FullStack Bench: Evaluating LLMs as Full Stack Coders

Bytedance, Seed Foundation Code Team

Abstract

As the capabilities of code large language models (LLMs) continue to expand, their applications across diverse code intelligence domains are rapidly increasing. However, most existing datasets only evaluate limited application domains. To address this gap, we have developed a comprehensive code evaluation dataset FullStack Bench[1,2] focusing on full-stack programming, which encompasses a wide range of application domains (e.g., basic programming, data analysis, software engineering, mathematics, and machine learning). Besides, to assess multilingual programming capabilities, in FullStack Bench, we design real-world instructions and corresponding unit test cases from 16 widely-used programming languages to reflect real-world usage scenarios rather than simple translations. Moreover, we also release an effective code sandbox execution tool (i.e., SandboxFusion[3]) supporting various programming languages and packages to evaluate the performance of our FullStack Bench efficiently. Comprehensive experimental results on our FullStack Bench demonstrate the necessity and effectiveness of our FullStack Bench and SandboxFusion.

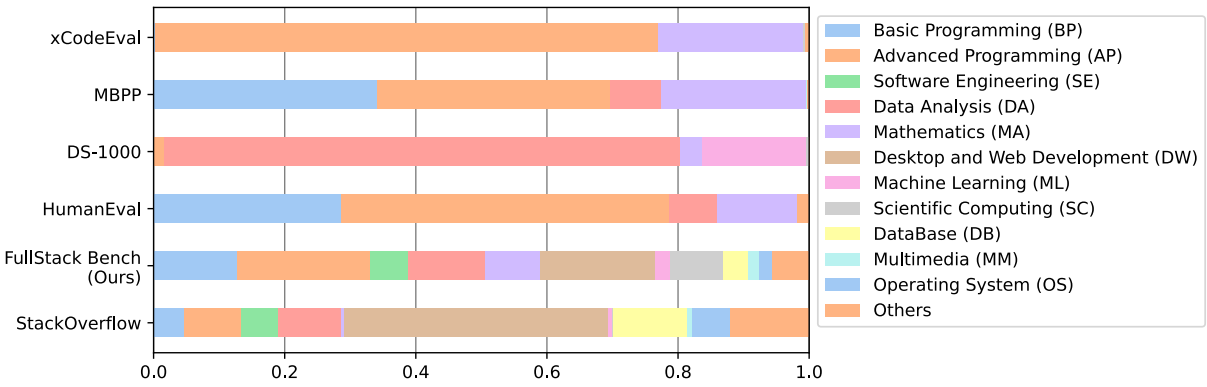


Figure 1. Application domain distributions of different code evaluation datasets.

Author contributions listed at end of paper.

[1] https://huggingface.co/datasets/ByteDance/FullStackBench

[2] https://github.com/bytedance/FullStackBench

[3] https://github.com/bytedance/SandboxFusion

Contents

1. Introduction

The code large language models (LLMs) have achieved significant improvements in code intelligence [Roziere et al., 2023, Zheng et al., 2023, Guo et al., 2024a, Hui et al., 2024, Huang et al., 2024b], which are pre-trained on extensive datasets comprising billions of code-related tokens. Recently, to discover the limitations of existing code LLMs and facilitate further development of code intelligence, many code evaluation benchmark datasets (e.g., HumanEval [Chen et al., 2021a], MBPP [Austin et al., 2021b], DS-1000 [Lai et al., 2022], xCodeEval [Khan et al., 2023]) have been proposed as shown in Figure 1.

However, as shown in Figure 1, we observe that **the existing benchmarks cover limited application domain types, which cannot access the code-related abilities of the real-world code development scenarios**. Specifically, in Figure 1, we sample 500k questions from the widely-used software development community (i.e., "StackOverflow") and tag the application domain label for these questions based on LLMs[4]. Then, based on the labels on "StackOverflow", we summarize 11 main-stream application domains (e.g., Basic Programming, Software Engineering, Data Analysis), which cover about 88.1% problems in "StackOverflow". Meanwhile, using these domain labels, we also tag four popular code evaluation datasets (i.e., HumanEval, MBPP, DS-1000, xCodeEval), and observe that these benchmarks usually focus on very limited domains. For example, a large portion of DS-1000 (>95%) is related to data analysis and machine learning tasks, and even the so-called multi-task benchmark xCodeEval (with code understanding, generation, translation and retrieval tasks) mainly focuses on advanced programming and mathematics domains.

To address the abovementioned limitation, we propose the **FullStack Bench**, an evaluation set spanning multiple computer science domains and programming languages, which aims to assess large models' capabilities across various real-world code development scenarios. As shown in Figure 1, when compared to existing benchmarks, our FullStack Bench covers more application domains, which demonstrates the diversity and necessity of our FullStack Bench. Besides, based on the analysis of StackOverflow, we observe that our FullStack Bench can simulate StackOverflow well for real-world programming scenes, where ratios of the selected 11 application domains (excluding "Others") for our FullStack Bench and StackOverflow are 94.3% and 88.1%, respectively.

Moreover, automating the evaluation on FullStack Bench is challenging due to the various data formats and dependencies for different application domains and programming languages. Recently, some sandbox execution environments (i.e., DifySandbox [LangGenius, 2024], MultiPL-E [Cassano et al., 2023], MPLSandbox [Dou et al., 2024]) have been proposed. However, **there are significant limitations (e.g., supporting limited packages and programming languages) in these sandboxes**, which cannot evaluate our FullStack Bench well. For example, the front-end browsers and deep-learning packages (e.g., PyTorch [Paszke et al., 2019], Tensorflow [Abadi et al., 2015]) are not supported in these sandboxes. Besides, our FullStack Bench has 16 programming languages (i.e., Bash, C++, C#, D, Go, HTML, Java, Javascript, PHP, Python, R, Ruby, Rust, Scala, SQL, Typescript), and many sandboxes do not fully support these languages. Therefore, we also introduce a new execution environment (i.e., **SandboxFusion**) to support the evaluation on our FullStack Bench, and the main features of SandboxFusion are as follows: (1) **Supporting various languages**: our SandboxFusion supports 23 commonly-used programming languages, which satisfies different real-world usage scenes (e.g., front-end development, backend development,

[4]Prompt: *You are an expert in the field of computer programming, proficient in various programming knowledge. Below, I will provide a set of user questions and AI assistant answers. You need to output application domain tags for this Q&A pair. Here are some tags for reference: Mathematics, Data Analysis, Database, Desktop and Web Development.*

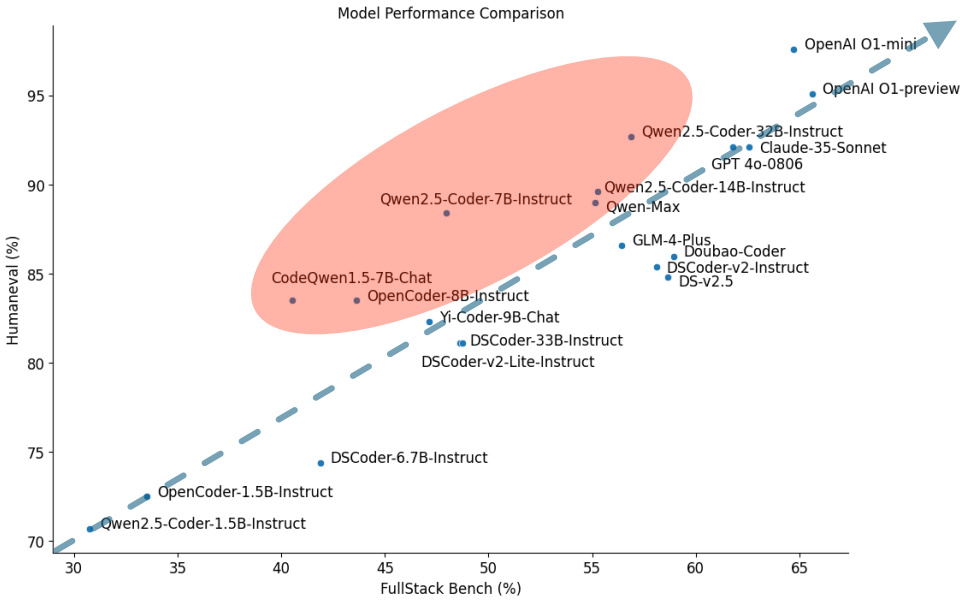


Figure 2. Performance plot of tested LLMs on HumanEval and FullStack Bench.

ML training) (2) **Easy-to-deploy**: we only need a single server to deploy our SandboxFusion with high throughput for large model evaluation scenarios. (3) **Unified multi-dataset execution environment**: Apart from our FullStack Bench, we additionally support 10+ widely-used code evaluation benchmarks.

Overall, the contributions are summarized as follows:

- To access the code-related abilities of the real-world code development scenarios, we propose the FullStack Bench dataset with 3374 problems for 16 programming languages, which covers more mainstream application domains when compared to existing code evaluation benchmarks.
- Meanwhile, to evaluate our FullStack Bench efficiently, we also release an effective code sandbox execution tool (i.e., SandboxFusion) to evaluate different programming tasks from different languages.
- Comprehensive experimental results and detailed analysis demonstrate the necessity and effectiveness of our FullStack Bench and SandboxFusion. Notably, We compared the performance scores on the test set of FullStack Bench with the performances of models on HumanEval in Figure 2. Most models are located in the upper triangular area of the graph, with many models scoring high on Humaneval but exhibiting relatively lower performance on FullStack Bench.

