HumanEval Pro and MBPP Pro: Evaluating Large Language Models on Self-invoking Code Generation

Zhaojian Yu[1] **Yilun Zhao**[2] **Arman Cohan**[2] **Xiao-Ping Zhang**[1] *

[1]Tsinghua University [2]Yale University

github.com/CodeEval-Pro/CodeEval-Pro

Abstract

We introduce self-invoking code generation, a new task designed to evaluate the progressive reasoning and problem-solving capabilities of LLMs. In this task, models are presented with a base problem and a related, more complex problem. They must solve the base problem and then utilize its solution to address the more complex one. This work features three key contributions. First, we propose a general recipe for generating more challenging versions of existing benchmarks, resulting in three new benchmarks: HumanEval Pro, MBPP Pro, and BigCodeBench-Lite Pro, specifically designed to assess LLMs on self-invoking code generation. Second, our analysis of more than twenty LLMs reveals two key observations: (i) Most LLMs excel in traditional code generation benchmarks, but their performance declines on self-invoking tasks. For example, o1-mini achieves 96.2% pass@1 on HumanEval but only 76.2% on HumanEval Pro. (ii) On self-invoking code generation task, the instruction-tuned models demonstrate only marginal improvements compared to the base models. Third, we disclose the types of failure modes observed in current models. All these results underscore the need for further advancements in self-invoking code generation tasks and provide a new direction for enhancing LLMs' code reasoning capabilities.

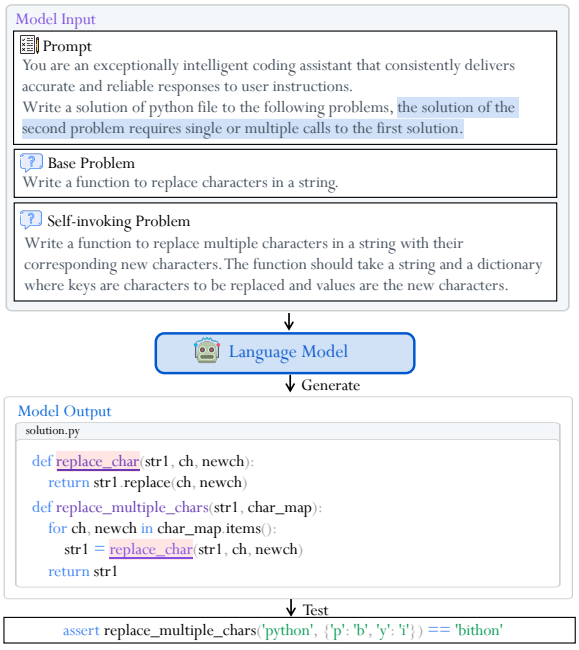


Figure 1: The overview of self-invoking code generation in HumanEval Pro and MBPP Pro. Given a base problem and a related, more complex problem, they are required to solve the base problem and use its solution to address the complex problems.

1 Introduction

Large Language Models (LLMs) have demonstrated significant progress in various code-related tasks including code generation (Roziere et al., 2023; Zhang et al., 2023; Ni et al., 2024), program repair (Xia et al., 2022; Jin et al., 2023), and code translation (Zhu et al., 2022), etc. Traditional human-annotated benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) have been widely adopted to evaluate the code generation abilities of LLMs, providing standardized evaluation protocols for assessing their performance on code-related tasks. However, these existing benchmarks primarily focus on isolated, single-function code generation, which represents only a subset of the challenges encountered in real-world software development scenarios.

To evaluate LLMs under more realistic problem-solving scenarios, BigCodeBench (Zhuo et al., 2024) presents a benchmark that comprises of complex and practical problems requiring LLMs to use multiple function calls from diverse libraries. While BigCodeBench highlights the use of *external* function calls, it falls short in assessing LLMs'

*Corresponding Author

Findings of the Association for Computational Linguistics: ACL 2025, pages 13253–13279

reasoning ability to generate and invoke their own generated functions in problem-solving. CRUX-Eval (Gu et al., 2024) assesses LLMs' code reasoning by predicting function inputs and outputs. However, the direct input and output prediction does not involve explicit code generation. In practical software engineering contexts, developers must not only write code but also comprehend, modify, and utilize existing code to solve more complex problems. Hence, the ability to understand and subsequently leverage one's own generated code, namely *self-invoking code generation* (Figure 1), plays an important role for LLMs to leverage their reasoning capabilities to code generation that current benchmarks fail to capture.

Therefore, we present **HumanEval Pro** and **MBPP Pro**, two expanded versions of the traditional HumanEval and MBPP benchmarks to evaluate LLMs on self-invoking code generation task. As illustrated in Figure 1, HumanEval Pro and MBPP Pro extend beyond simple code generation by introducing self-invoking problems which requires LLMs to solve the base problem and invoke their self-generated code to solve a more complex problem. By evaluating LLMs on self-invoking code generation task, HumanEval Pro and MBPP Pro provide a useful and important probe to better understand the programming capabilities of LLMs. The capability of self-invoking code generation also facilitates LLMs to tackle difficult tasks with greater autonomy and effectiveness.

To obtain HumanEval Pro and MBPP Pro, we propose a general recipe for constructing self-invoking code generation benchmarks by building upon existing datasets. First, we use Deepseek-V2.5 (DeepSeek-AI, 2024) to generate self-invoking problems based on the original problems in HumanEval and MBPP. These problems are designed to be more complex than the base problems and closely related to them, ensuring progressive reasoning and coherent code invocation. Second, we generate the candidate solution and test inputs for each problem. Third, we execute the code of candidate solution to generate output and use the *assert* command in Python to build test cases. In the third stage, human experts are assigned to manually review each problem and continuously modify and execute the code of solutions to ensure that all canonical solutions could correctly solve the problem and cover the test cases.

Through extensive evaluation of various LLMs on HumanEval Pro and MBPP Pro, we uncover a significant disparity between traditional code generation and self-invoking code generation capabilities. Our findings reveal that while frontier LLMs excel at generating individual code snippets, they often struggle to effectively utilizing their own generated code for solving more complex problems. For example, o1-mini achieves 96.2% pass@1 on HumanEval but only 76.2% on HumanEval Pro, demonstrating the challenges inherent in self-invoking code generation. From the comparison between instruction-tuned models and their base models, we found that instruction-tuned models are less efficient on self-invoking code generation than traditional code generation task. Furthermore, our detailed statistics of failure cases in HumanEval Pro and MBPP Pro also reflect the shortcomings of LLMs in self-invoking code generation, thereby providing complementary insights on real-world coding capabilities of LLMs.

2 Related Work

Benchmarks for Code Generation The evaluation landscape for Code LLMs has evolved significantly. HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) serve as fundamental benchmarks, focusing on Python function completion tasks with test-driven evaluation. Several benchmarks have expanded code evaluation benchmarks to encompass multiple programming languages (Zheng et al., 2023; Athiwaratkun et al., 2022), complex tasks like program repair (Haque et al., 2022; Jiang et al., 2023; Muennighoff et al., 2024; Xia et al., 2024), dynamic problem sets (Jain et al., 2024), and simulated execution (Gu et al., 2024). PECC (Haller et al., 2024) delivers a two-prompt benchmark derived from Advent Of Code (AoC) challenges and Project Euler. To evaluate LLMs in professional software engineering, benchmarks like SWE-Bench (Jimenez et al., 2023), EvoCodeBench (Li et al., 2024), RepoBench (Liu et al., 2023), and GoogleCodeRepo (Shrivastava et al., 2023) focus on real-world tasks, code evolution, and repository-level challenges. These benchmarks collectively drive the advancement of LLMs, providing valuable insights into their strengths and limitations. Our benchmarks introduce novel self-invoking code generation task, which addresses gaps left by existing benchmarks. This addition provides a more holistic framework to evaluate LLMs on leveraging their reasoning capabilities to code generation.

