive performance compared to others of similar scale, highlighting the effectiveness of MDEVAL-INSTRUCT in enhancing the debugging capabilities of models.

Bug Localization Table 5 illustrates the accuracy of different models on the multilingual bug localization task. It is evident that closed-source models outperform open-source models by a significant margin, demonstrating the superior bug localization capabilities of closed-source models. Specifically, open-source models with smaller parameter sizes like OpenCoder-1.5B-Instrcut, due to their poor instruction-following capabilities, are unable to output the correct format as required, resulting in lower accuracy in localization. Besides, it is observed that for the same model, the bug localization accuracy is lower than its pass@1 scores in automated program repair task. We hypothesize that this discrepancy arises because the bug localization task requires a strong understanding of location information, which happens to be a weakness of large language models. Therefore, improving LLMs' ability to understand location information is a critical issue that needs to be addressed.

Bug Identification. In the bug identification task, the goal is to identify the error type in a given code snippet, where the LLMs analyze source code for defects and choose the correct bug type from the pre-defined 47 bug types. Table 4 lists the

Model	Size	Avg$_{all}$	C	C#	CLISP	CPP	F#	Go	HTML	JS	Java	Json	Julia	MD	PHP	Pascal	Python	R	Ruby	Rust	Scala	Swift
Closed-Source Models																						
o1-preview	🔒	70.2	68.6	73.1	91.7	63.8	89.2	38.6	15.5	84.0	90.0	39.0	89.6	20.0	84.1	80.0	91.8	60.0	93.7	85.7	84.4	56.7
o1-mini	🔒	72.9	65.7	76.1	60.0	68.1	81.5	68.7	5.2	80.0	91.7	42.4	92.5	25.0	87.0	78.5	90.2	88.3	96.8	98.6	87.5	60.0
GPT-4o-240806	🔒	67.5	14.3	64.1	85.0	66.7	86.2	56.6	13.8	57.3	83.3	42.4	80.6	20.0	87.0	72.3	91.8	86.7	84.1	84.3	89.1	86.7
GPT-4o-mini-240718	🔒	65.3	18.6	64.1	71.7	57.6	75.4	56.6	8.6	61.3	85.0	47.5	83.6	23.3	85.5	67.7	88.5	80.0	81.0	87.1	76.6	85.0
GPT-4-Turbo-240409	🔒	61.7	24.3	53.7	63.3	49.3	84.6	50.6	3.4	60.0	80.0	35.6	74.6	21.7	81.2	75.4	95.0	76.7	82.5	81.4	81.2	56.7
Claude-3.5-sonnet-240620	🔒	66.0	34.3	56.2	83.3	60.6	83.1	63.9	5.2	65.3	70.0	47.5	68.7	20.0	76.8	67.7	91.8	71.7	84.1	90.0	93.8	80.0
Claude-3.5-sonnet-241022	🔒	70.3	81.4	57.8	86.7	59.1	89.2	44.6	8.6	60.0	91.7	44.1	82.1	21.7	82.6	75.4	82.0	80.0	85.7	88.6	93.8	90.0
Yi-lighting	🔒	57.8	24.3	53.7	60.0	55.1	67.7	41.0	5.2	60.0	76.7	25.4	82.1	8.3	78.3	63.1	91.7	75.0	79.4	81.4	78.1	46.7
Doubao-Pro	🔒	60.2	68.6	55.2	56.7	55.1	78.5	53.0	8.6	56.0	80.0	15.3	70.1	8.3	72.5	66.2	85.0	81.7	82.5	87.1	78.1	35.0
0.5B+ Models																						
Qwen2.5-Instruct	0.5B	20.6	28.6	10.4	8.3	14.5	9.2	1.2	13.8	45.3	28.3	10.2	26.9	5.0	17.4	13.8	39.3	6.7	58.7	24.3	18.8	31.7
DS-Coder-Instruct	1.3B	33.6	28.6	42.2	13.3	43.9	24.6	38.6	5.2	44.0	48.3	18.6	47.8	1.7	33.3	27.7	44.3	16.7	61.9	41.4	34.4	45.0
Qwen2.5-Instruct	1.5B	35.5	24.3	32.8	15.0	27.5	23.1	18.1	8.6	60.0	50.0	28.8	55.2	8.3	34.8	30.8	62.3	20.0	69.8	67.1	32.8	35.0
OpenCoder-Instruct	1.5B	34.8	15.7	13.4	20.0	26.1	26.2	15.7	12.1	57.3	58.3	15.3	55.2	8.3	36.2	47.7	54.1	31.7	68.3	52.9	51.6	28.3
Yi-Coder-Chat	1.5B	32.4	37.1	34.4	3.3	30.3	7.7	28.9	8.6	45.3	53.3	15.3	55.2	1.7	34.8	41.5	52.5	28.3	49.2	42.9	28.1	40.0