2. FullStack Bench

2.1. Data Overview

As illustrated in Table 1, the FullStack Bench consists of 3374 problems, where each problem in FullStack Bench includes *question, unit test cases, reference solution, and labels*. Besides, we also calculate the token lengths of the question and correct code using the LLaMA3 tokenizer [Team, 2024], where the average question length is 210.2 tokens. To ensure judgment accuracy, the overall number of unit tests for the dataset is 15168, where the average number of unit tests is

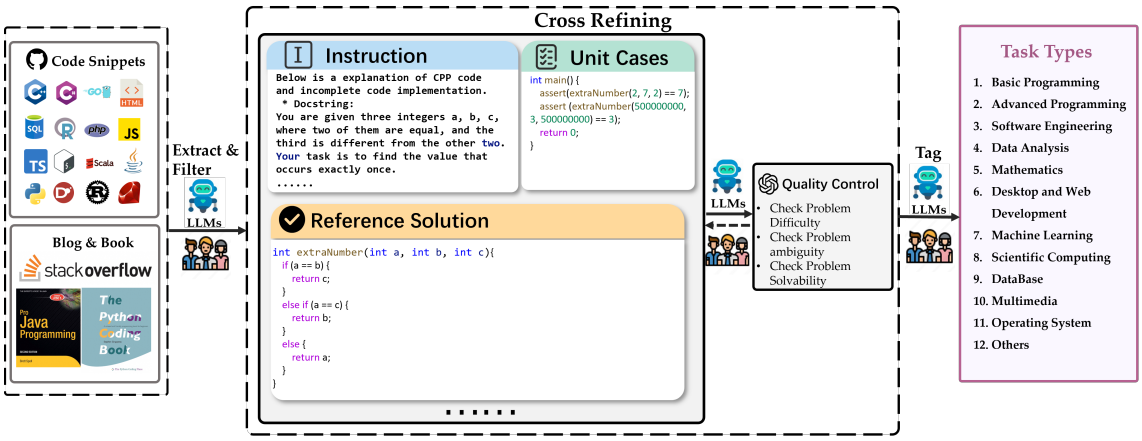


Figure 3. Overview of data collection process of FullStack Bench.

4.5. 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 and Quality Control

To curate the multilingual full stack code evaluation benchmark FullStack Bench, we employ a comprehensive and systematic human annotation process for producing code samples of different application domains, where meticulously pre-defined guidelines are provided to guarantee accuracy and consistency.

Specifically, as shown in Figure 3, we illustrate the overall dataset construction process. Specifically, we first collect code snippets from Github, code-related documents (e.g., blog and Book) and XLCoST [Zhu et al., 2022] . Then, we use LLM and human verification to generate the instruction, unit cases and corresponding reference solution. Besides, we also employ programming experts actively in each field to create domain-specific questions for LLMs. These questions do not involve proprietary information, but are designed to assess essential skills in the respective application domains, similar to interview questions. For example, we engaged our internal data engineering team to develop a series of data analysis questions, including data filtering, data mining, and data visualization. After obtaining the initial dataset, to improve the annotation quality, the annotators evaluate the annotated code based on three criteria: problem difficulty, ambiguity and solveability. 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

Statistics	**Number**
#Problems	3374
Difficulty Level	
- Easy/Medium/Hard	1,466/1,184/724
Length	
Question	
- *maximum length*	1931 tokens
- *minimum length*	35 tokens
- *avg length*	210.2 tokens
Reference Solution	
- *maximum length*	2720 tokens
- *minimum length*	4 tokens
- *avg length*	153.0 tokens

Table 1. Dataset statistics of FullStack Bench.

through consensus or with input from senior annotators.

Additionally, to improve the difficulty of our FullStack Bench, we follow the LIME [Zhu et al., 2024a] and implement a voting method using six selected models (i.e., DeepSeek-Coder-6.7B [Guo et al., 2024a], DeepSeek-Coder-33B [Guo et al., 2024a], Qwen2.5-Coder-7B [Hui et al., 2024], LLaMA3.1-70B [Team, 2024], Claude-3.5-Sonnet [5], GPT-4o [Achiam et al., 2023]) to filter out samples that can be correctly answered by all these LLMs. Specifically, for each question, if only one model obtains the correct answer, this question is classified as a hard sample, and if five or six models obtain the correct answer, this question is classified as an easy sample. Apart from the easy and hard samples, the difficulty of the remained samples is medium.

Moreover, to simulate real-world usage of full-stack developments, we summarize the common application domains by analyzing the distributions of "Stackoverflow.com". As shown in Figure 4, we sample 500k questions from "Stackoverflow.com" and then prompt the LLMs to label the application domain type for each question. After that, we preserve the top 11 application domains, which dominate 88.1% of the questions. Meanwhile, we name other application domain types as "Others". In this way, we also prompt GPT to label the domain types of our annotated questions and generate our final FullStack Bench, where the domain types are as follows:

- **Basic Programming (BP)**: Basic programming involves fundamental concepts and skills to write simple computer programs. This typically includes understanding data types, variables, control structures, functions, and basic input/output operations.
- **Advanced Programming (AP)**: Advanced programming involves developing complex software solutions and focuses on creating efficient, scalable, and robust applications while implementing sophisticated algorithms, data structures, and design patterns.
- **Software Engineering (SE)**: Software engineering covers the design, development, testing, and maintenance of software systems, and includes tasks of requirements analysis, software architecture design, coding, quality assurance, and project management.
- **Data Analysis (DP)**: Data analysis is the cleaning, processing, and analysis of collected data to discover meaningful patterns and relationships to make data-driven decisions.
- **Mathematics (MA)**: Mathematical problems involve solving various problems through mathematical methods and theories, covering multiple fields such as algebra, geometry, calculus, number theory, probability, and statistics
- **Desktop and Web Development (DW)**: Desktop development encompasses a wide range of programming languages, frameworks, and tools to design, build, and maintain user-friendly interfaces and robust backend systems.
- **Machine Learning (ML)**: Machine learning algorithms are developed to learn from data for tasks such as classification, prediction, and pattern recognition.
- **Scientific Computing (SC)**: Scientific computing solves complex scientific and engineering problems, which encompasses the tasks of numerical analysis, and high-performance computing to simulate, model, and analyze phenomena across various scientific disciplines.
- **DataBase (DB)**: Database includes tasks such as insertion, querying, updating, and deletion, and these tasks are typically performed using query languages such as SQL to ensure efficient storage and retrieval of data.
- **Multimedia (MM)**: Multimedia involves processing and manipulating various forms of content, including text, images, audio, and video.
- **Operating System (OS)**: Operating system includes tasks such as memory management, process scheduling, file system management, and device control, which aims to manage

[5] https://www.anthropic.com/news/claude-3-5-sonnet

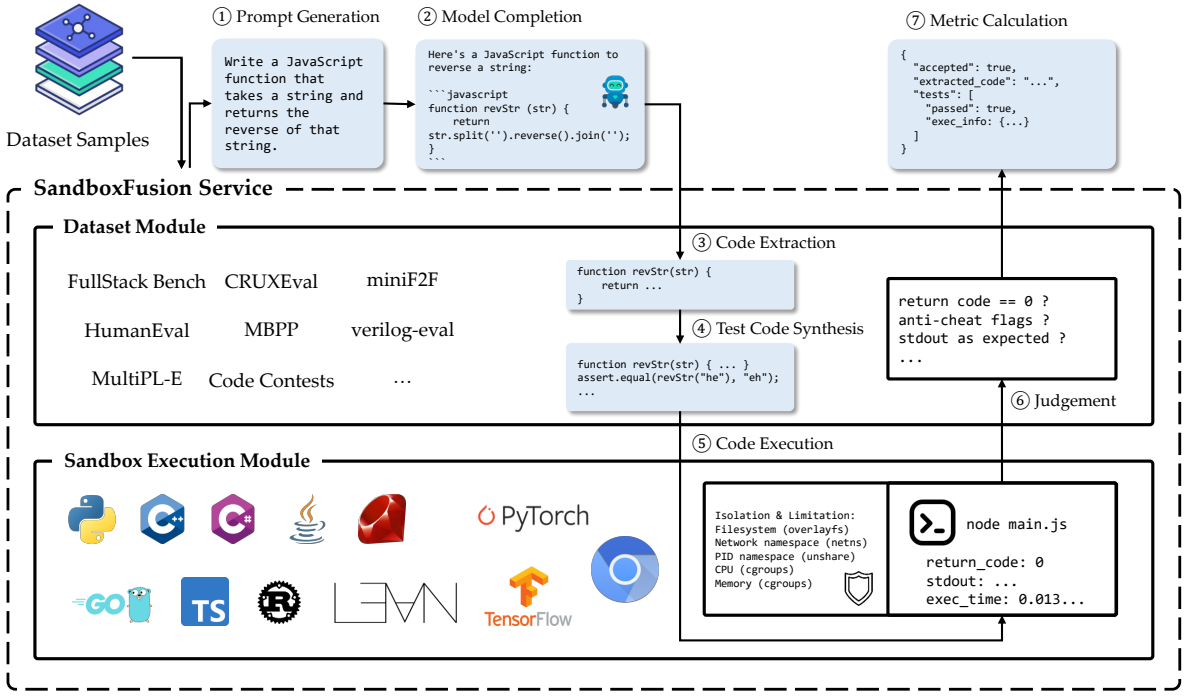


Figure 4. SandboxFusion Architecture.

computer hardware and software resources.