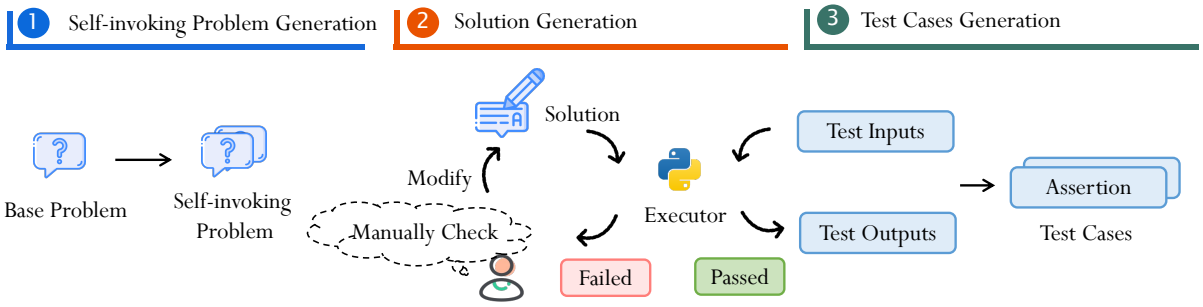


Figure 2: The overview of benchmark construction. An example is shown in Figure 8. We summarize the entire benchmark construction process as follows: (1) **Self-invoking problem Generation**: We use Deepseek-V2.5 (we discuss the impact of data generating model in Appendix A) to generate the self-invoking problems, as well as their candidate solutions and test inputs. (2) **Solutions Generation**: We execute the generated solution with the test inputs in a controlled Python environment to obtain ground truth outputs. (3) **Test Cases Generation**: We employ an iterative method involving Python execution check and manual review to ensure that all test cases pass successfully. The final execution results are then used to construct complete test cases with `assert` command.

LLMs for Code Generation The development of LLMs for Code Generation has seen significant progress. CodeX (Chen et al., 2021) pioneered this direction by fine-tuning GPT models on code-specific data. Subsequent models like CodeGeeX (Zheng et al., 2023) and CodeLLaMA (Roziere et al., 2023) further advanced the field by incorporating multilingual code understanding and generation capabilities. StarCoder (Li et al., 2023), DeepseekCoder (Zhu et al., 2024) and Qwen2.5-Coder (Hui et al., 2024) demonstrated the importance of high-quality code data curation and specialized architecture designs. Building upon these models, researchers have explored instruction-tuning approaches using GPT-4 or GPT-3.5 as teachers. Notable examples include WizardCoder (Luo et al., 2023), Magicoder (Wei et al., 2024), WaveCoder (Yu et al., 2024), OpenCodeInterpreter (Zheng et al., 2024). These models have achieved impressive performance on code generation benchmarks through well-desighed post-training recipe.

3 Benchmark Construction

To facilitate a meaningful comparison between self-invoking code generation and traditional code generation, we have crafted two new benchmarks, HumanEval Pro and MBPP Pro. These benchmarks are extensions of the original HumanEval and MBPP, requiring the model to solve both the base problem and a more complex self-invoking problem. In addressing the self-invoking problems, LLMs are required to apply the solutions they have independently generated for the base problem. This evaluation of self-invoking code generation offers deeper insights into the programming capabilities of LLMs, extending beyond the scope of single-problem code generation. The benchmark construction process, illustrated in Figure 2, will be discussed in detail in the following subsections.

3.1 Self-invoking Problem Generation

To ensure that all benchmarks are permissively licensed, we employ one of the state-of-the-art (SoTA) open-source models, DeepSeek-V2.5, to create new problems and solutions derived from the original HumanEval and MBPP datasets. Two main guidelines is established for self-invoking problems generation to rigorously evaluate LLMs. 1) **Complexity Enhancement**: The self-invoking problems should introduce additional programming challenges while preserving the core functionality of the original problems. This ensures that successful solutions require both understanding of the original code and ability to extend it appropriately. 2) **Semantic Relevance**: The self-invoking problems should maintain sufficient semantic similarity to their original counterparts to enable meaningful self-invoking code generation process. Appendix H.1 presents the prompt for self-invoking problem generation.

3.2 Solution Generation

In self-invoking problem generation process, the candidate solution and test inputs are generated simultaneously with the self-invoking problem. However, when dealing with self-invoking problems, these generated solutions are often flawed, which can lead to execution errors during the verifica-

Iteration	HumanEval Pro (%)	MBPP Pro (%)
Round 1	64.0	84.7
Round 2	98.8	99.7
Round 3	100.0	100.0

Table 1: Pass@1 (%) of candidate solutions across different iteration rounds for canonical solution and test case generation with human manual review.

tion process, thereby highlighting a significant challenge in maintaining the accuracy and effectiveness of these test cases. Therefore, as shown in Figure 2, we propose a method to iteratively execute the solution code with test inputs and obtain expected outputs correctly. For the execution errors, the authors manually analyze these errors and modify the solutions to ensure that the final solution can cover all the test cases comprehensively. The manual review process involves (1) identifying the root causes of the errors, (2) making necessary adjustments to the code or algorithm, and (3) re-evaluating the solution against the entire set of test cases to confirm its correctness and completeness. Table 1 shows that our rigorous verification process ensures the high quality of our benchmarks.

3.3 Test Cases Generation

After obtaining the self-invoking problem and its candidates solution, a critical challenge is ensuring the reliability of the test cases (with both test inputs and expected execution outputs) to validate the the generated solutions. Despite the apparent simplicity of using the same LLM context to generate both problems and test cases, CRUXEval (Gu et al., 2024) results show that even leading models like GPT-4 achieve only a 63.4% pass@1 rate in test output prediction. This suggests that using models like GPT-4 to directly generate test cases for problems will lead to many inaccurate evaluation results. Our iterative verification method effectively addresses this challenge. By combining Python execution checks with manual reviews, we ensure that all test cases accurately assess solution correctness and achieves a 100% pass@1 under correct implementation conditions. Furthermore, we categorize the common execution errors that occur during test case generation into four main types: *variable type mismatches*, *index out of bounds*, *invalid input handling*, and *edge case failures*. To obtain the high-quality self-invoking problem solutions, we adopt main remediation strategies including: (1) implementing input validation, (2) adding type checking, (3) handling edge cases explicitly, and (4) refining problem specifications when necessary. Beyond basic execution correctness, we also verify the self-invoking problem and solutions in the following aspects: (1) logical consistency between problem statements and test cases, (2) coverage of essential edge cases, and (3) alignment with original problem objectives.

4 Experiment and Analysis

Following previous work (Chen et al., 2021), We use the *pass@k* (Chen et al., 2021) score as the evaluation metric of HumanEval Pro and MBPP Pro. We use greedy decoding strategy to generate solutions for all open-source models and set *temperature=0.2* for all API-models (model information is shown in Appendix E). For all previous benchmarks, we use the reported results whenever available; otherwise, we evaluate using the EvalPlus codebase (Liu et al., 2024).

Table 2 presents the *pass@1* scores of HumanEval Pro and MBPP Pro alongside those of other relevant benchmarks, including HumanEval, HumanEval+, MBPP, and MBPP+ (Liu et al., 2024), highlighting the following salient observations: 1) Most LLMs have a 10% to 15% absolute performance drop on self-invoking code generation benchmarks. 2) Large size open-source LLMs have comparable performance with proprietary LLMs on self-invoking benchmarks. Notably, DeepseekCoder-V2-instruct achieves 77.4% on HumanEval Pro, surpassing the score of all proprietary LLMs. 3) Most instruction-tuned models have less improvements on self-invoking code generation benchmarks (e.g., HumanEval Pro) than traditional benchmarks (e.g.,HumanEval). For instance, Qwen2.5Coder-32B-instruct have 26.8% absolute improvement on HumanEval compared to Qwen2.5Coder-32B-base (from 65.9% to 92.7%) but only 8.5% on HumanEval Pro (from 61.6% to 70.1%). Appendix C also presents the evaluation results for different k values with the sampling generation strategy.