Qwen2.5-Coder-Instruct	1.5B	34.8	11.4	26.9	15.0	30.4	16.9	20.5	17.2	61.3	45.0	28.8	58.2	10.0	40.6	36.9	55.7	28.3	60.3	58.6	29.7	40.0
Qwen2.5-Instruct	3B	46.0	51.4	40.3	28.3	36.2	41.5	41.0	12.1	69.3	61.7	27.1	61.2	11.7	53.6	47.7	63.9	45.0	58.7	65.7	53.1	38.3
6B+ Models																						
DS-Coder-Instruct	6.7B	56.3	37.1	60.9	56.7	63.6	60.0	56.6	8.6	61.3	75.0	23.7	64.2	6.9	52.2	60.0	78.7	51.7	88.9	80.0	60.9	68.3
CodeQwen1.5-chat	7B	42.6	34.3	34.4	43.3	33.3	41.5	42.2	10.3	54.7	55.0	20.3	62.7	8.6	49.3	41.5	52.5	30.0	69.8	62.9	34.4	61.7
CodeLlama-Instruct	7B	27.2	2.9	20.3	25.0	25.8	24.6	22.9	19.0	53.3	6.7	15.3	37.3	12.1	24.6	33.8	42.6	16.7	50.8	48.6	14.1	41.7
CodeGemma-Instruct	7B	45.9	34.3	32.8	3.3	43.9	44.6	44.6	19.0	60.0	68.3	25.4	64.2	0.0	56.5	36.9	65.6	40.0	73.0	67.1	56.2	70.0
Qwen2.5-Instruct	7B	50.4	57.1	47.8	38.3	50.7	61.5	26.5	8.6	60.0	81.7	32.2	61.2	8.3	73.9	47.7	70.5	50.0	58.7	62.9	60.9	45.0
Qwen2.5-Coder-Instruct	7B	61.7	58.6	60.9	61.7	60.6	70.8	47.0	19.0	60.0	81.7	37.3	74.6	22.4	73.9	61.5	77.0	65.0	73.0	78.6	70.3	75.0
OpenCoder-Instruct	8B	53.4	10.0	56.2	46.7	50.0	66.2	8.4	13.8	66.7	78.3	27.1	74.6	15.5	62.3	53.8	77.0	61.7	76.2	81.4	71.9	75.0
Meta-Llama-3-Instruct	8B	37.9	51.4	35.8	8.3	42.0	30.8	21.7	10.3	60.0	41.7	20.3	49.3	0.0	55.1	40.0	57.4	50.0	49.2	48.6	46.9	28.3
Meta-Llama-3.1-Instruct	8B	42.1	57.1	41.8	26.7	40.6	44.6	21.7	6.9	56.0	60.0	20.3	49.3	8.3	65.2	32.3	50.8	40.0	58.7	61.4	53.1	38.3
Yi-Coder-Chat	9B	50.6	45.7	54.7	28.3	47.0	40.0	42.2	22.4	65.3	76.7	20.3	58.2	3.4	52.2	58.5	65.6	45.0	68.3	68.6	71.9	68.3
14B+ Models																						
Qwen2.5-Instruct	14B	57.7	58.6	62.7	61.7	66.7	60.0	21.7	13.8	62.7	78.3	28.8	59.7	10.0	69.6	66.2	80.3	68.3	74.6	77.1	76.6	56.7
DS-Coder-V2-Lite-Instruct	2.4/16B	56.7	10.0	56.2	43.3	56.1	81.5	50.6	10.3	58.7	76.7	28.8	68.7	17.2	65.2	63.1	72.1	71.7	76.2	80.0	60.9	81.7
Starcoder2-Instruct-v0.1	15B	34.2	10.0	34.3	20.0	33.3	29.2	25.3	5.2	50.7	46.7	16.9	50.7	0.0	37.7	56.9	44.3	35.0	57.1	58.6	39.1	25.0
20B+ Models																						
Codestral-v0.1	22B	56.1	72.9	64.2	43.3	63.8	63.1	31.3	10.3	64.0	85.0	27.1	79.1	11.7	63.8	47.7	68.9	55.0	79.4	72.9	68.8	41.7
Qwen2.5-Instruct	32B	65.8	64.3	53.7	75.0	50.7	87.7	53.0	10.3	65.3	93.3	32.2	74.6	13.3	81.2	75.4	90.2	80.0	85.7	82.9	84.4	58.3
Qwen2.5-Coder-Instruct	32B	68.2	78.6	60.9	75.0	56.1	83.1	44.6	13.8	61.3	91.7	33.9	85.1	22.4	82.6	64.6	91.8	80.0	79.4	82.9	82.8	91.7
DS-Coder-Instruct	33B	57.7	65.7	59.4	46.7	50.0	70.8	39.8	19.0	65.3	75.0	28.8	73.1	10.3	58.0	55.4	73.8	61.7	79.4	68.6	78.1	70.0
CodeLlama-Instruct	34B	28.6	70.0	23.9	18.3	26.1	15.4	18.1	10.3	40.0	18.3	25.4	46.3	3.3	24.6	24.6	49.2	11.7	60.3	38.6	14.1	25.0
Meta-Llama-3-Instruct	70B	50.1	27.1	29.9	61.7	34.8	73.8	4.8	10.3	56.0	75.0	27.1	76.1	13.3	75.4	73.8	70.5	60.0	73.0	60.0	64.1	43.3
Meta-Llama-3.1-Instruct	70B	56.6	48.6	49.3	55.0	44.9	75.4	8.4	17.2	61.3	71.7	35.6	83.6	16.7	79.7	67.7	75.4	63.3	76.2	77.1	81.2	46.7
DS-V2.5	21/236B	65.1	14.3	60.9	70.0	62.1	78.5	51.8	12.1	61.3	80.0	40.7	83.6	23.3	82.6	69.2	83.6	80.0	81.0	87.1	92.2	86.7
DS-V3	37/671B	64.9	42.9	59.7	66.7	55.1	83.1	42.2	8.6	70.7	81.7	40.7	88.1	13.3	81.2	76.9	90.0	75.0	88.9	82.9	92.2	55.0
Qwen2.5-Instruct	72B	63.6	62.9	53.7	68.3	56.5	81.5	34.9	10.3	62.7	81.7	37.3	67.2	21.7	82.6	69.2	90.2	76.7	87.3	82.9	82.8	61.7
xDebugGen (Our Method)	7B	47.5	21.4	57.8	36.7	62.1	60.0	31.3	17.2	56.0	33.3	25.4	67.2	12.1	62.3	41.5	60.7	43.3	61.9	77.1	45.3	70.0