- **Others**: Apart from the above 11 mainstream application domains, other domains are categorized as "Others".

2.3. Bilingual Benchmark Construction

The collected questions are in Chinese or English. For Chinese or English problems, we translate these problems into English or Chinese, which results in both Chinese and English versions. Finally, in FullStack Bench, the numbers of Chinese and English problems are both 3374/2 = 1687.

2.4. Evaluation Metrics

Following HumanEval and MBPP, we directly use the Pass@1 as the default evaluation metric for our proposed FullStack Bench.

3. SandboxFusion

Execution-based datasets are crucial for discriminating code generation tasks [Hendrycks et al., 2021]. Automating the evaluation of these datasets requires extracting complete code from the model's responses, and executing it in a compatible environment. This is a complex task due to the varying data formats and dependencies. To facilitate the evaluation of FullStack Bench, we also propose the SandboxFusion execution environment. SandboxFusion is a unified architecture that is compatible with many datasets as well as Fullstack Bench. This makes the sandbox widely applicable for data processing, model evaluation, reinforcement learning, etc.

As shown in Figure 4, the overall evaluation process of SandboxFusion usually involves the following steps:

- **Prompt Generation**: The system generates diverse prompts based on the original problem specifications and evaluation paradigms (e.g., few-shot, zero-shot), enabling systematic assessment of model capabilities.
- **Model Completion**: Users need to perform model completion using the generated prompts independently, as our framework does not provide built-in inference capabilities. While many efficient inference engines exist (e.g., vLLM, text-generation-inference), we focus on prompt generation and evaluation.
- **Code Extraction**: The system extracts executable code segments from model outputs, primarily focusing on code contained within markdown blocks.
- **Test Code Synthesis**: The framework combines the extracted code with predefined test cases to create executable test programs. This process handles various language-specific requirements, such as distributing classes across files in Java or adapting main functions for unit testing.
- **Code Execution**: The system executes the synthesized code with all dependent files and captures program output.
- **Judgement**: The framework assesses solution correctness based on execution results, typically through standard unit testing frameworks where zero return values indicate successful execution.
- **Metric Calculation**: The evaluation primarily focuses on pass rates across different problem instances.

SandboxFusion mainly contains two modules: the **Dataset Module** and the **Sandbox Execution Module**. The dataset module is responsible for implementing various datasets and abstracting out common components for reuse. The sandbox execution module focuses on executing code in different languages, controlling resource usage, and ensuring execution safety. Please See Appendix A.2 for more details of SandboxFusion and more comparisons with other sandboxes in Appendix A.3.

4. Experiments

4.1. Experimental Setup

FullStack AI Coders. We select 27 popular (code) language models as full-stack AI coders and test them with FullStack Bench. For open-sourced models, we select AI coders from well-known and uprising code LLM series, including CodeQwen1.5 [Bai et al., 2023], Qwen2.5-Coder [Hui et al., 2024], DeepSeek-Coder [Guo et al., 2024b], Deep-Seek-Coder-v2 [Zhu et al., 2024b], CodeLlama [Roziere et al., 2023], Yi-Coder [Young et al., 2024], StarCoder2 [Lozhkov et al., 2024], and OpenCoder [Huang et al., 2024a]. Further, we involve two open-sourced general LLMs, Qwen2.5 [6] and Llama3.1 [Team, 2024], into the comparison. As the majority of problems in FullStack Bench are complex natural language instructions, we adopt the instruction-tuned version of those AI coders rather than their base models. According to the model size, we categorize the AI coders into five groups: 1B+, 6B+, 13B, 20B+, and 70B+.

On the other hand, we also evaluate some prominent close-sourced LLMs including GPT-4o, OpenAI-o1, Claude, GLM4, DeepSeek-v2.5, Qwen-Max, and the upcoming Doubao-Coder-Preview. The access links of the open-sourced and close-sourced models are listed in Table 5 and Table 6, respectively.

[6] https://qwenlm.github.io/blog/qwen2.5/

Model	BP	AP	SE	DP	MA	DW	ML	SC	DB	MM	OS	Others	Overall
1B+ Instruction Tuned Coder													
OpenCoder-1.5B-Instruct	26.05	40.03	31.50	42.64	25.17	39.12	23.75	13.97	30.16	26.67	44.12	38.30	**33.52**
Qwen2.5-Coder-1.5B-Instruct	18.37	34.75	29.00	33.50	28.32	41.33	17.50	15.81	40.48	23.33	47.06	28.19	30.74
DeepSeek-Coder-1.3B-Instruct	16.74	29.91	32.50	37.06	22.73	35.54	18.75	9.19	27.78	25.00	36.76	30.32	27.65
6B+ Instruction Tuned Coder													
Qwen2.5-Coder-7B-Instruct	38.60	53.23	39.00	63.20	49.65	44.56	37.50	33.46	46.83	55.00	63.24	54.26	**47.95**
Yi-Coder-9B-Chat	39.07	46.04	39.50	64.97	46.50	49.66	42.50	34.93	48.41	41.67	58.82	49.47	47.13
OpenCoder-8B-Instruct	39.53	49.12	38.00	55.58	36.01	45.92	27.50	26.47	47.62	46.67	45.59	45.74	43.63
DeepSeek-Coder-7B-Instruct-v1.5	38.37	45.16	36.00	57.36	35.66	47.96	30.00	30.88	46.03	53.33	45.59	44.15	43.48
DeepSeek-Coder-6.7B-Instruct	34.19	43.40	38.50	58.12	38.11	43.88	33.75	23.90	46.03	38.33	60.29	44.15	41.88
CodeQwen1.5-7B-Chat	36.74	44.87	46.00	51.78	29.72	40.82	26.25	24.26	42.06	41.67	48.53	44.68	40.52
CodeLlama-7B-Instruct	21.40	21.70	30.50	34.26	20.28	40.48	8.75	11.76	34.92	15.00	50.00	29.26	27.06
13B+ Instruction Tuned Coder													
Qwen2.5-Coder-14B-Instruct	53.26	58.50	41.00	69.54	69.23	46.26	51.25	43.01	49.21	60.00	69.12	57.45	**55.28**
DeepSeekCoder-v2-Lite-Instruct	45.81	57.18	38.50	56.85	52.80	44.56	42.50	33.82	52.38	33.33	50.00	51.60	48.73
StarCoder2-15B-Instruct-v0.1	38.37	42.23	29.00	59.90	37.06	40.99	42.50	28.68	54.76	33.33	42.65	45.74	41.79
CodeLlama-13B-Instruct	24.88	21.41	31.00	31.47	18.18	41.67	16.25	13.24	35.71	15.00	45.59	32.45	27.59
20B+ Instruction Tuned Coder													
DeepSeekCoder-v2-Instruct	52.79	63.64	43.00	71.57	75.87	47.45	46.25	52.94	53.97	51.67	63.24	59.57	**58.09**
Qwen2.5-Coder-32B-Instruct	51.86	60.85	43.00	73.10	69.93	47.11	55.00	44.85	56.35	61.67	61.76	60.64	56.88
DeepSeekCoder-33B-Instruct	38.37	50.59	35.50	65.99	50.00	49.49	43.75	39.71	49.21	53.33	54.41	48.40	48.61
CodeLlama-34B-Instruct	23.72	22.73	26.50	37.56	18.18	43.71	17.50	17.65	38.10	26.67	51.47	30.85	29.22
70B+ Instruction Tuned General Language Model													
Qwen2.5-72B-Instruct	52.56	61.44	43.00	66.50	76.57	48.47	55.00	51.10	52.38	51.67	55.88	55.32	**56.88**
Llama3.1-70B-Instruct	46.51	54.69	34.50	65.48	64.69	45.24	51.25	38.60	56.35	46.67	57.35	53.72	51.45
Close-Sourced API Model													
OpenAI o1-preview	71.63	71.99	49.50	72.59	80.77	51.53	50.00	63.97	57.14	60.00	67.65	68.09	**65.62**
OpenAI o1-mini	70.23	75.66	41.50	71.07	81.47	48.47	56.25	59.19	54.76	60.00	69.12	67.55	64.73
Claude-35-Sonnet	61.63	65.40	53.00	71.83	77.27	48.81	53.75	63.24	58.73	68.33	64.71	68.62	62.57
GPT 4o-0806	57.21	67.60	46.00	74.37	76.92	48.47	63.75	55.88	60.32	63.33	70.59	64.89	61.77
Doubao-Coder-Preview	56.98	64.66	43.00	71.07	74.48	49.15	45.00	59.19	50.00	48.33	60.29	55.32	58.92
DeepSeek-v2.5	51.86	63.78	43.00	69.54	75.17	49.66	47.50	57.35	53.17	60.00	60.29	61.70	58.65
GLM-4-Plus	49.77	59.97	43.00	71.32	72.73	46.43	56.25	50.00	57.14	61.67	63.24	52.66	56.40
Qwen-Max	47.21	61.14	42.00	63.20	72.73	47.11	53.75	55.15	57.94	41.67	54.41	50.53	55.16

Table 2. Model performance across domains.