4.1 Base Model vs Instruct Model

Currently, the training of LLMs is typically divided into two stages: a pre-training stage that relies on self-supervised learning, and a subsequent supervised fine-tuning stage based on <instruction, response> pairs. Previous studies (Luo et al., 2023; Hui et al., 2024; Wei et al., 2024) have shown that

Model	Params	HumanEval (+)	HumanEval Pro (0-shot)	HumanEval Pro (1-shot)	MBPP (+)	MBPP Pro (0-shot)	MBPP Pro (1-shot)
Proprietary Models							
o1-mini	-	97.6 (90.2)	76.2	84.8	93.9 (78.3)	68.3	81.2
GPT-4o	-	90.2 (86.0)	75.0	77.4	86.8 (72.5)	70.9	80.2
GPT-4-Turbo	-	90.2 (86.6)	72.0	76.2	85.7 (73.3)	69.3	73.3
Claude-3.5-sonnet	-	92.1 (86.0)	72.6	79.9	91.0 (74.6)	66.4	76.2
Open-source Models							
Deepseek-V2.5	-	90.2 (83.5)	73.8	76.8	87.6 (74.1)	71.2	77.5
DeepseekCoder-V2-instruct	21/236B	90.2 (84.8)	77.4	82.3	89.4 (76.2)	71.4	76.5
Qwen2.5-Coder-1.5B-base	1.5B	43.9 (36.6)	37.2	39.6	69.2 (58.6)	48.4	51.3
Qwen2.5-Coder-1.5B-instruct	1.5B	70.7 (66.5)	33.5	37.8	69.2 (59.4)	42.1	43.7
DeepseekCoder-6.7B-base	6.7B	49.4 (39.6)	35.4	36.6	70.2 (51.6)	50.5	55.0
DeepseekCoder-6.7B-instruct	6.7B	78.6 (71.3)	55.5	61.6	74.9 (65.6)	57.1	58.2
Magicoder-S-DS-6.7B	6.7B	76.8 (70.7)	54.3	56.7	75.7 (64.4)	58.7	64.6
WaveCoder-Ultra-6.7B	6.7B	78.6 (69.5)	54.9	59.8	74.9 (63.5)	60.1	64.6
Qwen2.5-Coder-7B-base	7B	61.6 (53.0)	54.9	56.1	76.9 (62.9)	61.4	68.0
Qwen2.5-Coder-7B-instruct	7B	88.4 (84.1)	65.9	67.1	83.5 (71.7)	64.8	69.8
OpenCoder-8B-base	8B	66.5 (63.4)	39.0	42.1	79.9 (70.4)	52.4	53.7
OpenCoder-8B-instruct	8B	83.5 (78.7)	59.1	54.9	79.1 (69.0)	57.9	61.4
Yi-Coder-9B-base	9B	53.7 (46.3)	42.7	50.0	78.3 (64.6)	60.3	61.4
Yi-Coder-9B-chat	9B	85.4 (74.4)	59.8	64.0	81.5 (69.3)	64.8	71.7
Codestral-22B-v0.1	22B	81.1 (73.2)	59.1	65.9	78.2 (62.2)	63.8	71.2
DeepseekCoder-33B-base	33B	56.1 (47.6)	49.4	49.4	74.2 (60.7)	59.0	65.1
DeepseekCoder-33B-instruct	33B	79.3 (75.0)	56.7	62.8	80.4 (70.1)	64.0	68.3
Qwen2.5-Coder-32B-base	32B	65.9 (60.4)	61.6	67.1	83.0 (68.2)	67.7	73.3
Qwen2.5-Coder-32B-instruct	32B	92.7 (87.2)	70.1	80.5	90.2 (75.1)	69.8	77.5
LLaMA3-70B-instruct	70B	81.7 (72.0)	60.4	64.6	82.3 (69.0)	63.5	70.4

Table 2: Main result of different models on HumanEval Pro and MBPP Pro. More results is shown in Appendix C.

the instruction-based supervised fine-tuning stage can significantly enhance the code generation capabilities of base models on traditional benchmarks. For example, as shown in Table 2, Qwen2.5-Coder-instruct 7B started with the Qwen2.5-Coder-7B base model and improved the HumanEval pass@1 score from 61.6% to 88.4%. There remains new curiosity about whether these instruction-tuned models still show such significant improvements under a new problem solving scenario. In this section, we explore this through our new benchmarks.

The instruction-tuned models demonstrate only marginal improvements compared to the base models on self-invoking code generation. In Figure 3, we plot the previous reported HumanEval (or MBPP) scores against the results on HumanEval Pro and MBPP Pro (HumanEval+ and MBPP+). From the Figure 3, we have an interesting finding: When observing the correlation between HumanEval (or MBPP) and HumanEval Pro (or MBPP Pro), we see that the orange dot (indicates base model) is always to the upper left of the blue dot (indicates instruction-tuned model). However, for the comparison between HumanEval (or MBPP) and HumanEval+ (or MBPP+), the blue dot is always distributed to the upper of orange dot (even in a line on HumanEval vs HumanEval+). Overall, this suggests that while instruction-based fine-tuning significantly improves performance on simpler benchmarks like HumanEval (+) (or MBPP (+)), its efficiency diminishes for more complex self-invoking code generation tasks. On the other hand, base models like Qwen2.5-Coder-base and Deepseek-Coder-base have a higher

$$\text{Ratio} = \frac{\text{pass@k on HumanEval Pro (or MBPP Pro)}}{\text{pass@k on HumanEval (or MBPP)}} \quad (1)$$

than instruct models, which indicates that they have elevated training potential on self-invoking code generation task.

4.2 Confusion Matrix Correlation for Different Models

From Table 2, we observe that most LLMs have a score gap between direct code generation and self-invoking code generation tasks. To better understand the correlation and overlap between these two