Table 3: Pass@1 (%) scores of different models for Automated Program Repair tasks on MDEVAL. The underlined numbers are the best scores for each language. "Avg$_{all}$" represents the average scores of all code languages.

all results of the bug identification. Notably, the closed-source LLMs, such as GPT-4o and Claude series, have the dominant advantages, outperforming the open-source LLMs by nearly +10 points. Bug identification with 47 bug types poses a daunting challenge to the LLMs, requiring alignment capability of LLMs between the given code snippet and its corresponding bug type. As a result, some open-source models with smaller parameter sizes perform poorly in this task

4 Further Analysis

Performance across Different Error Types. In Figure 7, The performance of models on the automated program repair task varies across different error types, highlighting the strengths and weaknesses of these models in addressing specific challenges. Consistently, the models demonstrate robust capabilities in repairing syntax errors, reference errors, and logic errors. These error types tend to be more straightforward and well-defined, allowing the models to leverage their knowledge effectively to identify and correct issues with high accuracy. In contrast, the models exhibit their worst performance when dealing with language-specific errors. Language-specific errors can arise from unique syntax rules, idiomatic expressions, or even cultural programming practices that are not universally applicable. As a result, addressing these types of errors presents a significant challenge and underscores the need for further improvements in model training.

Other APR Settings In Figure 6, we explore two additional automated program repair settings that aim to simulate realistic user queries in software debugging. Part (a) presents the results for the scenario in which models are given both buggy code and corresponding example test cases. This setup allows for a comprehensive evaluation of the ability of models to understand and correct specific issues based on contextual examples. In contrast, Part (b) illustrates the results for a more challenging scenario where only the buggy code is provided to the models, requiring them to identify and rectify errors without any additional context. This comparison highlights the varying capabilities of models in different settings, emphasizing the importance of context in automated program repair.

Anslysis of Code Review task Besides automated program repair tasks, code review tasks also play a crucial role in software development. To analyze the performance of different models on code review tasks, we conducted experiments based on MDEVAL. For the code review task, we present two versions of code to LLMs: the correct code

Model	Size	Avg$_{all}$	C	C#	CLISP	CPP	F#	Go	HTML	JS	Java	Json	Julia	MD	PHP	Pascal	Python	R	Ruby	Rust	Scala	Swift
Closed-Source Models																						
o1-preview	🔒	37.0	34.3	37.3	18.3	36.2	38.5	47.0	25.9	52.0	31.7	13.6	40.3	28.3	33.3	32.3	52.5	41.7	38.1	60.0	35.9	31.7
o1-mini	🔒	32.8	32.9	29.9	25.0	30.4	23.1	38.6	27.6	53.3	30.0	13.6	28.4	23.3	29.0	29.2	49.2	35.0	34.9	55.7	29.7	28.3
GPT-4o-240806	🔒	24.2	30.0	25.0	15.0	25.8	12.3	39.8	24.1	44.0	20.0	8.5	14.9	23.3	21.7	13.8	45.9	21.7	30.2	31.4	14.1	13.3
GPT-4o-mini-240718	🔒	20.9	21.4	18.8	11.7	21.2	16.9	25.3	19.0	29.3	23.3	10.2	11.9	26.7	24.6	12.3	29.5	38.3	22.2	27.1	9.4	16.7
Claude-3.5-sonnet-240620	🔒	31.7	44.3	26.6	20.0	24.2	26.2	44.6	19.0	45.3	23.3	13.6	35.8	25.0	33.3	33.8	45.9	38.3	34.9	30.0	29.7	30.0
Claude-3.5-sonnet-241022	🔒	33.1	37.1	28.1	16.7	30.3	29.2	37.3	22.4	45.3	23.3	8.5	38.8	25.0	33.3	43.1	55.7	40.0	36.5	38.6	34.4	30.0
1B+ Models																						
Qwen2.5-Instruct	1.5B	2.0	1.4	4.5	1.7	4.3	1.5	0.0	5.2	1.3	1.7	0.0	1.5	5.0	1.4	0.0	4.9	0.0	1.6	4.3	0.0	0.0
OpenCoder-Instruct	1.5B	4.2	0.0	0.0	18.3	1.4	0.0	2.4	1.7	1.3	0.0	25.4	1.5	10.0	7.2	1.5	0.0	0.0	1.6	12.9	1.6	0.0
Qwen2.5-Instruct	3B	10.2	10.0	10.4	3.3	5.8	16.9	14.5	5.2	10.7	8.3	3.4	6.0	18.3	8.7	4.6	11.5	15.0	6.3	10.0	14.1	20.0
7B+ Models																						
Qwen2.5-Coder-Instruct	7B	8.4	11.4	4.7	8.3	3.0	6.2	15.7	8.6	14.7	8.3	10.2	7.5	13.8	4.3	1.5	16.4	8.3	11.1	8.6	0.0	3.3
Meta-Llama-3-Instruct	8B	3.0	2.9	4.5	3.3	2.9	0.0	3.6	0.0	8.0	6.7	1.7	0.0	0.0	4.3	0.0	9.8	1.7	0.0	7.1	0.0	1.7
Meta-Llama-3.1-Instruct	8B	5.4	7.1	10.4	3.3	11.6	0.0	7.2	1.7	5.3	3.3	1.7	6.0	3.3	8.7	3.1	9.8	1.7	6.3	4.3	1.6	10.0
Yi-Coder-Chat	9B	8.7	25.7	9.4	5.0	9.1	4.6	10.8	5.2	12.0	11.7	1.7	16.4	6.9	5.8	4.6	14.8	5.0	3.2	5.7	7.8	5.0
20B+ Models																						
Codestral-v0.1	22B	16.2	21.4	19.4	10.0	21.7	6.2	31.3	12.1	20.0	18.3	6.8	16.4	20.0	7.2	10.8	32.8	8.3	20.6	14.3	12.5	6.7
Qwen2.5-Instruct	32B	19.4	28.6	25.4	10.0	23.2	9.2	34.9	12.1	24	18.3	10.2	26.9	30.0	11.6	10.8	32.8	13.3	25.4	14.3	9.4	10.0
Qwen2.5-Coder-Instruct	32B	23.6	30.0	25.0	13.3	31.8	15.4	37.3	22.4	28.0	16.7	11.9	31.3	29.3	18.8	21.5	36.1	36.7	28.6	20.0	4.7	6.7
DS-Coder-Instruct	33B	12.3	14.3	15.6	0.0	15.2	6.2	15.7	12.1	22.7	8.3	6.8	11.9	27.6	10.1	10.8	24.6	6.7	17.5	11.4	4.7	1.7
Qwen2.5-Instruct	72B	17.6	28.6	16.4	13.3	18.8	7.7	25.3	19.0	26.7	15.0	8.5	20.9	28.3	13.0	6.2	26.2	21.7	22.2	12.9	7.8	10.0
DS-Coder-V2.5	21/236B	19.2	21.4	17.2	11.7	16.7	13.8	32.5	19.0	29.3	18.3	5.1	13.4	20.0	8.7	15.4	37.7	15.0	23.8	25.7	17.2	15.0
xDebugGen (Our Method)	7B	2.3	2.9	4.7	3.3	4.5	0.0	1.2	6.9	8.0	0.0	0.0	0.0	0.0	1.4	3.1	4.9	0.0	0.0	2.9	0.0	1.7