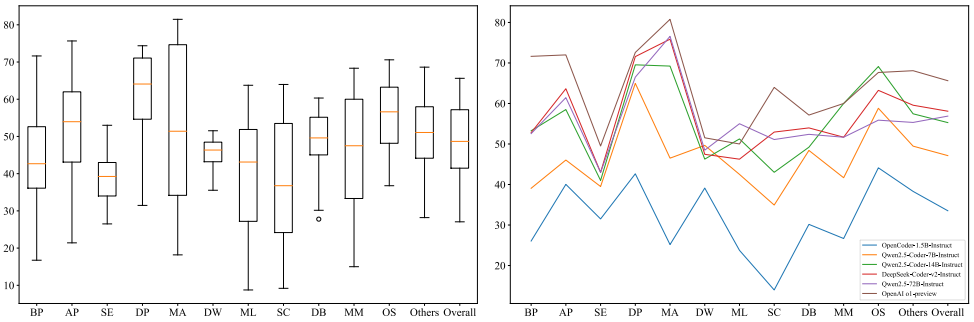
Implementation Details. For open-sourced coders, we pull model checkpoints from Hugging Face[7] and load the model with `vLLM` [Kwon et al., 2023] to accelerate the evaluation. We set *temperature* to 0, *max_completion_tokens* to 2048, and keep all other settings as default. For the input prompt, we set each problem text in FullStack Bench as the user prompt, while leaving the system prompt as default for each model.

We chat with the close-sourced models via API calls similarly, where *temperature* is set to 0 and *max_completion_tokens* is set to 2048. The prompt template is kept the same. After model inference, the first code block with the corresponding programming language formatted in Markdown is extracted from the generated output. If no Markdown code block is detected, a heuristic approach is employed to identify and extract incomplete code snippets. The extracted code, combined with predefined test cases, is then used to synthesize the complete code, which is evaluated for correctness.

4.2. Results and Analysis

We conduct a systematic evaluation of those AI coders with FullStack Bench. Results across the 11+ real-world domains are presented in Table 2. Owing to the powerful reasoning capability, OpenAI o1-preview unsurprisingly leads the board. However, the dominant position of closed-

[7] https://huggingface.co/



(a) General performance (b) Performance of selected models

Figure 5. Visualization on domain performance.

sourced models has been challenged, with some closed-sourced models being matched or even surpassed by pioneers in open-sourced ones. DeepSeekCoder-v2-Instruct, a 236B-MoE model, is the best behavior of open-sourced models, which pulls away the runner-ups in AP, OS, and Others. OpenCoder-1.5B-Instruct, Qwen2.5-Coder-7B-Instruct, and Qwen2.5-Coder-14B-Instruct achieve the top spot in their groups and outperform some models in the closest higher level.

As illustrated in Figure 5, we visualize the model performance on different domains in FullStack Bench. From Figure 5a, we could find that the performance of AI coders varies significantly in BP, AP, MA, ML, SC, and MM. The largest range occurs in MA, with the best mathematician being OpenAI o1-mini (81.47) while the worst is CodeLlama-34B-Instruct (18.18). Mathematical programming requires models to be proficient in both math and code, and those trained on a code-highly-concentrated corpus would struggle to achieve high scores in MA. Similarly, the variances of ML, SC, and MM, are also remarkable, as each of these problems requires domain knowledge beyond coding. Moreover, in Figure 5b, we visualize the first place in each model division. The performance trends of OpenAI-o1-preview, Qwen2.5-72B-Instruct, DeepSeek-Coder-v2-Instruct, and Qwen2.5-Coder-14B-Instruct are generally consistent. OpenAI-o1-Preview has a clear advantage on BP, AP, and SC. On ML and OS, OpenAI-o1-preview only achieves the second place. On the remaining domains, there are slight performance gaps between OpenAI-o1-preview and the runner-ups. Independent of those larger models, the performance trends of OpenCoder-1.5B-Instruct and Qwen2.5-Coder-7B-Instruct are close to each other. These two small models fall behind on BP, AP, MA, and SC, suggesting that reasoning power in these domains may only emerge in the larger models.

4.3. Analysis on the performance of different programming languages

We present results of different programming languages in Table 3 and Figure 6. Except for Java, the performance of AI coders on different languages has significant variance. The performance gaps are relatively larger in C#, D, PHP, Ruby, Rust, and Scala. Besides, as our SandboxFusion provides feedback from the compilers, we evaluate the compilation pass rates of model responses in compiled languages including C++, D, Java, Rust, and Scala. In Figure 7a, there is a positive correlation between the compile pass rate and the test pass rate, but a compile pass does not necessarily mean a test pass. Moreover, from Figure 7b, we found that the written language of the prompts would affect the model performance. We are surprised that DeepSeek-Coder-1.5B-Instruct, proposed by a Chinese institution, is the most favorable model for English questions. On the other hand, some native English speakers, such as StarCoder2-15B-Instruct-v0.1, Open AI-o1-preview, and Open AI-o1-mini, perform better on Chinese questions.

Model	Bash	C++	C#	D	Go	HTML	Java	JS	PHP	Python	R	Ruby	Rust	Scala	SQL	TS	Overall
1B+ Instruction Tuned Coder																	
OpenCoder-1.5B-Instruct	73.33	20.09	50.00	10.87	39.71	51.88	38.65	51.28	58.33	29.27	20.00	42.11	33.33	28.57	35.00	50.00	**33.52**
Qwen2.5-Coder-1.5B-Instruct	86.67	12.62	22.22	6.52	42.65	58.75	37.99	51.28	47.22	29.09	12.50	21.05	16.67	7.14	43.75	57.89	30.74
DeepSeek-Coder-1.3B-Instruct	60.00	10.28	43.94	4.35	30.88	46.88	35.15	52.56	63.89	24.35	5.00	34.21	15.00	7.14	28.75	44.74	27.65
6B+ Instruction Tuned Coder																	
Qwen2.5-Coder-7B-Instruct	93.33	34.58	58.08	23.91	54.41	65.63	43.01	73.08	63.89	47.22	18.75	52.63	56.67	35.71	60.00	68.42	**47.95**
Yi-Coder-9B-Chat	76.67	25.23	75.25	13.04	41.18	67.50	42.58	74.36	55.56	45.62	27.50	63.16	51.67	46.43	56.25	65.79	47.13
OpenCoder-8B-Instruct	66.67	29.91	61.62	30.43	48.53	61.25	41.27	70.51	58.33	40.05	25.00	39.47	50.00	51.79	61.25	60.53	43.63
DeepSeek-Coder-7B-Instruct-v1.5	50.00	24.77	68.18	28.26	54.41	63.13	40.83	66.67	44.44	41.29	21.25	52.63	43.33	39.29	55.00	50.00	43.48
DeepSeek-Coder-6.7B-Instruct	76.67	23.36	56.06	13.04	52.94	65.00	41.48	65.38	50.00	39.34	31.25	47.37	43.33	28.57	57.50	60.53	41.88
CodeQwen1.5-7B-Chat	73.33	30.37	41.92	15.22	48.53	57.50	44.76	60.26	44.44	37.44	15.00	52.63	58.33	46.43	52.50	60.53	40.52
CodeLlama-7B-Instruct	76.67	13.08	43.94	11.96	23.53	54.38	32.97	37.18	27.78	21.98	17.50	21.05	20.00	25.00	41.25	50.00	27.06
13B+ Instruction Tuned Coder																	
Qwen2.5-Coder-14B-Instruct	93.33	39.25	65.66	41.30	63.24	66.25	43.45	80.77	77.78	56.22	32.50	71.05	73.33	42.86	62.50	68.42	**55.28**
DeepSeekCoder-v2-Lite-Instruct	50.00	36.45	52.53	27.17	61.76	64.38	41.27	73.08	58.33	49.35	38.75	55.26	53.33	42.86	57.50	60.53	48.73
StarCoder2-15B-Instruct-v0.1	56.67	21.03	60.61	29.35	47.06	49.38	31.44	70.51	44.44	42.36	28.75	71.05	35.00	32.14	63.75	52.63	41.79
CodeLlama-13B-Instruct	63.33	11.68	50.00	15.22	33.82	55.63	33.84	35.90	36.11	21.33	17.50	42.11	25.00	21.43	38.75	47.37	27.59
20B+ Instruction Tuned Coder																	
DeepSeekCoder-v2-Instruct	83.33	43.46	72.73	28.26	66.18	69.38	42.36	80.77	61.11	60.43	41.25	65.79	68.33	69.64	66.25	68.42	**58.09**
Qwen2.5-Coder-32B-Instruct	83.33	36.92	76.77	46.74	54.41	71.25	40.39	79.49	58.33	59.06	35.00	63.16	76.67	50.00	70.00	57.89	56.88
DeepSeekCoder-33B-Instruct	60.00	26.64	68.18	19.57	57.35	71.25	44.10	66.67	50.00	48.10	32.50	60.53	50.00	46.43	60.00	57.89	48.61
CodeLlama-34B-Instruct	76.67	12.62	47.98	11.96	35.29	63.75	33.62	37.18	44.44	23.34	18.75	36.84	21.67	21.43	48.75	47.37	29.22
70B+ Instruction Tuned General Language Model																	
Qwen2.5-72B-Instruct	80.00	32.24	57.07	33.70	63.24	73.75	41.92	79.49	77.78	61.02	37.50	63.16	61.67	66.07	68.75	68.42	**56.88**
Llama3.1-70B-Instruct	76.67	28.50	72.73	40.22	50.00	77.50	28.38	71.79	41.67	54.50	36.25	65.79	58.33	46.43	68.75	57.89	51.45
Close-Sourced API Model																	
OpenAI o1-preview	90.00	51.40	80.81	55.43	64.71	75.63	42.36	78.21	75.00	68.19	62.50	84.21	86.67	83.93	71.25	78.95	**65.62**
OpenAI o1-mini	90.00	66.36	76.77	53.26	67.65	75.00	34.06	80.77	88.89	67.71	58.75	78.95	83.33	82.14	68.75	68.42	64.73
Claude-35-Sonnet	86.67	41.12	75.76	64.13	50.00	81.88	40.39	70.51	47.22	66.59	46.25	84.21	80.00	83.93	71.25	55.26	62.57
GPT 4o-0806	93.33	50.47	72.73	28.26	67.65	71.25	43.23	83.33	44.44	64.81	52.50	73.68	85.00	73.21	76.25	57.89	61.77
Doubao-Coder-Preview	76.67	40.65	74.24	43.48	52.94	66.25	37.77	84.62	77.78	62.74	45.00	71.05	66.67	75.00	66.25	65.79	58.92
DeepSeek-v2.5	86.67	45.33	74.75	30.43	61.76	76.25	40.83	75.64	91.67	60.90	35.00	68.42	65.00	64.29	66.25	71.05	58.65
GLM-4-Plus	83.33	38.79	74.24	32.61	54.41	75.63	41.70	80.77	80.56	58.29	31.25	52.63	60.00	58.93	72.50	55.26	56.40
Qwen-Max	76.67	32.71	40.40	28.26	61.76	73.75	42.79	85.90	52.78	60.37	35.00	60.53	76.67	46.43	72.50	52.63	55.16