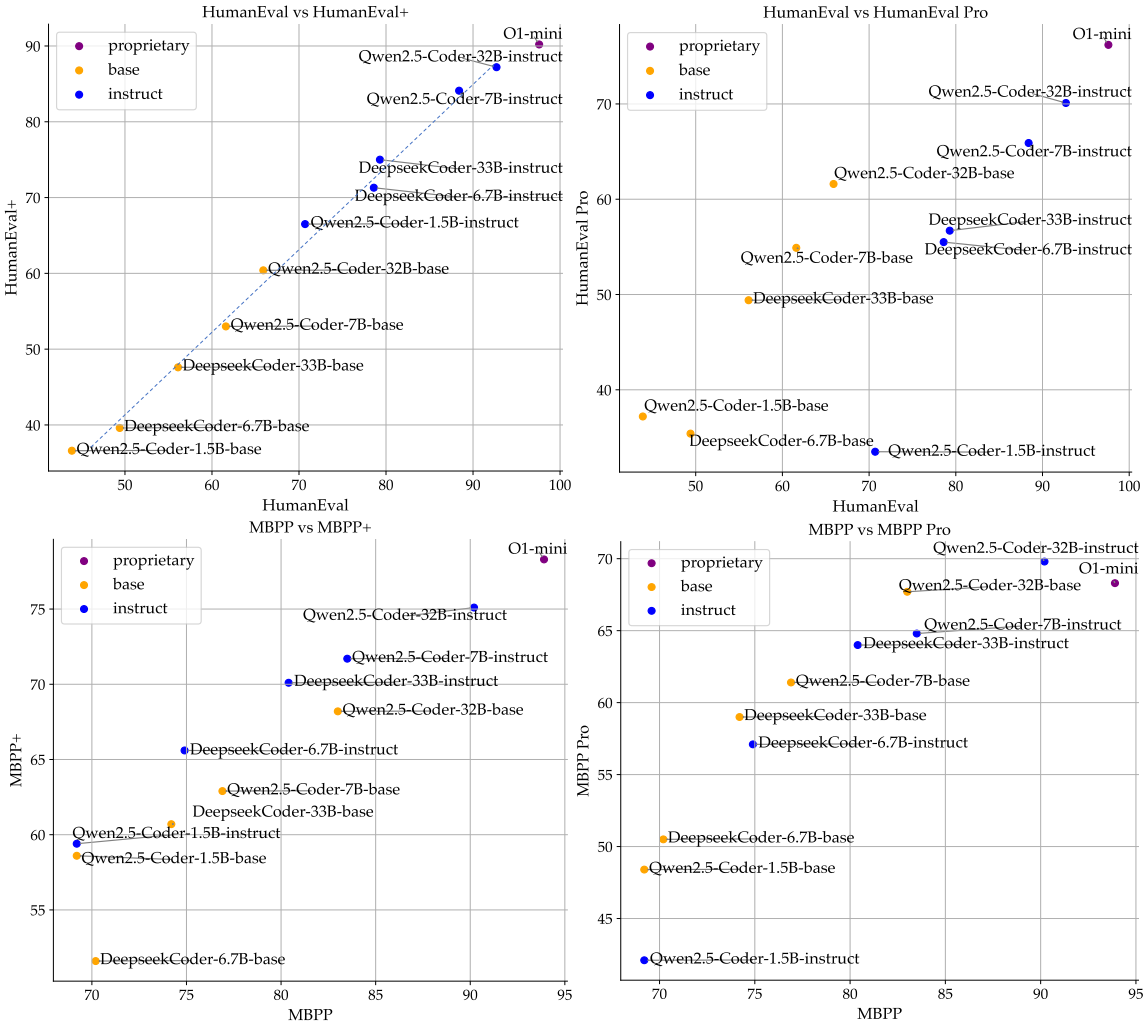


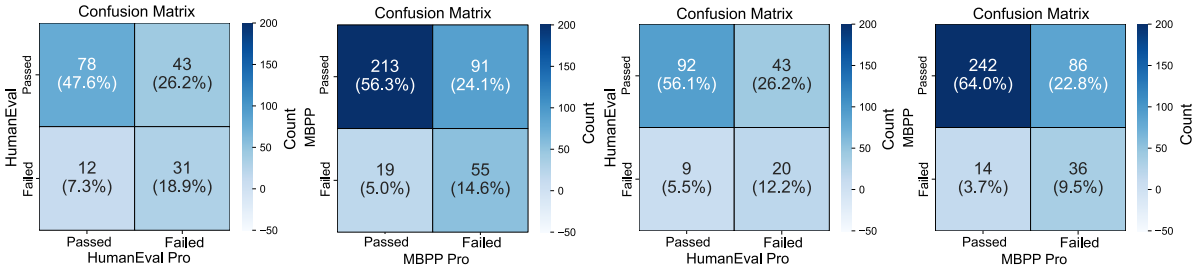
Figure 3: HumanEval (or MBPP) scores against the results on HumanEval Pro and MBPP Pro (HumanEval+ and MBPP+). We presents the comparison between base model and instruct model.

kinds of tasks, we compare the number of problems passed and failed in HumanEval Pro and MBPP Pro with their corresponding base problems in HumanEval and MBPP. Figure 4 presents an array of confusion matrix over problems, highlighting the following observation:

The instruction-tuned model does not significantly outperform the base model in self-invoking code generation task. Although some SoTA LLMs such as Qwen2.5-Coder-32B-instruct successfully solve 90% of base problems on the original HumanEval and MBPP benchmarks, over 25% of problems still fail on more challenging HumanEval Pro and MBPP Pro benchmarks with self-invoking code generation (as shown in the top right of each subfigure in Figure 4). This suggests that the drop in the model's scores on HumanEval Pro and MBPP Pro is largely due to its lower accuracy in generating self-invoking code compared to direct code generation. From the confusion matrices of the base model and the instruct model in Figure 4, we can observe a trend: the instruction-tuned model typically has a significantly higher number of `(Passed, Passed)` instances compared to the base model. However, for samples that pass the base problems but fail in HumanEval Pro and MBPP Pro, i.e., `(Failed, Passed)`, the instruct model does not demonstrate notable improvement. This observation underscores our argument in Section 4.1: current instruction-based fine-tuning approaches are insufficiently effective for more complex self-invoking code generation tasks.

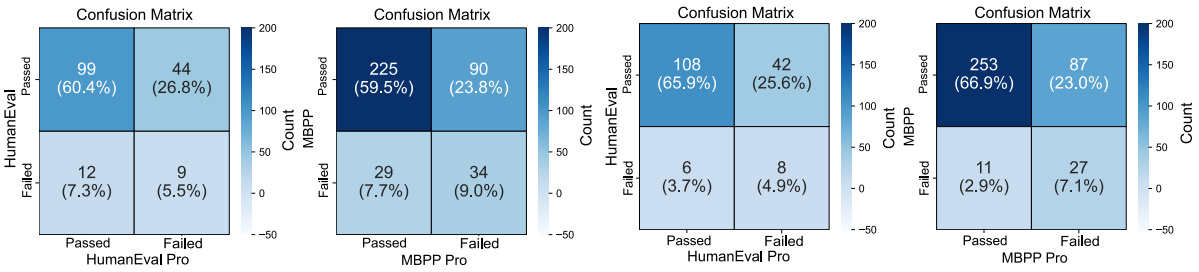
4.3 The Impact of Problem Complexity

In appendix Appendix G, we use the line counts of the canonical solution as the indicator of the problem complexity. (More complex needs longer solution code intuitively.) As shown in Figure 11,



(a) Qwen2.5-Coder-7B-base

(b) Qwen2.5-Coder-32B-base



(c) Qwen2.5-Coder-7B-instruct

(d) Qwen2.5-Coder-32B-instruct

Figure 4: The confusion matrix of different models. We use (`Failed`, `Passed`) to indicate samples that fail in HumanEval Pro (or MBPP Pro) but pass in HumanEval (or MBPP).

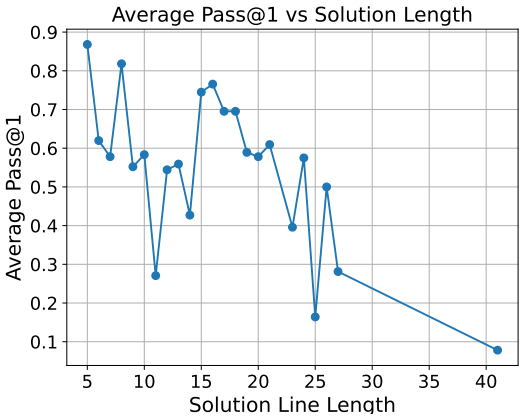


Figure 5: The Impact of Problem Difficulty.

Model	CoT	HE Pro	MBPP Pro
GPT-4o	✘	75.0	70.9
	✔	78.0	70.9
DeepseekV2.5	✘	73.8	71.2
	✔	74.4	71.4
Qwen2.5-Coder-32B-ins	✘	70.1	69.8
	✔	72.0	70.1
Qwen2.5-Coder-7B-ins	✘	65.9	64.8
	✔	71.3	64.8

Table 3: The execution error types and their descriptions in our evaluation results.

most self-invoking problems have a higher complexity than base problems. Furthermore, we analyzed the impact of the number of solution lines on the pass rate. As shown in Figure 9, longer solution lengths are associated with lower pass rates, which highlights the challenges in long self-invoking code generation.