Table 4: Accuracy of different models for Bug Identification tasks on MDEVAL. The underlined numbers are the best scores for each language. "Avg$_{all}$" represents the average accuracy of all code languages.

Model	Size	Avg$_{all}$	C	C#	CLISP	CPP	F#	Go	HTML	JS	Java	Json	Julia	MD	PHP	Pascal	Python	R	Ruby	Rust	Scala	Swift
Closed-Source Models																						
o1-preview	🔒	64.9	72.9	50.7	30.0	62.3	60.0	49.4	74.1	72.0	66.7	79.7	74.6	50.0	79.7	55.4	86.9	71.7	66.7	54.3	71.9	73.3
o1-mini	🔒	68.1	78.6	65.7	41.7	63.8	67.7	47.0	69.0	81.3	71.7	67.8	76.1	53.3	84.1	60.0	78.7	78.3	79.4	60.0	73.4	66.7
GPT-4o-240806	🔒	56.1	60.0	53.1	30.0	54.5	61.5	47.0	65.5	65.3	60.0	62.7	50.7	46.7	58.0	53.8	63.9	61.7	66.7	55.7	57.8	48.3
GPT-4o-mini-240718	🔒	36.8	51.4	28.1	23.3	25.8	32.3	37.3	58.6	54.7	36.7	45.8	20.9	40.0	30.4	26.2	44.3	36.7	50.8	37.1	23.4	31.7
Claude-3.5-sonnet-240620	🔒	62.9	74.3	62.5	61.7	60.6	50.8	59.0	67.2	73.3	63.3	69.5	71.6	55.0	60.9	58.5	68.9	68.3	58.7	58.6	57.8	56.7
Claude-3.5-sonnet-241022	🔒	64.2	67.1	60.9	58.3	59.1	55.4	59.0	69.0	73.3	56.7	72.9	73.1	65.0	60.9	66.2	68.9	76.7	55.6	67.1	57.8	61.7
1B+ Models																						
Qwen2.5-Instruct	1.5B	22.8	22.9	40.3	13.3	21.7	9.2	26.5	8.6	26.7	36.7	16.9	34.3	21.7	15.9	21.5	19.7	30.0	12.7	24.3	15.6	33.3
OpenCoder-Instruct	1.5B	10.5	1.4	17.9	10.0	5.8	6.2	16.9	5.2	25.3	13.3	8.5	10.4	8.3	14.5	15.4	6.6	1.7	6.3	14.3	7.8	8.3
Qwen2.5-Instruct	3B	21.4	30.0	14.9	16.7	13.0	23.1	18.1	29.3	21.3	31.7	27.1	19.4	21.7	20.3	20.0	36.1	26.7	17.5	12.9	23.4	8.3
7B+ Models																						
Qwen2.5-Coder-Instruct	7B	26.8	42.9	21.9	16.7	27.3	23.1	31.3	8.6	32.0	33.3	32.2	25.4	10.3	23.2	32.3	41.0	18.3	36.5	28.6	31.2	13.3
Meta-Llama-3-Instruct	8B	7.8	8.6	6.0	3.3	5.8	1.5	10.8	12.1	10.7	11.7	18.6	6.0	18.3	4.3	3.1	3.3	1.7	6.3	11.4	10.9	1.7
Meta-Llama-3.1-Instruct	8B	7.0	7.1	9.0	10.0	5.8	3.1	12.0	17.2	0.0	6.7	27.1	4.5	13.3	2.9	1.5	3.3	3.3	0.0	8.6	7.8	0.0
Yi-Coder-Chat	9B	29.5	42.9	40.6	20.0	34.8	29.2	22.9	0.0	60.0	40.0	18.6	28.4	1.7	42.0	30.8	6.6	26.7	36.5	35.7	25.0	33.3
20B+ Models																						
Codestral-v0.1	22B	43.6	52.9	41.8	35.0	44.9	43.1	42.2	32.8	62.7	48.3	32.2	53.7	20.0	50.7	43.1	60.7	40.0	47.6	38.6	42.2	31.7
Qwen2.5-Instruct	32B	58.4	68.6	50.7	33.3	65.2	55.4	49.4	75.9	65.3	60.0	66.1	68.7	53.3	56.5	50.8	62.3	68.3	61.9	58.6	51.6	46.7
Qwen2.5-Coder-Instruct	32B	59.4	81.4	50.0	46.7	65.2	64.6	63.9	13.8	72.0	68.3	52.5	68.7	12.1	66.7	55.4	73.8	85.0	63.5	60.0	59.4	50.0
DS-Coder-Instruct	33B	17.8	28.6	12.5	8.3	19.7	16.9	20.5	1.7	38.7	33.3	8.5	20.9	3.4	13.0	12.3	1.6	10.0	27.0	31.4	15.6	21.7
Qwen2.5-Instruct	72B	57.4	74.3	44.8	43.3	44.9	66.2	55.4	65.5	64.0	68.3	61.0	67.2	45.0	50.7	47.7	68.9	61.7	60.3	50.0	59.4	50.0
DS-Coder-V2.5	21/236B	52.3	75.7	50.0	28.3	56.1	40.0	54.2	63.8	64.0	68.3	59.3	44.8	43.3	58.0	27.7	70.5	53.3	34.9	47.1	57.8	46.7
xDebugGen (Our Method)	7B	18.7	30.0	21.9	3.3	15.2	10.8	20.5	8.6	30.7	36.7	28.8	23.9	8.6	18.8	15.4	13.1	5.0	33.3	20.0	15.6	6.7