Table 3. Model performance across programming languages.

4.4. Scaling Laws on FullStack Bench

We categorized the model into 5 series based on the criteria in Table 4 of Appendix and visualize the performance of different model series in Figure 8. As the parameter increases, performance gains are achieved for all the model families. We can say that the scaling law still holds, but the improvement in model performance is diminishing as the model size increases.

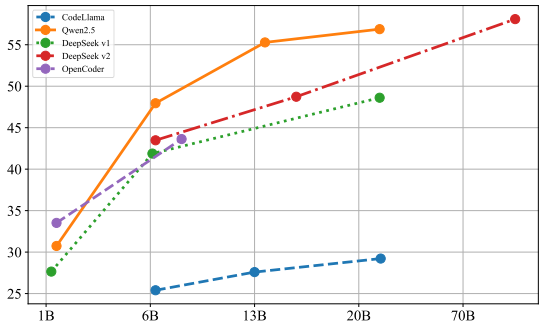


Figure 8. Performance of different sizes.

4.5. Analysis on the performance of different difficulties

Figure 9 presents model performance on different difficulties. As stated in Section 2.2, we implement a voting method with the involvement of six AI coders to determine the difficulty of each problem. Overall, the 1B+ models and CodeLlama series are less effective on all difficulty levels. While the rest of the models could solve simple problems equally well, and gaps appear in the medium questions. As for hard questions, the closed-source models generally outperform the open-source coders.

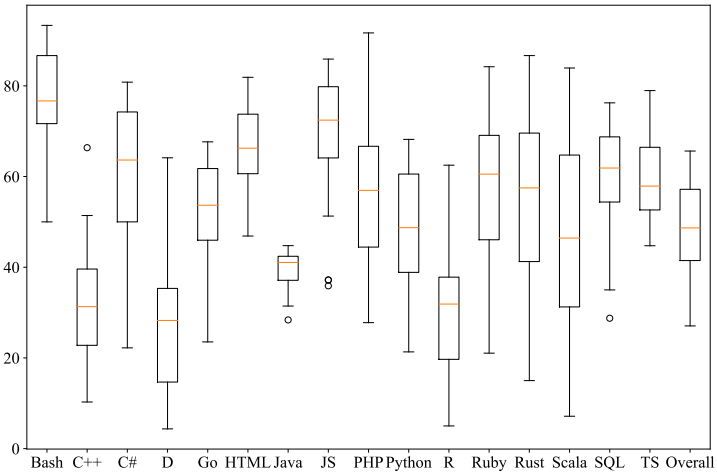
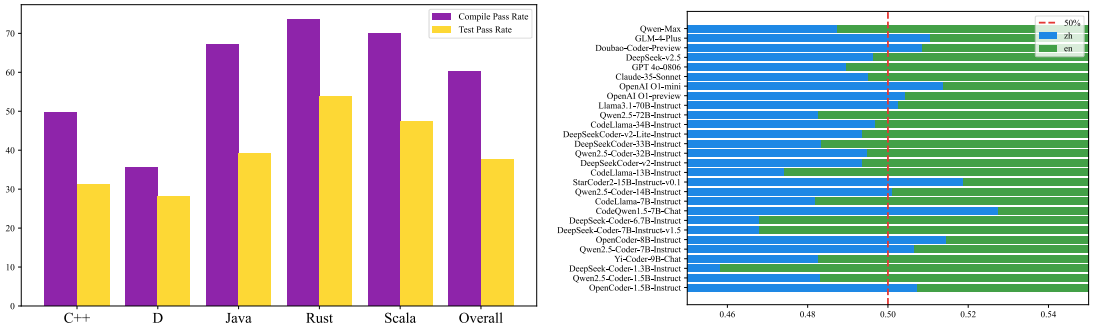


Figure 6. General performance on different programming languages.



(a) Compile Pass@1 and Test Pass@1

(b) Pass@1 in Chinese/English.

Figure 7. Visualization of performance on different languages.

4.6. Analysis on the effect of feedback from SandboxFusion

As shown in Figure 10, to demonstrate the effectiveness of the feedback using SandboxFusion, we compare the "Reflection" and "BoN" strategies. For "Reflection", we reproduce the self-refine strategy [Madaan et al., 2024] by refining the answers of N times using the feedback context of SandboxFusion. For "BoN", we just infer N times to obtain the results. In Figure 10, we observe that the "Reflection" is better than "BoN" a lot, which demonstrates the effectiveness of the feedback context provided by the SandboxFusion.

5. Related Works

Code Large Language Models. Code large language models (LLMs) [Chen et al., 2021b, Zhao et al., 2024, Black et al., 2021, 2022, Le et al., 2022, Chowdhery et al., 2023, Nijkamp et al., 2023, Fried et al., 2023, Xu et al., 2022, Sun et al., 2024, Hui et al., 2024] has shown powerful capabilities in code generation [Li et al., 2022, Allal et al., 2023], code debug, code translation [Zheng et al., 2023, Li et al., 2023], and other coding tasks, which is essential for modern software engineering, For example, there is a wide variety of in-file benchmarks to evaluate different capabilities of code LLMs [Zheng et al., 2023, Austin et al., 2021b, Jain et al., 2024], which focus on a limited range of programming languages (e.g. Python and Java). Further, recent code LLMs such as Code Llama [Roziere et al., 2023], DeepSeek-Coder [Guo et al., 2024a], and Qwen2.5-

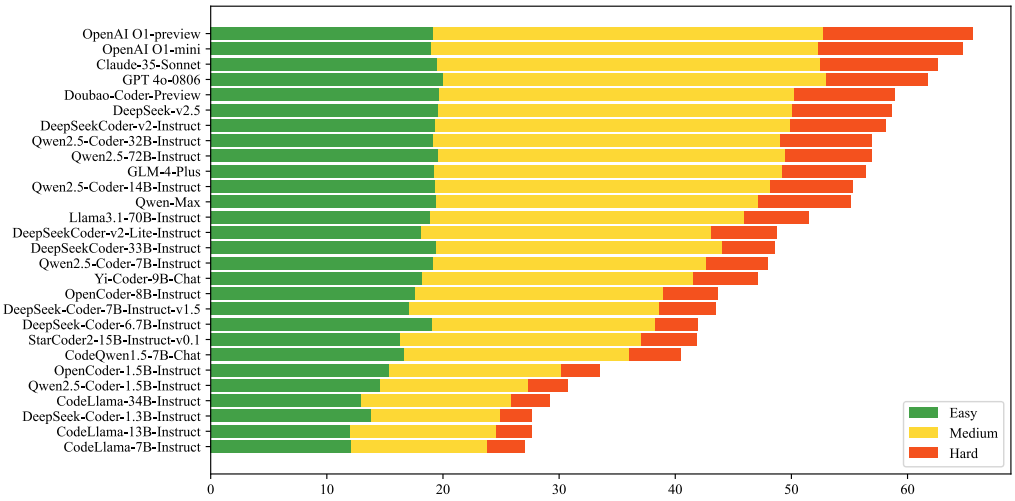


Figure 9. Visualization of performance on different difficulties.