4.4 Chain-of-Thought Prompting

To evaluate the impact of the model's reasoning ability, we evaluated the performance of GPT-4o, DeepseekV2.5, Qwen2.5-Coder-instruct (7B and 32B) with and without Chain-of-Thought (CoT) prompting (Wei et al., 2022) on HumanEval Pro and MBPP Pro. The full prompt we use is shown in Appendix H.2. For CoT prompting, we used the greedy decoding strategy for generation to align the results before. As shown in Table 3, after applying CoT, the pass@1 of the selected models on HumanEval Pro witnesses a significant improvement. Notably, the accuracy of GPT-4o increases from 75.0% to 78.0%. On MBPP Pro, although the model does not show a significant improvement, it still maintains its original performance level, indicating that CoT can enhance the accuracy of model-generated code to a notable degree.

CoT could help Code LLMs to generate more reliable code when scheduling across multiple code-related problems. To further study which aspects of code LLM can be improved by CoT, we use Python to run the code generated by GPT4o with and without CoT, and present the number of

Error Type	Description	Examples
AssertionError	Failing to pass the test cases.	Examples in Appendix I.1
NameError	The code includes undefined variables.	Examples in Appendix I.2
ValueError	Unaware of the value of variables	Examples in Appendix I.3
IndexError	Array out of bounds	Examples in Appendix I.4
TypeError	Incorrect variable type usage.	Examples in Appendix I.5
Other Errors	KeyError, SyntaxError, ZeroDivisionError, IndentationError, etc.	–

Table 4: The execution error types and their descriptions in our evaluation results.

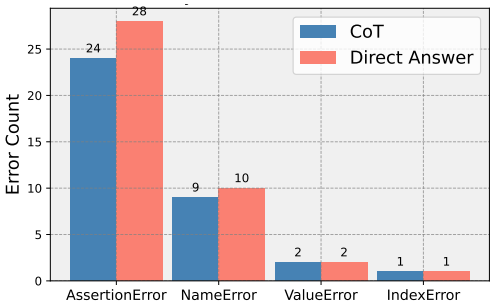


Figure 6: Error types of GPT-4o with and without CoT reasoning on HumanEval Pro.

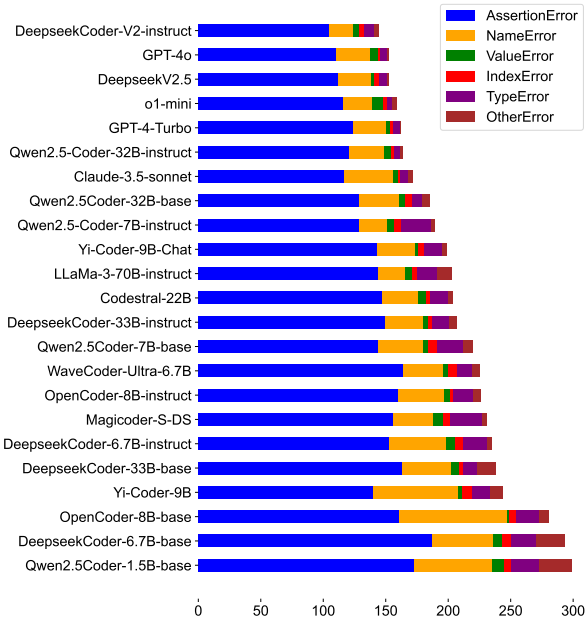


Figure 7: Statistics of error type across different LLMs on HumanEval Pro and MBPP Pro. We sum up all kinds of errors on the two benchmarks. Exact number is shown in Appendix J.

all error types that occurred in Figure 6. We have two main observations: (1) With CoT prompting, the *AssertionError* number decreases from 28 to 24. This indicates that CoT prompting enables the model to generate code that more frequently passes test cases. (2) The *NameError* number decreases, which indicates that CoT prompting helps the model produce more self-contained code snippets and reduces the use of undefined variables. These findings highlight that CoT prompting could help LLMs to generate more accurate and reliable solution on self-invoking code generation task.

4.5 Error Analysis

In order to further understand the failure modes across different LLMs, we analyze the errors encountered in code generated by different LLMs for HumanEval Pro and MBPP Pro problems and categorize them by error type. The result is shown in Figure 7. Primarily, *AssertionErrors* constitute the primary source of errors for all models on self-invoking code generation task, which suggests that the majority of errors are still due to failing test cases. Secondly, the *NameErrors*, which is often caused by the undefined variable or function, contribute significantly to the error rate. This suggests that despite the function infomation being provided in the prompt, many functions still fail to generate the correct function header. This may indicate that the LLM has issues with understanding or correctly utilizing the provided information. Finally, we also found that some *TypeErrors* and *ValueErrors* accounted for a relatively small proportion of errors, which shows that LLM still has some deficiencies in handling variable types and usage when generating self-invoking code.

5 Generalization Study of Self-invoking Code Generation

5.1 BigCodeBench-Lite Pro Benchmark

To study self-invoking code generation on a wider range of programming problems, we construct BigCodeBench-Lite Pro, a small self-invoking code generation benchmark derived from BigCodeBench (Zhuo et al., 2024). We first construct the BigCodeBench-Lite benchmark by selecting 57 problems with solve rate between 50% and

70% from BigCodeBench[1]. For each examples in BigCodeBench-Lite, we then curate the corresponding self-invoking problem as well as test cases, following the same procedure described in Section 3. After further filtering by human experts, BigCodeBench-Lite Pro contains 57 self-invoking programming problems from different topics.

Model	BCB-Lite	Pro (%)
GPT-4o	64.9	52.6
GPT4-Turbo	61.4	52.6
Claude-3.5-sonnet	73.7	50.9
DeepseekV2.5	80.7	50.9
Qwen2.5Coder-1.5B-base	50.9	15.8
Qwen2.5Coder-1.5B-instruct	50.9	10.5
OpenCoder-8B-base	56.1	10.5
OpenCoder-8B-instruct	75.4	22.8
DeepseekCoder-6.7B-base	59.6	35.1
DeepseekCoder-6.7B-instruct	56.1	35.1
WaveCoder-Ultra-6.7B	61.4	26.3
Magicoder-S-DS-6.7B	50.9	33.3
Yi-Coder-9B	57.9	21.1
Yi-Coder-9B-Chat	66.7	31.6
Qwen2.5Coder-7B-base	59.6	38.6
Qwen2.5Coder-7B-instruct	64.9	35.1
DeepseekCoder-33B-base	71.9	38.6
DeepseekCoder-33B-instruct	80.7	43.9
Qwen2.5Coder-32B-base	68.4	49.1
Qwen2.5Coder-32B-instruct	80.7	52.6
Codestral-22B	78.9	54.4
QwQ-32B-preview	86.0	59.6

Table 5: Passing rate (%) of LLMs on BigCodeBench (BCB)-Lite and BCB-Lite-Pro. A dataset example of BCB-Lite-Pro is shown in Appendix I.6.

5.2 Results Analysis

We evaluate a set of LLMs on BigCodeBench-Lite Pro. Table 5 presents the results (pass@1) of various Proprietary and Open-source LLMs, highlighting the following observations: (1) Although the base problems we selected has a solving rate of between 50% and 70% on BigCodeBench, only a small number of models in Table 5 have a passing rate of more than 50% on BigCodeBench-Lite Pro. This highlights the difficulty of the self-invoking code generation task. (2) The instruction-tuned models still demonstrate marginal improvements (sometimes decrease) compared to base models, which also reinforces our argument in Section 4.1. (3) On Bigcodebench-Lite Pro, LLMs show consistent performance trend with HumanEval Pro and MBPP Pro, which emphasizes the generalizability of our construction pipeline. Therefore, our benchmark construction approach can also be extended to adapt other code generation benchmarks, particularly as the capabilities of LLMs advance and older benchmarks become obsolete.