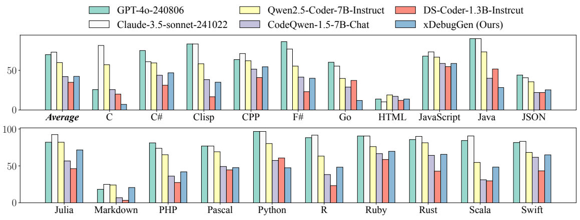
Table 5: Accuracy of different models for Bug Localization tasks on MDEVAL. The underlined numbers are the best scores for each language. "Avg$_{all}$" represents the average scores of all code languages.

b^{L_k} and the buggy code c^{L_k} with only a few minor differences between them. The correct code and buggy code are listed in a random order to feed into LLM for distinguishing the buggy code. Figure 8 displays the accuracy for code review tasks. The results show that closed-source models still significantly outperform open-source models in the code review task. The closed-source models demonstrate a strong ability to understand complex code logic, achieving an accuracy rate of approximately 90%. In contrast, the smaller open-source model exhibits significant challenges, with an accuracy rate of around 50%. This disparity underscores the limitations of the current open-source model in effectively interpreting intricate coding patterns.

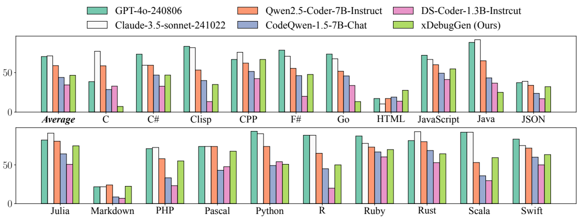
Effect of Bug Location for APR In previous studies, bug localization has been regarded as the first step in program repair, playing a critical role. To verify whether the bug location information can also have a positive impact when using large language models for automated program repair, we designed and conducted a series of comparative experiments, as shown in Figure 9. We test two scenarios: providing the bug location information and not providing it and task the model with repairing buggy code in both cases. The results indicate that providing the bug location information significantly improves Pass@1 scores of automated program repair. However, our prior experimental results reveal that for LLMs, the difficulty of the bug localization task is notably higher than that of the automated program repair task. Therefore, improving the bug localization capabilities of the model is essential for enhancing its overall automated program repair performance.

5 Related Work

The rapid progress of large language models(OpenAI, 2023; Touvron et al., 2023b; AI, 2024; Bai et al., 2023; Yang et al., 2024a) has enabled complex code-related tasks. Early models like BERT(Devlin et al., 2019) and GPT (Radford et al., 2018), trained on billions of code snippets, focused on code understanding and generation (Chen et al.,



(a) Buggy code with example test cases



(b) Only buggy code

Figure 6: Two additional automated program repair settings are designed to simulate realistic user queries. Part (a) presents results for the scenario where models are provided with buggy code along with example test cases, while Part (b) illustrates results for the scenario where only the buggy code is provided to the models.

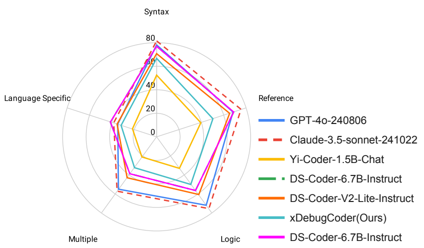


Figure 7: Performance of models on the automated program repair task across error types.