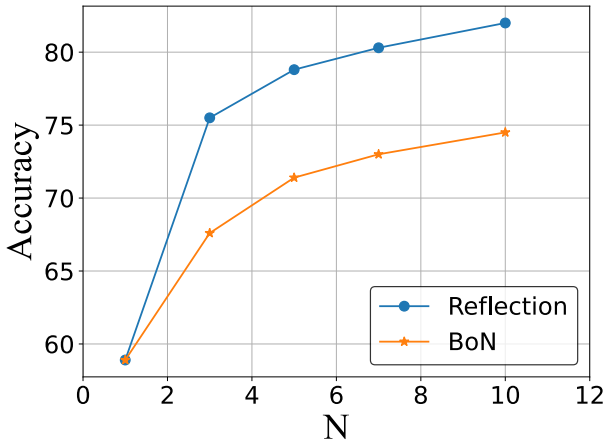


Figure 10. Comparison between BoN and Reflection.

Coder [Hui et al., 2024] gains remarkable progress in multilingual programming code generation and debugging tasks, such as MultiPL-E [Cassano et al., 2022], McEval [Chai et al., 2024], and MdEval [Liu et al., 2024b].

Code Benchmark. Program synthesis is an important task for code LLM, which forces the LLM to read the natural language description and then generates the corresponding code snippet meeting the user requirements [Athiwaratkun et al., 2023, Austin et al., 2021a, Gu et al., 2024, Lai et al., 2023, Liu et al., 2023a, Yu et al., 2024, Lu et al., 2022]. To further comprehensively evaluate the different aspects of capabilities of LLMs, numerous benchmarks are proposed, such as code translation [Jiao et al., 2023, Yan et al., 2023, Zhu et al., 2022], code retrieval [Huang et al., 2021, Husain et al., 2019, Li et al., 2024, Lu et al., 2021], and vulnerability repair [Huq et al., 2022, Prenner and Robbes, 2023, Richter and Wehrheim, 2022, Tian et al., 2024, Liu et al., 2024b], and structured data understanding [Wu et al., 2024, Su et al., 2024]. The recent work McEval [Chai et al., 2024] extends the number of programming languages to 40 for multilingual evaluation scenarios and MdEval [Liu et al., 2024b] creates a multilingual code debugging benchmark covering nearly 20 programming languages. Besides, many benchmarks have been proposed on repository-level code tasks [Agrawal et al., 2023, Allal et al., 2023, Bairi et al., 2023, Ding et al., 2022, Liu et al., 2023c, Pei et al., 2023, Shrivastava et al., 2023b,a, Zhang et al., 2023, Liu

et al., 2024a, Deng et al., 2024]. However, most previous workers focus on the capabilities of one aspect of the LLMs, neglecting to test the abilities of the LLM as a program developer across various real-world code development scenarios. In this work, we propose the FullStack Bench to assess the capabilities of LLMs across various real-world code development scenarios

6. Conclusion

In this paper, we provide a more holistic evaluation framework FullStack Bench with a corresponding effective execution environment SandboxFusion for code intelligence, which aims to evaluate multilingual programming capabilities in real-world code development scenarios. Specifically, first, our FullStack Bench mainly involves mainstream application domains (e.g., basic programming, software engineering, and machine learning) from 3374 problems, where each problem has corresponding unit test cases. Second, our SandboxFusion has three distinct features (i.e., Supporting various languages, Easy-to-deploy and Unified multi-dataset execution environment), which can satisfy the requirements of evaluating FullStack Bench. Finally, we hope that FullStack Bench could guide the researchers to better understand the code intelligence abilities of existing LLMs and accelerate the growth of foundation models.

7. Acknowledgements

We extend our heartfelt gratitude to the larger Seed team, whose dedication and expertise were crucial to the success of this project. Special thanks go to our engineering team for their technical prowess; our data teams, whose diligent efforts in data collection, annotation, and processing were indispensable; our evaluation team for their rigorous testing and insightful feedback; and the Seed-Foundation team for their valuable knowledge sharing. Their contributions have been instrumental to FullStack Bench.

8. Contributions

Siyao Liu implemented the dataset execution environment together with SandboxFusion, and performed quality assurance.

He Zhu validated the experimental results through analysis and made graphical visualizations. He Zhu, Jerry Liu, Jack Yang, Jinxiang Xia, Shukai Liu and Z.Y. Peng are affiliated with M-A-P.

Shulin Xin conducted experiments on pass rate and reflection.

Aoyan Li and Yifan Sun proposed and implemented the framework for domain classification.

Jerry Liu, Siyao Liu and He Zhu wrote the paper.

Li Chen, Aoyan Li, Siyao Liu, Rui Long, Yongsheng Xiao and Shulin Xin contributed to dataset samples, established and maintained an iterative process for organizing and curating datasets.

Shukai Liu, Z.Y. Peng, Jinxiang Xia and Jack Yang contributed supplementary samples in minority languages and domains, and performed revisions on these samples.

Kai Shen, Liang Xiang and Hongxia Yang provided research guidance and management support.

Yao Cheng, Jianfeng Chen, Jie Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Bo Li, Bowen Li, Linyi Li, Boyi Liu, Kaibo Liu, Qi Liu, Tianyi Liu, Tingkai Liu, Yongfei Liu, Guanghan Ning, Jiahao Su, Jing Su, Tao Sun, Yifan Sun, Yunzhe Tao, Guoyin Wang, Siwei Wang, Xuwu Wang, Yite Wang, Xia Xiao, Chenguang Xi, Jingjing Xu, Shikun Xu, Yingxiang Yang, Jianbo Yuan, Jun Zhang, Yufeng Zhang, Yuyu Zhang, Shen Zheng and Ming Zhu contributed to dataset samples and provided feedback during dataset iterations.

Aoyan Li, Qi Liu, Siyao Liu, Jing Mai, Zihan Wang and Shulin Xin implemented existing open-source datasets in SandboxFusion.

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

L. A. Agrawal, A. Kanade, N. Goyal, S. K. Lahiri, and S. K. Rajamani. Guiding language models of code with global context using monitors. *arXiv preprint arXiv:2306.10763*, 2023.

L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, et al. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023. URL https://arxiv.org/abs/2301.03988.

B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang, S. K. Gonugondla, H. Ding, V. Kumar, N. Fulton, A. Farahani, S. Jain, R. Giaquinto, H. Qian, M. K. Ramanathan, R. Nallapati, B. Ray, P. Bhatia, S. Sengupta, D. Roth, and B. Xiang. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=Bo7eeXm6An8.

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732, 2021a. URL https://arxiv.org/abs/2108.07732.

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021b. URL https://arxiv.org/abs/2108.07732.

J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

R. Bairi, A. Sonwane, A. Kanade, A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, S. Shet, et al. Codeplan: Repository-level coding using llms and planning. *arXiv preprint arXiv:2309.12499*, 2023.

S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, 2021. URL https://doi.org/10.5281/zenodo.5297715.

S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang, M. Pieler, U. S. Prashanth, S. Purohit, L. Reynolds, J. Tow, B. Wang, and S. Weinbach. GPT-NeoX-20B: An open-source autoregressive language model. In *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models*, pages 95–136, virtual+Dublin, 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.bigscience-1.9. URL https://aclanthology.org/2022.bigscience-1.9.

F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, et al. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*, 2022.

F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023. doi: 10.1109/TSE.2023.3267446.

L. Chai, S. Liu, J. Yang, Y. Yin, K. Jin, J. Liu, T. Sun, G. Zhang, C. Ren, H. Guo, et al. Mceval: Massively multilingual code evaluation. *arXiv preprint arXiv:2406.07436*, 2024.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code, 2021a. URL https://arxiv.org/abs/2107.03374.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374, 2021b. URL https://arxiv.org/abs/2107.03374.

A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

J. Dai, J. Lu, Y. Feng, D. Huang, G. Zeng, R. Ruan, M. Cheng, H. Tan, and Z. Guo. Mhpp: Exploring the capabilities and limitations of language models beyond basic code generation, 2024. URL https://arxiv.org/abs/2405.11430.

K. Deng, J. Liu, H. Zhu, C. Liu, J. Li, J. Wang, P. Zhao, C. Zhang, Y. Wu, X. Yin, Y. Zhang, W. Su, B. Xiang, T. Ge, and B. Zheng. R2c2-coder: Enhancing and benchmarking real-world repository-level code completion abilities of code large language models. *ArXiv*, abs/2406.01359, 2024.

Y. Ding, Z. Wang, W. U. Ahmad, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007*, 2022. URL https://arxiv.org/abs/2212.10007.

S. Dou, J. Zhang, J. Zang, Y. Tao, W. Zhou, H. Jia, S. Liu, Y. Yang, Z. Xi, S. Wu, S. Zhang, M. Wu, C. Lv, L. Xiong, W. Zhan, L. Zhang, R. Weng, J. Wang, X. Cai, Y. Wu, M. Wen, R. Zheng, T. Ji, Y. Cao, T. Gui, X. Qiu, Q. Zhang, and X. Huang. Multi-programming language sandbox for llms, 2024. URL https://arxiv.org/abs/2410.23074.

D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=hQwb-lbM6EL.

L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023.

A. Gu, B. Rozière, H. Leather, A. Solar-Lezama, G. Synnaeve, and S. I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024a. URL https://arxiv.org/abs/2401.14196.

D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024b.

D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt. Measuring coding challenge competence with APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL https://openreview.net/forum?id=sD93GOzH3i5.

J. Huang, D. Tang, L. Shou, M. Gong, K. Xu, D. Jiang, M. Zhou, and N. Duan. Cosqa: 20,000+ web queries for code search and question answering. *arXiv preprint arXiv:2105.13239*, 2021.