[1]We use reported statistics in https://huggingface.co/datasets/bigcode/bigcodebench-solve-rate.

6 Conclusion

We present HumanEval Pro and MBPP Pro, a series of benchmarks to evaluate LLMs on self-invoking code generation task where the LLMs are employed to solve the base problem and use its solution to address more complex problems. Our evaluation of over 20 LLMs reveals that, despite notable advancements in traditional code generation, these models still face challenges in self-invoking code generation. HumanEval Pro and MBPP Pro are positioned to serve as valuable benchmarks for code-related evaluations and to inspire future LLM development by shedding light on current model shortcomings and encouraging innovation in training recipe.

Limitations

In this paper, we present HumanEval Pro and MBPP Pro, a series of benchmarks evaluate LLMs on self-invoking code generation task. One limitation is that the programming languages of our benchmarks only includes Python due to the intrinsic limitation of original HumanEval and MBPP. Secondly, although the models have shown shortcomings in the self-invoking problem, the diversity of existing self-invoking problems in HumanEval Pro and MBPP Pro is still subject to the constraints of the original problems. Hence, future work should pay more attention to more diverse and multi-lingual self-invoking problem benchmarks.

Acknowledgement

Zhaojian Yu and Xiao-Ping Zhang are with the Shenzhen Key Laboratory of Ubiquitous Data Enabling Laboratory, Shenzhen International Graduate School, Tsinghua University, Shenzhen 518055, China and supported by Shenzhen Ubiquitous Data Enabling Key Lab under grant ZDSYS20220527171406015, and by Tsinghua Shenzhen International Graduate School-Shenzhen Pengrui Endowed Professor- ship Scheme of Shenzhen Pengrui Foundation.

References

01.AI. 2024. Meet yi-coder: A small but mighty llm for code.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*.

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*.

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

DeepSeek-AI. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *Preprint*, arXiv:2405.04434.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. In *Forty-first International Conference on Machine Learning*.

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

Patrick Haller, Jonas Golde, and Alan Akbik. 2024. Pecc: Problem extraction and coding challenges. *arXiv preprint arXiv:2404.18766*.

Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2022. Fixeval: Execution-based evaluation of program fixes for competitive programming problems.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, et al. 2024. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*.

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

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020*.

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*.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263*.

Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. 2024. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

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

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

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*.

Mistral. 2024. Codestral.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations*.

Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, Shafiq Joty, Yingbo Zhou, Dragomir Radev, Arman Cohan, and Arman Cohan.

2024. L2CEval: Evaluating language-to-code generation capabilities of large language models. *Transactions of the Association for Computational Linguistics*, 12:1311–1329.

OpenAI. 2024a. Gpt-4o.

OpenAI. 2024b. Openai o1 system card.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

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

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pages 31693–31715. PMLR.

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

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning*.

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. *arXiv preprint*.

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

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5140–5153.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. A survey on language models for code.

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

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

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*.

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

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

Appendix Contents

A Impact of Data Generating Model

In the early phases of benchmark construction, we use the open-source model DeepseekV2.5 to generate self-invoking problems due to its license availability. However, we were concerned that using DeepseekV2.5 to generate the benchmark would give the model an unfair advantage. Therefore, we checked the performance of a few models when generating data with GPT-4o. The results are shown in Table 7. We compute the Pearson and Spearman correlation coefficients for two different data generating models. From Table 6, we observe the strong correlations (>0.9) for both coefficients, suggesting the robustness of our benchmarks across different data generating models. In addition, as shown in Table 7, the evaluation results of GPT-4o under the data models of DeepseekV2.5 is higher than under itself. These results provide some evidence that evaluating a model on its own generated data does not seem to provide it a significant advantage.

Pearson Correlation	Spearman Correlation
0.955	0.917

Table 6: Results Correlation of Different Data Generating Models.

Evaluating Model	Data Model (DeepseekV2.5)	Data Model (GPT-4o)
GPT-4o	75.0	73.8
Qwen2.5-Coder-7B-base	54.9	54.9
Qwen2.5-Coder-7B-instruct	65.9	65.9
Qwen2.5-Coder-32B-base	61.6	56.1
Qwen2.5-Coder-32B-instruct	70.1	76.8
Deepseek-Coder-6.7B-base	35.4	37.8
Deepseek-Coder-6.7B-instruct	55.5	60.4
Deepseek-Coder-33B-base	49.4	51.2
Deepseek-Coder-33B-instruct	56.7	58.5

Table 7: Impact of Data Generating Model.

B Example in Benchmark Construction

Base Problem

```
def eat(number, need, remaining):
"""
You're a hungry rabbit, and you already have eaten a certain number of carrots,
but now you need to eat more carrots to complete the day's meals.
you should return an array of [ total number of eaten carrots after your meals,
the number of carrots left after your meals ]
if there are not enough remaining carrots, you will eat all remaining carrots, but will
still be hungry.
Example:
* eat(5, 6, 10) -> [11, 4]
* eat(4, 8, 9) -> [12, 1]
* eat(1, 10, 10) -> [11, 0]
* eat(2, 11, 5) -> [7, 0]
Variables:
@number : integer
the number of carrots that you have eaten.
@need : integer
the number of carrots that you need to eat.
@remaining : integer
the number of remaining carrots thet exist in stock
Constrain:
* 0 <= number <= 1000
* 0 <= need <= 1000
* 0 <= remaining <= 1000
Have fun :)
"""
if(need <= remaining):
    return [ number + need , remaining-need ]
else:
    return [ number + remaining , 0]
```

Self-invoking Problem

```
# You are a farmer who needs to feed a group of hungry rabbits. Each rabbit has
a specific number of carrots it has already eaten and a specific number it still
needs to eat. You have a limited number of carrots in stock. Write a function that
takes in a list of rabbits, where each rabbit is represented by a tuple (number,
need), and the total number of carrots in stock. The function should return the
total number of carrots eaten by all rabbits and the number of carrots left in stock
after feeding all the rabbits.
```

Canonical Solution

```
def feed_rabbits(rabbits, stock):

    total_eaten = 0
    remaining_carrots = stock

    for rabbit in rabbits:
        number, need = rabbit
        eaten, remaining_carrots = eat(number, need, remaining_carrots)
        total_eaten += eaten + number

    return [total_eaten, remaining_carrots]
```

Test Cases

```
assert feed_rabbits([(5, 6), (4, 8), (1, 10)], 25) == [24, 1]
assert feed_rabbits([(2, 11), (3, 5), (4, 7)], 20) == [20, 0]
assert feed_rabbits([(0, 5), (5, 5), (10, 5)], 30) == [15, 15]
assert feed_rabbits([(1, 10), (2, 11), (3, 12)], 50) == [33, 17]
```

Figure 8: An example of self-invoking problems in HumanEval Pro

C Detailed Results

Model	**HumanEval Pro** (0-shot)	**MBPP Pro** (0-shot)
LLaMA-3.1-8B-base	25.0	36.5
LLaMA-3.1-8B-instruct	45.7	53.7
LLaMA-3.1-70B-base	40.9	57.4
LLaMA-3.1-70B-instruct	60.4	63.8
Qwen-2.5-72B-base	62.2	65.3
Qwen-2.5-72B-instruct	68.9	68.8
QwQ-32B-preview	72.0	67.5
LLaMA-3.3-70B-instruct	67.1	64.6
Mistral-Large-instruct-2411	75.0	69.3

Table 8: Results of Other LLMs on HumanEval Pro and MBPP Pro (greedy decoding).