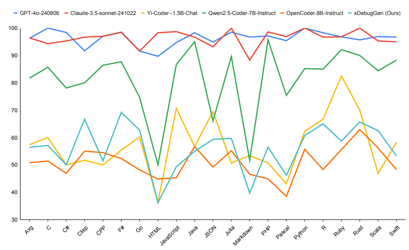


Figure 8: Accuracy of different models for Code Review tasks on MDEVAL.

2021; Feng et al., 2020; Scao et al., 2022; Li et al., 2022; Wang et al., 2021; Allal et al., 2023). Recent advances in domain-specific pre-training and instruction fine-tuning (Zheng et al., 2024a; Yue et al., 2024) have enhanced models like CodeLlama (Rozière et al., 2023) and WizardCoder (Luo et al., 2023), achieving strong performance in code completion, synthesis, and repair.

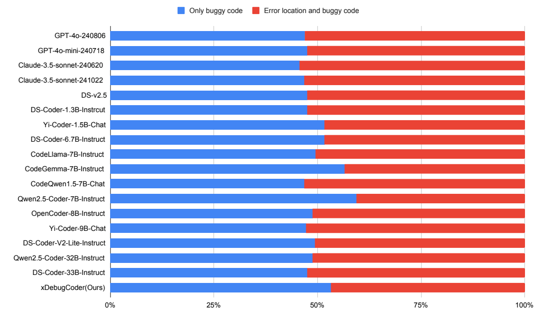


Figure 9: Comparison of the Pass@1 (%) scores with only the buggy code provided versus when additional bug location information is supplied.

LLMs have also gained popularity for automatic program debugging, a critical task for bug detection, vulnerability identification (Pradel and Sen, 2018; Allamanis et al., 2021), fuzz testing (Deng et al., 2023; Xia et al., 2024), and program repair (Wen et al., 2024; Gu et al., 2024). Benchmark tests, such as DebugBench (Tian et al., 2024) and DebugEval (Yang et al., 2024c), assess LLM debugging capabilities across error categories and tasks. However, these focus on 1-3 languages, neglecting language-specific errors. To fill this gap, we propose MDEVAL, a comprehensive debugging benchmark for 20 languages to evaluate LLM performance from a broader perspective.

6 Conclusion

In this work, we introduce MDEVAL of instruction corpora MDEVAL-INSTRUCT, evaluation benchmark, and a strong baseline xDebugCoder, where the benchmark includes automated program repair (APR), bug localization (BL), and bug identification (BI) of 20 programming languages (total 3.9K samples), aiming to assess the debugging capabilities of large language models (LLMs) in multilingual environments. Further, we propose xDebugGen to construct a multilingual debugging instruction corpus, where we inject the bugs into the query or answer to create the pair of the buggy code and correct code. Based on MDEVAL-INSTRUCT, we develop xDebugCoder, a multilingual LLM for debugging in a wide range of programming languages as a strong baseline. Through extensive experiments, this paper reveals a substantial performance gap between open-source and closed-source LLMs, underscoring the need for further improvements in multilingual code debugging. In the future, we will continue expanding the number of languages in MDEVAL.

Limitations

Language Coverage. Although MDEVAL covers 20 programming languages, there are still many languages not included, particularly those that are less commonly used or have niche applications. Expanding the benchmark to include more languages would provide a more comprehensive evaluation of multilingual debugging capabilities.

Real-world Applicability. While MDEVAL aims to simulate realistic debugging scenarios, the tasks and data may not fully capture the complexity and variability of real-world software development. Incorporating more diverse and complex real-world projects into the benchmark could improve its applicability and relevance.

Instruction Tuning Data. The instruction corpora MDEVAL-INSTRUCT used for fine-tuning the baseline model xDebugCoder is generated by LLM-based bug injection. While this approach has shown promise, the quality and diversity of the generated data could be further improved. Exploring alternative methods for generating high-quality instruction data, such as leveraging more advanced LLMs or incorporating feedback from real-world debugging sessions, could enhance the effectiveness of the instruction tuning process.

Ethical Considerations

Potential Risks

MDEVAL, as evaluation tools, can comprehensively assess the capability of large language models in debugging tasks across a wide range of programming languages, thereby advancing the development of large language models in this domain. However, improper or erroneous use of MDEVAL may pose significant risks, such as incorrect program analysis and faulty program repair, which could even lead to severe consequences such as program crashes or operating system failures. Therefore, to ensure the security and reliability of the evaluation process, we strongly recommend using MDEVAL within a sandbox environment. Such an environment can effectively isolate potential system risks, ensuring the accuracy and safety of the evaluation.

References

Meta AI. 2024. Introducing meta llama 3: The most capable openly available llm to date. https://ai.meta.com/blog/meta-llama-3/.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: Don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876.

Anthropic. 2023. Introducing Claude.

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

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 arXiv:2309.16609*, abs/2309.16609.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*.

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. *arXiv preprint arXiv:2406.01304*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, abs/2107.03374.

Ken Deng, Jiaheng Liu, He Zhu, Congnan Liu, Jingxin Li, Jiakai Wang, Peng Zhao, Chenchen Zhang, Yanan Wu, Xueqiao Yin, et al. 2024. R2c2-coder: Enhancing and benchmarking real-world repository-level code completion abilities of code large language models. *arXiv preprint arXiv:2406.01359*.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Google Gemma Team. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*.

Alex Gu, Wen-Ding Li, Naman Jain, Theo X Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. 2024. The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? *arXiv preprint arXiv:2402.19475*.

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

Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi Li, Ruibo Liu, Yue Wang, et al. 2024b. Codeeditorbench: Evaluating code editing capability of large language models. *arXiv preprint arXiv:2404.03543*.