S. Huang, T. Cheng, J. Klein Liu, J. Hao, L. Song, Y. Xu, J. Yang, J. Liu, C. Zhang, L. Chai, et al. Opencoder: The open cookbook for top-tier code large language models. *arXiv e-prints*, pages arXiv–2411, 2024a.

S. Huang, T. Cheng, J. K. Liu, J. Hao, L. Song, Y. Xu, J. Yang, J. H. Liu, C. Zhang, L. Chai, R. Yuan, Z. Zhang, J. Fu, Q. Liu, G. Zhang, Z. Wang, Y. Qi, Y. Xu, and W. Chu. Opencoder: The open cookbook for top-tier code large language models. In *arXiv preprint*, 2024b.

B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

F. Huq, M. Hasan, M. M. A. Haque, S. Mahbub, A. Iqbal, and T. Ahmed. Review4repair: Code review aided automatic program repairing. *Information and Software Technology*, 143:106765, 2022.

H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019. URL https://arxiv.org/abs/1909.09436.

N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

M. Jiao, T. Yu, X. Li, G. Qiu, X. Gu, and B. Shen. On the evaluation of neural code translation: Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1529–1541. IEEE, 2023.

M. A. M. Khan, M. S. Bari, X. L. Do, W. Wang, M. R. Parvez, and S. Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval, 2023.

W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-T. Yih, D. Fried, S. Wang, and T. Yu. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501, 2022.

Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W. Yih, D. Fried, S. I. Wang, and T. Yu. DS-1000: A natural and reliable benchmark for data science code generation. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR, 2023. URL https://proceedings.mlr.press/v202/lai23b.html.

LangGenius. Difysandbox. https://github.com/langgenius/dify-sandbox, 2024.

H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *ArXiv*, abs/2207.01780, 2022. URL https://api.semanticscholar.org/CorpusID:250280117.

R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023. URL https://arxiv.org/abs/2305.06161.

Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, et al. Competition-level code generation with alphacode. *ArXiv preprint*, abs/2203.07814, 2022. URL https://arxiv.org/abs/2203.07814.

Z. Li, J. Zhang, C. Yin, Y. Ouyang, and W. Rong. Procqa: A large-scale community-based programming question answering dataset for code search. *arXiv preprint arXiv:2403.16702*, 2024.

J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023a. URL http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html.

J. Liu, K. Deng, C. Liu, J. Yang, S. Liu, H. Zhu, P. Zhao, L. Chai, Y. Wu, K. Jin, G. Zhang, Z. M. Wang, G. Zhang, B. Xiang, W. Su, and B. Zheng. M2rc-eval: Massively multilingual repository-level code completion evaluation. In *arXiv preprint*, 2024a.

M. Liu, N. Pinckney, B. Khailany, and H. Ren. Verilogeval: Evaluating large language models for verilog code generation, 2023b. URL https://arxiv.org/abs/2309.07544.

S. Liu, L. Chai, J. Yang, J. Shi, H. Zhu, L. Wang, K. Jin, W. Zhang, H. Zhu, S. Guo, et al. Mdeval: Massively multilingual code debugging. *arXiv preprint arXiv:2411.02310*, 2024b.

T. Liu, C. Xu, and J. J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *CoRR*, abs/2306.03091, 2023c. doi: 10.48550/ARXIV.2306.03091. URL https://doi.org/10.48550/arXiv.2306.03091.

A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. GONG, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. LIU. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL https://openreview.net/forum?id=6lE4dQXaUcb.

S. Lu, N. Duan, H. Han, D. Guo, S.-w. Hwang, and A. Svyatkovskiy. ReACC: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.431. URL https://aclanthology.org/2022.acl-long.431.

A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=iaYcJKpY2B_.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. *ArXiv*, abs/1912.01703, 2019.

H. Pei, J. Zhao, L. Lausen, S. Zha, and G. Karypis. Better context makes better code language models: A case study on function call argument completion. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'23/IAAI'23/EAAI'23. AAAI Press, 2023. ISBN 978-1-57735-880-0. doi: 10.1609/aaai.v37i4.25653. URL https://doi.org/10.1609/aaai.v37i4.25653.

J. A. Prenner and R. Robbes. Runbugrun–an executable dataset for automated program repair. *arXiv preprint arXiv:2304.01102*, 2023.

C. Richter and H. Wehrheim. Tssb-3m: Mining single statement bugs at massive scale. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 418–422, 2022.

B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

D. Shrivastava, D. Kocetkov, H. de Vries, D. Bahdanau, and T. Scholak. Repofusion: Training code models to understand your repository. *arXiv preprint arXiv:2306.10998*, 2023a.

D. Shrivastava, H. Larochelle, and D. Tarlow. Repository-level prompt generation for large language models of code. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and

J. Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31693–31715. PMLR, 23–29 Jul 2023b. URL https://proceedings.mlr.press/v202/shrivastava23a.html.

A. Su, A. Wang, C. Ye, C. Zhou, G. Zhang, G. Zhu, H. Wang, H. Xu, H. Chen, H. Li, et al. Tablegpt2: A large multimodal model with tabular data integration. *arXiv preprint arXiv:2411.02059*, 2024.

T. Sun, L. Chai, Y. Y. Jian Yang, H. Guo, J. Liu, B. Wang, L. Yang, and Z. Li. Unicoder: Scaling code large language model via universal code. *ACL*, 2024.

L. Team. The llama 3 herd of models. *arXiv preprint arXiv: 2407.21783*, 2024.

R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Z. Liu, and M. Sun. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621*, 2024.

T. Wei, L. Zhao, L. Zhang, B. Zhu, L. Wang, H. Yang, B. Li, C. Cheng, W. Lü, R. Hu, C. Li, L. Yang, X. Luo, X. Wu, L. Liu, W. Cheng, P. Cheng, J. Zhang, X. Zhang, L. Lin, X. Wang, Y. Ma, C. Dong, Y. Sun, Y. Chen, Y. Peng, X. Liang, S. Yan, H. Fang, and Y. Zhou. Skywork: A more open bilingual foundation model, 2023. URL https://arxiv.org/abs/2310.19341.

X. Wu, J. Yang, L. Chai, G. Zhang, J. Liu, X. Du, D. Liang, D. Shu, X. Cheng, T. Sun, et al. Tablebench: A comprehensive and complex benchmark for table question answering. *arXiv preprint arXiv:2408.09174*, 2024.

F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

W. Yan, Y. Tian, Y. Li, Q. Chen, and W. Wang. Codetransocean: A comprehensive multilingual benchmark for code translation. *arXiv preprint arXiv:2310.04951*, 2023.

A. Young, B. Chen, C. Li, C. Huang, G. Zhang, G. Zhang, H. Li, J. Zhu, J. Chen, J. Chang, et al. Yi: Open foundation models by 01. ai. *arXiv preprint arXiv:2403.04652*, 2024.

H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

F. Zhang, B. Chen, Y. Zhang, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023. URL https://arxiv.org/abs/2303.12570.

S. Zhang, H. Zhao, X. Liu, Q. Zheng, Z. Qi, X. Gu, X. Zhang, Y. Dong, and J. Tang. Naturalcodebench: Examining coding performance mismatch on humaneval and natural user prompts, 2024. URL https://arxiv.org/abs/2405.04520.

C. T. H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C. A. Choquette-Choo, J. Shen, J. Kelley, K. tij Bansal, L. Vilnis, M. Wirth, P. Michel, P. Choy, P. Joshi, R. Kumar, S. Hashmi, S. Agrawal, Z. Gong, J. Fine, T. B. Warkentin, A. J. Hartman, B. Ni, K. Korevec, K. Schaefer, and S. Huffman. Codegemma: Open code models based on gemma. *ArXiv*, abs/2406.11409, 2024. URL https://api.semanticscholar.org/CorpusID:270560319.

K. Zheng, J. M. Han, and S. Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics, 2022. URL https://arxiv.org/abs/2109.00110.

Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*, abs/2303.17568, 2023. doi: 10.48550/ARXIV.2303.17568. URL https://doi.org/10.48550/arXiv.2303.17568.

K. Zhu, Q. Zang, S. Jia, S. Wu, F. Fang, Y. Li, S. Guo, T. Zheng, B. Li, H. Wu, et al. Lime-m: Less is more for evaluation of mllms. *arXiv preprint arXiv:2409.06851*, 2024a.

M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C. K. Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*, 2022.

Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024b.

Series Name	Model Name (Size)
Llama	CodeLlama-7B-Instruct (7B) CodeLlama-13B-Instruct (13B) CodeLlama-34B-Instruct (34B)
Qwen2.5	Qwen2.5-Coder-1.5B-Instruct (1.5B) Qwen2.5-Coder-7B-Instruct (7B) Qwen2.5-Coder-14B-Instruct (14B) Qwen2.5-Coder-32B-Instruct (32B)
DeepSeek v1	DeepSeek-Coder-1.3B-Instruct (1.3B) DeepSeek-Coder-6.7B-Instruct (6.7B) DeepSeekCoder-33B-Instruct (33B)
DeepSeek v2	DeepSeek-Coder-7B-Instruct-v1.5 (7B) DeepSeekCoder-v2-Lite-Instruct (16B) DeepSeekCoder-v2-Instruct (236B)
OpenCoder	OpenCoder-1.5B-Instruct (1.5B) OpenCoder-8B-Instruct (8B)

Table 4. Model Series discussed in Section 4.4.