Model	**HumanEval Pro**			**MBPP Pro**		
	pass@1	**pass@5**	**pass@10**	**pass@1**	**pass@5**	**pass@10**
DeepseekCoder-6.7B-base	38.0	50.9	54.7	51.6	60.4	63.1
DeepseekCoder-6.7B-instruct	55.9	64.1	66.5	55.2	62.6	64.9
Magicoder-S-DS-6.7B	55.1	62.7	65.1	57.7	64.9	67.2
WaveCoder-Ultra-6.7B	55.7	61.4	63.0	58.2	64.4	66.3
DeepseekCoder-33B-base	49.4	60.8	65.2	59.1	67.2	69.3
DeepseekCoder-33B-instruct	59.1	68.6	71.3	63.4	70.6	72.9
Qwen2.5-Coder-7B-base	51.8	62.1	66.2	61.3	69.9	72.3
Qwen2.5-Coder-7B-instruct	65.7	72.5	75.0	64.2	70.5	72.6
OpenCoder-9B-base	44.5	56.2	59.9	54.8	62.9	65.0
OpenCoder-9B-instruct	59.8	68.5	70.8	58.1	63.7	65.1
Yi-Coder-9B-base	47.9	59.0	61.9	59.6	67.7	69.7
Yi-Coder-9B-chat	59.7	66.4	67.9	65.0	69.8	71.2
Codestral-22B	59.5	66.2	67.7	63.2	67.7	68.9
Qwen2.5-Coder-32B-base	62.4	70.3	72.2	67.6	75.0	76.9
Qwen2.5-Coder-32B-instruct	69.2	72.3	73.3	70.6	74.7	76.0
QwQ-32B-preview	70.9	77.7	79.5	67.0	73.0	74.5

Table 9: The results of different models on HumanEval Pro and MBPP Pro . We generate 20 samples for each problems with random sampling strategy where temperature is set to 0.2 and top_p is set to 0.95.

D Fine-tuning Experiments

In order to analyze the impact of specifically fine-tuning on self-invoking problems. We use GPT4o to generate a dataset containing 20K self-invoking problems and solutions and fine-tune Qwen2.5-Coder-7B-Base (Hui et al., 2024). Compared to Qwen2.5-Coder-7B-Base in Figure 4 (a), who has 26.2% (Failed, Pass) (i.e., samples that pass the base problems but fail in HumanEval Pro and MBPP Pro) samples on HumanEval Pro and 24.1% on MBPP Pro, the fine-tuned model didn't show significant advantages on HumanEval Pro (23.8% (Failed, Pass) samples) and MBPP Pro (28.0% (Failed, Pass) samples).

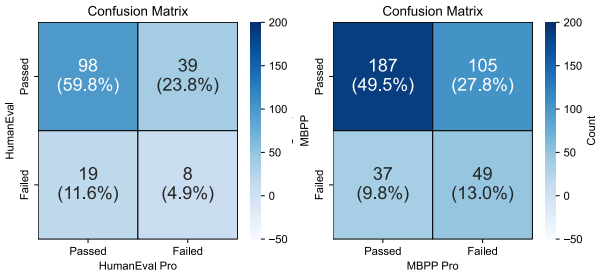


Figure 9: The confusion matrix of fine-tuned model on self-invoking problems.

E Model Information

We present results of proprietary models and open-source models on HumanEval Pro and MBPP Pro: Qwen-2.5-Coder (Base and Instruct, 1.5B, 7B, 33B) (Hui et al., 2024), DeepseekCoder (Base and Instruct) (Guo et al., 2024), DeepseekCoder-V2 (DeepSeek-AI, 2024), Yi-Coder-9B (Base and Instruct) (01.AI, 2024), OpenCoder (Base and instruct) (Huang et al., 2024), Magicoder-S-DS-6,7B (Wei et al., 2024), WaveCoder-Ultra-6.7B (Yu et al., 2024), Codestral-22B (Mistral, 2024), GPT-3.5 (Ouyang et al., 2022), GPT-4o (OpenAI, 2024a), Claude-3.5-sonnet (Anthropic, 2024) and o1-mini (OpenAI, 2024b). To facilitate reproducibility, the HuggingFace checkpoints of all open-source models and API name of proprietary models are provided in Appendix E. Our prompts for evaluation is shown in Appendix H.2.

Model Name	API Name
O1-mini	`o1-mini-2024-09-12`
GPT-4o	`gpt-4o-2024-08-06`
GPT-4-Turbo	`gpt-4-turbo-2024-04-09`
Claude-3.5-sonnet	`claude-3-5-sonnet-20241022`
Deepseek-V2.5	`deepseek-chat`

Model Name	HuggingFace URL
DeepseekCoder-V2-instruct	https://huggingface.co/deepseek-ai/DeepSeek-Coder-V2-Instruct
Qwen2.5-Coder-1.5B-base	https://huggingface.co/Qwen/Qwen2.5-Coder-1.5B
Qwen2.5-Coder-1.5B-instruct	https://huggingface.co/Qwen/Qwen2.5-Coder-1.5B-Instruct
DeepseekCoder-6.7B-base	https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-base
DeepseekCoder-6.7B-instruct	https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct
Magicoder-S-DS-6.7B	https://huggingface.co/ise-uiuc/Magicoder-S-DS-6.7B
WaveCoder-Ultra-6.7B	https://huggingface.co/microsoft/wavecoder-ultra-6.7b
Qwen2.5-Coder-7B-base	https://huggingface.co/Qwen/Qwen2.5-Coder-7B
Qwen2.5-Coder-7B-instruct	https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct
OpenCoder-8B-base	https://huggingface.co/infly/OpenCoder-8B-Base
OpenCoder-8B-instruct	https://huggingface.co/infly/OpenCoder-8B-Instruct
Yi-Coder-9B-base	https://huggingface.co/01-ai/Yi-Coder-9B
Yi-Coder-9B-chat	https://huggingface.co/01-ai/Yi-Coder-9B-Chat
Codestral-22B-v0.1	https://huggingface.co/mistralai/Codestral-22B-v0.1
DeepseekCoder-33B-base	https://huggingface.co/deepseek-ai/deepseek-coder-33b-base
DeepseekCoder-33B-instruct	https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct
Qwen2.5-Coder-32B-base	https://huggingface.co/Qwen/Qwen2.5-Coder-32B
Qwen2.5-Coder-32B-instruct	https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct
LLaMA3-70B-instruct	https://huggingface.co/meta-llama/Meta-Llama-3-70B-Instruct
QwQ-32B-Preview	https://huggingface.co/Qwen/QwQ-32B-Preview
LLaMA3.1-8B-base	https://huggingface.co/meta-llama/Llama-3.1-8B
LLaMA3.1-8B-instruct	https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct
LLaMA3.1-70B-base	https://huggingface.co/meta-llama/Llama-3.1-70B
LLaMA3.1-70B-instruct	https://huggingface.co/meta-llama/Llama-3.1-70B-Instruct
Qwen2.5-72B-base	https://huggingface.co/Qwen/Qwen2.5-72B
Qwen2.5-72B-instruct	https://huggingface.co/Qwen/Qwen2.5-72B-Instruct

Table 10: The corresponding API names and HuggingFace model URLs for the evaluated models are listed in Table 2.

F Comparison between HumanEval (Pro), MBPP (Pro) and BigCodeBench-Lite (Pro)

Figure 10: Comparison between HumanEval Family, MBPP Family and BigCodeBench-Lite Family.

G Difficulty of Benchmark Samples

We analyze the complexity comparison between a base problem and its self-invoking counterpart by examining the line counts of their canonical solutions. The line count serves as a proxy for the complexity of each problem. By comparing the number of lines required to solve the base problem with those needed for the self-invoking version, we gain insight into how the introduction of self-invocation affects the overall complexity. Generally, self-invoking problems, which often involve recursion or similar constructs, may require more lines of code to handle additional logic and edge cases, thereby increasing the complexity. This comparison helps in understanding the additional computational and conceptual challenges introduced by self-invocation.