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.

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

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

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*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, abs/2305.06161.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal

Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, abs/2203.07814.

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

Jiaheng Liu, Ken Deng, Congnan Liu, Jian Yang, Shukai Liu, He Zhu, Peng Zhao, Linzheng Chai, Yanan Wu, Ke Jin, Ge Zhang, Zekun Moore Wang, Guoan Zhang, Bangyu Xiang, Wenbo Su, and Bo Zheng. 2024. M2rc-eval: Massively multilingual repository-level code completion evaluation.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

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 arXiv: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 arXiv:2306.08568*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. OctoPack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, abs/2308.07124.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25.

Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can openai's codex fix bugs? an evaluation on quixbugs. In *Proceedings of the Third International Workshop on Automated Program Repair*, pages 69–75.

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

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. *OpenAI blog*.

Baptiste Rozière, 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 arXiv:2308.12950*.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 23–30. IEEE.

Tao Sun, Linzheng Chai, Jian Yang, Yuwei Yin, Hongcheng Guo, Jiaheng Liu, Bing Wang, Liqun Yang, and Zhoujun Li. 2024. UniCoder: Scaling code large language model via universal code. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1812–1824, Bangkok, Thailand. Association for Computational Linguistics.

Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. 2024. Bugs in large language models generated code: An empirical study. *CoRR*.

Wei Tao, Yucheng Zhou, Wenqiang Zhang, and Yu Cheng. 2024. Magis: Llm-based multi-agent framework for github issue resolution. *arXiv preprint arXiv:2403.17927*.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023a. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

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

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. 2024a. Mac-sql: A multi-agent collaborative framework for text-to-sql. *arXiv preprint arXiv:2312.11242*.

Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2023. Intervenor: Prompt the coding ability of large language models with the interactive chain of repairing. *arXiv preprint arXiv:2311.09868*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2024b. Where do large language models fail when generating code? *arXiv preprint arXiv:2406.08731*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, abs/2312.02120.

Hao Wen, Yueheng Zhu, Chao Liu, Xiaoxue Ren, Weiwei Du, and Meng Yan. 2024. Fixing code generation errors for large language models. *arXiv preprint arXiv:2409.00676*.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024a. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Liqun Yang, Jian Yang, Chaoren Wei, Guanglin Niu, Ge Zhang, Yunli Wang, Linzheng ChaI, Wanxu Xia, Hongcheng Guo, Shun Zhang, et al. 2024b. Fuzzcoder: Byte-level fuzzing test via large language model. *arXiv preprint arXiv:2409.01944*.

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

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *International conference on machine learning*, pages 11941–11952. PMLR.

Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240*.

Xiang Yue, Tuney Zheng, Ge Zhang, and Wenhu Chen. 2024. Mammoth2: Scaling instructions from the web. *arXiv preprint arXiv:2405.03548*.

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

Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A critical review of large language model on software engineering: An example from chatgpt and automated program repair. *arXiv preprint arXiv:2310.08879*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*, abs/2303.17568.

Tianyu Zheng, Shuyue Guo, Xingwei Qu, Jiawei Guo, Weixu Zhang, Xinrun Du, Chenghua Lin, Wenhao Huang, Wenhu Chen, Jie Fu, et al. 2024a. Kun: Answer polishment for chinese self-alignment with instruction back-translation. *arXiv preprint arXiv:2401.06477*.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024b. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*.

Zhiyuan Zhong, Sinan Wang, Hailong Wang, Shaojin Wen, Hao Guan, Yida Tao, and Yepang Liu. 2024. Advancing bug detection in fastjson2 with large language models driven unit test generation. *arXiv preprint arXiv:2410.09414*.

A Human Annotation

To construct the massively multilingual code debugging benchmark MDEVAL, we designed and implemented a comprehensive and systematic human annotation process to ensure the accuracy, consistency, and high quality of multilingual code samples. This process strictly adheres to carefully formulated annotation guidelines and incorporates multiple quality control mechanisms.

We recruited 13 computer science graduates as multilingual debugging annotators, all of whom are proficient in at least one programming language and possess a solid foundation in computer science. Prior to the formal annotation process, annotators underwent systematic training on annotation methods, covering core tasks such as problem definition, solution design, and buggy code generation.

Our annotation training guidelines focus on the following key aspects:

- **Standardized Format**: We provide detailed annotation examples and templates for 20 programming languages. Annotators must strictly adhere to a standardized format throughout the annotation process to ensure data consistency and reusability.

- **Accessibility**: All annotation reference data are sourced from open-source materials that allow free use and distribution, ensuring compliance with academic research purposes and relevant legal and ethical requirements.

- **Difficulty Classification**: We establish a detailed difficulty classification guideline for each programming language. Annotators must categorize each problem according to complexity, error type, and problem scale, assigning an appropriate difficulty level (e.g., easy, middle, hard) following the guidelines.

- **Self-Containment**: Annotators must ensure that each problem description is complete and unambiguous, containing all necessary information for problem-solving. Provided example inputs and outputs must be accurate, the generated buggy code must be ensured to fail execution correctly, and the reference solution must pass all test cases. Additionally, test cases should comprehensively cover various boundary conditions and exceptional scenarios.

To maintain annotation quality and incentivize annotators, we offered a compensation of approximately \$6 per problem. Moreover, we provided annotators with a comfortable working environment, free meals, souvenirs, and high-performance computing equipment. A total of approximately 1,300 problems were annotated, with additional annotators hired for quality inspection, leading to a total cost of around \$5,000. Quality inspection tasks included bug identification, bug localization, and code review.