A. Appendix

A.1. Visualization on the cases of FullStack Bench

A.2. Details of SandboxFusion

A.2.1. Dataset Module

The sandbox aims to provide a unified framework for most execution-based datasets, making it easier to add existing datasets or create new datasets based on them. In particular, SandboxFusion implements many open-source code evaluation datasets, including HumanEval [Chen et al., 2021a], MultiPL-E [Cassano et al., 2022], Shadow Humaneval [Wei et al., 2023], CodeContests [Li et al., 2022], MBPP [Austin et al., 2021b], MBXP [Chai et al., 2024], MHPP [Dai et al., 2024], CRUXEval [Gu et al., 2024], NaturalCodeBench [Zhang et al., 2024], PAL-Math [Gao et al., 2023], verilog-eval [Liu et al., 2023b] and miniF2F [Zheng et al., 2022]. Besides, as the judgment logic of most datasets is very similar, to maximize code reuse, we created two representative types of datasets: AutoEvalDataset and CommonOJDataset as follows:

- AutoEvalDataset. This type of dataset is primarily prepared for instruction-tuning models and can also be used to test the performance of pre-trained models through few-shot learning. The basic process is to extract the entire code block from the model's output, then concatenate the model code with pre-written test scripts and execute them, and finally check the return value of the executed program to determine whether the problem is passed. Note that we support FullStack Bench in this dataset mode.
- CommonOJDataset. Most algorithmic competition problems belong to this category. These problems are language-agnostic and can be tested in any programming language. Most algorithmic competition datasets only store the problem description of the algorithmic competition problem and the test cases in the form of standard input and output. In the prompt generation stage, the problem description is combined with the instruction of

Open-Sourced Model	Model Link
CodeQwen1.5-7B-Instruct	https://hf.co/Qwen/CodeQwen1.5-7B-Instruct
Qwen2.5-Coder-1.5B-Instruct	https://hf.co/Qwen/Qwen2.5-Coder-1.5B-Instruct
Qwen2.5-Coder-7B-Instruct	https://hf.co/Qwen/Qwen2.5-Coder-7B-Instruct
Qwen2.5-Coder-14B-Instruct	https://hf.co/Qwen/Qwen2.5-Coder-14B-Instruct
Qwen2.5-Coder-32B-Instruct	https://hf.co/Qwen/Qwen2.5-Coder-32B-Instruct
Qwen2.5-72B-Instruct	https://hf.co/Qwen/Qwen2.5-72B-Instruct
CodeLlama-7B-Instruct	https://hf.co/meta-llama/CodeLlama-7b-Instruct-hf
CodeLlama-34B-Instruct	https://hf.co/meta-llama/CodeLlama-34b-Instruct-hf
CodeLlama-13B-Instruct	https://hf.co/meta-llama/CodeLlama-13b-Instruct-hf
Llama3.1-70B-Instruct	https://hf.co/meta-llama/Llama-3.1-70B-Instruct
DeepSeek-Coder-1.3B-Instruct	https://hf.co/deepseek-ai/deepseek-coder-1.3b-instruct
DeepSeek-Coder-6.7B-Instruct	https://hf.co/deepseek-ai/deepseek-coder-6.7b-instruct
DeepSeek-Coder-33B-Instruct	https://hf.co/deepseek-ai/deepseek-coder-33b-instruct
DeepSeek-Coder-7B-Instruct-v1.5	https://hf.co/deepseek-ai/deepseek-coder-7b-instruct-v1.5
DeepSeek-Coder-V2-Lite-Instruct	https://hf.co/deepseek-ai/DeepSeek-Coder-V2-Lite-instruct
DeepSeek-Coder-V2-Instruct	https://hf.co/deepseek-ai/DeepSeek-Coder-V2-Instruct
OpenCoder-1.5B-Instruct	https://hf.co/infly/OpenCoder-1.5B-Instruct
OpenCoder-8B-Instruct	https://hf.co/infly/OpenCoder-8B-Instruct
Yi-Coder-9B-Chat	https://hf.co/01-ai/Yi-Coder-9B-Chat
StarCoder2-15B-Instruct-v0.1	https://hf.co/bigcode/starcoder2-15b-instruct-v0.1

Table 5. Open-sourced models adopted in our experiments.

Close-Sourced Model	API Entry
Claude-3.5-Sonnet	https://www.anthropic.com/news/claude-3-5-sonnet
OpenAI o1-preview	https://platform.openai.com/docs/models#o1
OpenAI o1-mini	https://platform.openai.com/docs/models#o1
GPT 4o-0806	https://platform.openai.com/docs/models#gpt-4o
DeepSeek-v2.5	https://www.deepseek.com
GLM-4-Plus	https://open.bigmodel.cn/dev/api/normal-model/glm-4
Qwen-Max	https://www.aliyun.com/product/bailian

Table 6. Close-sourced models (APIs) adopted in our experiments.

the corresponding language as the complete prompt. The entire code block is extracted from the model's output and directly executed, and the standard output under the given standard input is compared to see if it matches the expectation.

A.2.2. Sandbox Execution Module

SandboxFusion covers a wide range of programming languages that have received attention in the field of code generation and provides a unified interface for executing them. Currently, SandboxFusion supports a total of 23 programming languages, including Python, C++, C#, Go, Java, NodeJS, TypeScript, PHP, Rust, Bash, Lua, R, Perl, D, Ruby, Scala, Julia, Kotlin, Verilog, Lean, Swift, Racket, and CUDA. Together with these language compilers and interpreters, SandboxFusion also integrates many commonly used packages like PyTorch and TensorFlow for machine learning and a browser for front-end related application domains. The main features of the Sandbox Execution Module are as follows:

- Our execution interface accepts a code snippet, language, and other input contexts, and returns the program's return code, standard output, and other information. This interface hides many language-specific implementation details from researchers and greatly improves the efficiency of data cleaning, compiler feedback, etc.
- When dealing with incompatible versions of languages and packages, our principle is to select more advanced and popular versions. For example, while there are still many existing Lean codes written in Lean 3 on Github, we still use Lean 4 as the Lean community has fully transitioned to it. In cases where an older version of some package is indeed necessary, SandboxFusion also provides a function to run code in an isolated execution system to avoid affecting the main environment.
- SandboxFusion provides functionality in some task-specific areas. For example, sandbox supports running CUDA and Python code in a GPU environment, making it possible to explore enhancing the performance of CUDA code. Another example is the supported Jupyter mode, which executes multiple code blocks in sequence, providing training signals for online user interaction.
- SandboxFusion also has a built-in resource isolation functionality. Given limiting CPU and memory usage, it is possible to test a program's performance in a controlled environment, providing a reliable metric for program optimization. Its file system isolation allows testing scenarios that require file operations without affecting the host system, while network isolation enables multiple programs to bind to the same port simultaneously, preventing conflicts during concurrent testing.

A.3. Comparison with Other Sandboxes

For comparison, we selected three representative sandboxes (i.e., DifySandbox, MultiPL-E, and MPLSandbox) as follows:

- DifySandbox [LangGenius, 2024] represents online code execution sandboxes with powerful and fine-grained security controls. In contrast, SandboxFusion, primarily used for internal model code evaluation, has lower security isolation requirements and thus only implements comprehensive isolation through basic Linux kernel capabilities like namespaces and cgroups, without providing fine-grained permission controls. Additionally, DifySandbox only supports Python and NodeJS runtime environments and has not implemented any code evaluation datasets. Compared to DifySandbox, SandboxFusion is

	SandboxFusion	DifySandbox	MultiPL-E	MPLSandbox
# Datasets	10+	0	2	0
# Languages	23	2	18	8
Security	namespace & cgroup	seccomp	none	none(docker)
API Type	HTTP & SDK	HTTP	CLI	SDK & CLI
Deployment	Single Server	Single Server	CLI Tool	1 Server + 8 Language Workers
Others	Jupyter Mode	Fine-Grained Security Limitation	-	Language Analysis

Table 7. Comparison of different sandboxes.

optimized for large model evaluation and training scenarios, offering higher throughput and support for more datasets and languages.

- MultiPL-E [Cassano et al., 2023] represents dataset-bundled code sandboxes. MultiPL-E built a problem translation pipeline and includes multiple language environments within its sandbox. However, it only serves the MultiPL-E dataset, and runs offline evaluation through CLI, making it impossible to integrate with online training tasks. Therefore, compared to such dataset-bundled execution environment sandboxes, SandboxFusion's advantages lie in its online interface and unified multi-dataset execution environment.
- MPLSandbox [Dou et al., 2024] represents research-oriented code sandboxes with rich code feedback signals (e.g., code execution, static analysis, and quality assessment). However, compared to SandboxFusion, MPLSandbox's deployment and usage are more complex. SandboxFusion requires only one docker image to run, while MPLSandbox needs to launch a service to execute each language. Furthermore, SandboxFusion provides enhanced security features, whereas MPLSandbox can only rely on container runtime environments for isolation, unable to prevent code interference between requests within containers.

More importantly, SandboxFusion can be used to evaluate the capabilities of LLM in various real-world code development scenarios. For instance, it supports testing desktop programs written with PyQt, web applications built with Flask, and neural networks with PyTorch or Tensorflow in FullStack Bench. These codes are very common in real-world coding scenarios, but other sandboxes cannot support such testing.

Figure 11. Visualization on some cases of our FullStack Bench.