A.1 Quality Control

To ensure the high quality of the MDEVAL, we implemented a rigorous quality control mechanism. First, annotators were required to evaluate the annotated code based on four core criteria: problem difficulty, ambiguity, error type, and solvability. Second, we adopted a dual verification system, where each code snippet was independently annotated by at least two annotators to minimize subjective bias and human errors. In cases of disagreement, resolution was achieved through discussion or by a senior annotator making the final decision.

To further ensure the reliability of the benchmark, we employed three volunteers to assess whether MDEVAL achieved a correctness rate of at least 90% and to correct any errors, thereby guaranteeing the accuracy of the annotations.

B Experiment Detail

xDebugCoder Training Corpora. The training corpora consist of our debugging dataset MDEVAL-INSTRUCT, which contains 16K samples, and the Magicoder-Instruct code generation dataset (Wei et al., 2023), comprising 180K samples. This combination ensures that the model possesses a fundamental capability to follow instructions for basic code tasks. We apply data decontamination before training our xDebugGen. Following Li et al. (2023); Wei et al. (2023), we adopt the N-gram exact match decontamination method with MDEVAL, HumanEval (Chen et al., 2021), MultiPL-E (Cassano et al., 2023), MBPP (Austin et al., 2021).

xDebugCoder Optimization. Our model, xDebugCoder, based on Qwen2.5-Coder-7B, is trained for 3 epochs using a cosine scheduler, starting at a learning rate of 5×10^{-5} with 3% of total training steps for warmup. We utilize AdamW (Loshchilov and Hutter, 2017) as the optimizer; the batch size is set to 1024, with a maximum sequence length of 2048. All experiments are performed with 8 NVIDIA A800-80GB GPUs.

Code LLMs. We evaluate 40 popular models, both closed-source and open-source (sizes ranging from 1.3B to 605B parameters). For general models, we evaluate GPTs (OpenAI, 2023) (GPT4-o, GPT4-o-mini), Claude-3.5 (Anthropic, 2023). For code models, we test Qwen2.5-Coder (Hui et al., 2024), DeepSeekCoder (DS-Coder) (Guo et al., 2024a), CodeLlama (Rozière et al., 2023), and Codegemma (Gemma Team, 2024). Furthermore, we fine-tune the Qwen2.5-Coder-7B to provide a baseline model xDebugCoder for reference. For closed-source models, the responses are generated by the official API. For the open-source models, we perform inference on all models using the vLLM (Kwon et al., 2023) framework. All models adopt a greedy decoding strategy during inference, the temperature is set to 0, and the maximum generation length is 4096.

C Related Work

Code Large Language Model. With the rapid advancement of large language models(LLMs) (OpenAI, 2023; Touvron et al., 2023b; AI, 2024; Bai et al., 2023; Yang et al., 2024a), solving complex code-related tasks has become increasingly feasible, leading to the emergence of numerous Code LLMs. Early studies utilized models like BERT (Devlin et al., 2019) or GPT (Radford et al., 2018) as backbones, trained on billions of code snippets to enable tasks involving code understanding and generation (Chen et al., 2021; Feng et al., 2020; Scao et al., 2022; Li et al., 2022; Wang et al., 2021; Allal et al., 2023). Recently, advancements in domain-specific pre-training and instruction fine-tuning techniques (Zheng et al., 2024a; Yue et al., 2024) have led to extensive efforts in fine-tuning models on large-scale code corpora and crafting code-related task instructions (Rozière et al., 2023; Zheng et al., 2023; Luo et al., 2023; Muennighoff et al., 2023; Gemma Team, 2024; Zheng et al., 2024b; Guo et al., 2024a; Wei et al., 2023; Sun et al., 2024; Lozhkov et al., 2024; Jiang et al., 2023; Hui et al., 2024; Wang et al., 2024a; Deng et al., 2024; Liu et al., 2024). These models demonstrate remarkable performance in tasks like code completion, synthesis, and program repair.

Debugging with Large Language Models. Automatic program debugging holds substantial practical value. With the emergence of LLM capabilities, a growing number of individuals are utilizing LLMs for code debugging, leading to extensive research in this field. Code Debugging includes serval tasks such as bug or vulnerability detection (Pradel and Sen, 2018; Allamanis et al., 2021; Yuan et al., 2023; Zhang et al., 2024; Zhong et al., 2024), fuzz test (Deng et al., 2023; Xia et al., 2024; Yang et al., 2024b), program repair (Wen et al., 2024; Lin et al., 2017; Zhang et al., 2023; Prenner and Robbes, 2023; Gu et al., 2024; Tambon et al., 2024; Wang et al., 2024b), GitHub issues auto resolving (Jimenez et al., 2023; Chen et al., 2024; Tao et al., 2024). To effectively assess the code debugging capabilities of LLMs, several benchmark tests have been introduced (Prenner et al., 2022; Sobania et al., 2023; Xia and Zhang, 2023; Zhang et al., 2023; Tian et al., 2024; Yang et al., 2024c). Notably, DebugBench (Tian et al., 2024) provides a comprehensive classification of error types and analyzes the debugging capabilities of LLMs based on these categories. Similarly, DebugEval (Yang et al., 2024c) has designed various debugging-related tasks to evaluate LLM performance across different task dimensions. However, these studies focus on 1 to 3 languages. In reality, there are significant differences in code errors between languages, leading to numerous language-specific errors. To address this gap, we propose MdEval, a comprehensive code debugging benchmark covering 20 languages, aiming to assess LLM debugging capabilities from a broader perspective.