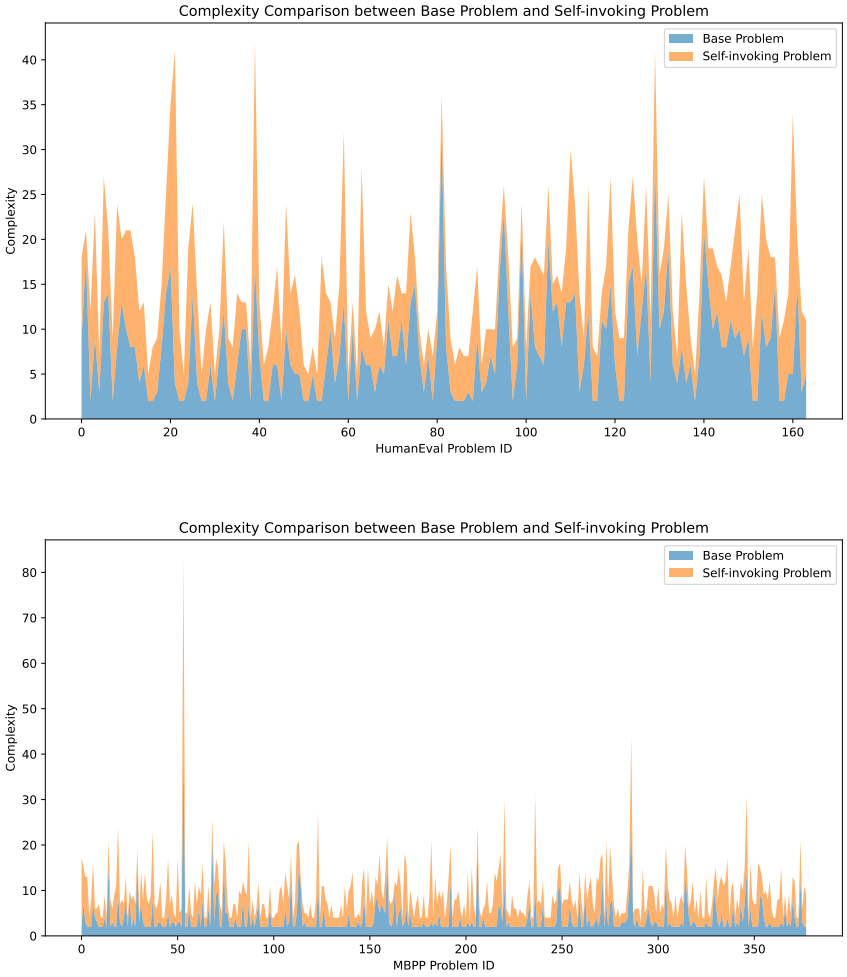


Figure 11: Complexity comparison between base problem and self-invoking problem. We use the line count of the canonical solution for both the base problem and the self-invoking problem as a measure of the problem's complexity.

H Prompts

H.1 Prompts for Benchmark Construction

We set the prompt in our benchmark construction as follows:

Prompt for benchmark construction:
I'll give you a raw programming question and its solution, please generate a new problem that requires multiple calls to the raw question to solve, and generate the solution in new_solution.
Please return with json format including 3 keys: 'new_problem','new_solution', 'test_input, I'll use json.loads() to transform it to dict type.
To solve new_problems, new_solution should include the multiple function calls of raw question.
So new_problems will be not only a related problem but also a more complex problem than raw problem.
raw problem:
{raw problem}
raw solution:
{raw solution}

H.2 Prompts for Evaluation

We set the prompt in our evaluation as follows:

Prompt of 0-shot: You are an exceptionally intelligent coding assistant that consistently delivers accurate and reliable responses to user instructions. Write a solution of python file to the following problems, the solution of the second problem requires single or multiple calls to the first
@@ Instruction
{base problem}
{self-invoking problem}
@@ Response

Prompt of 1-shot: You are an exceptionally intelligent coding assistant that consistently delivers accurate and reliable responses to user instructions. Write a solution of python file to the following problems, the solution of the second problem requires single or multiple calls to the first solution
@@ Instruction
{base problem}
{self-invoking problem}
{example}
@@ Response

Prompt with Chain-of-Thought:
You are an exceptionally intelligent coding assistant that consistently delivers accurate and reliable responses to user instructions. Write a solution of python file to the following problems, the solution of the second problem requires single or multiple calls to the first
@@ Instruction
{base problem}
{self-invoking problem}
@@ Response
Let's think step by step.

I Examples of Different Error Types

I.1 Examples of AssertionError

Listing 1: AssertionError in HumanEval/24-generated by Qwen2.5Coder-7B-base

```
Status: AssertionError
Traceback: assert largest_common_divisor([15, 30, 45]) == 15
Explanation: expected 15 but got 5.
```

```
def largest_divisor(n: int) -> int:
    """ For a given number n, find the largest number that divides n
       evenly, smaller than n
    >>> largest_divisor(15)
    5
    """
    for i in range(n-1, 0, -1):
        if n % i == 0:
            return i
    return 1

def largest_common_divisor(numbers: list) -> int:
    """ For a given list of integers, find the largest number that
       divides each of the numbers in the list evenly, smaller than
       the smallest number in the list
    >>> largest_common_divisor([15, 30, 45])
    15
    """
    smallest = min(numbers)
    for i in range(smallest-1, 0, -1):
        if all(n % i == 0 for n in numbers):
            return i
    return 1


assert largest_common_divisor([15, 30, 45]) == 15
assert largest_common_divisor([15, 30, 45]) == 15
assert largest_common_divisor([10, 20, 30]) == 10
assert largest_common_divisor([7, 14, 21]) == 7
assert largest_common_divisor([16, 32, 64]) == 16
assert largest_common_divisor([9, 18, 27]) == 9
```

I.2 Examples of NameError

Listing 2: NameError in HumanEval/14-generated by Qwen2.5Coder-7B-instruct

```
Status: NameError
Traceback: assert all_prefixes_for_list(['abc', 'def']) == [['a', 'ab
   ', 'abc'], ['d', 'de', 'def']]
Explanation: The expected function name is 'all_prefixes_for_list',
   but 'all_prefixes_of_strings' is generated though this information
    is given in prompt.
```

```
from typing import List
def all_prefixes(string: str) -> List[str]:
    """ Return list of all prefixes from shortest to longest of the
       input string
```

```
    >>> all_prefixes('abc')
    ['a', 'ab', 'abc']
    """
    return [string[:i] for i in range(1, len(string) + 1)]


def all_prefixes_of_strings(strings: List[str]) -> List[List[str]]:
    """ Return list of lists where each sublist contains all prefixes
        of the corresponding string in the input list, sorted from
       shortest to longest. If the input list is empty, return an
       empty list.
    >>> all_prefixes_of_strings(['abc', 'def', 'ghi'])
    [['a', 'ab', 'abc'], ['d', 'de', 'def'], ['g', 'gh', 'ghi']]
    """
    return [all_prefixes(s) for s in strings]



assert all_prefixes_for_list(['abc', 'def']) == [['a', 'ab', 'abc'],
   ['d', 'de', 'def']]
assert all_prefixes_for_list(['']) == [[]]
assert all_prefixes_for_list([]) == []
assert all_prefixes_for_list(['single']) == [['s', 'si', 'sin', 'sing
   ', 'singl', 'single']]
```

I.3 Examples of ValueError

Listing 3: ValueError in HumanEval/153-generated by O1-mini

```
Status: ValueError
Traceback:
>>> assert Strongest_Extensions({'Slices': ['SErviNGSliCes', 'Cheese'
   , 'StuFfed'], 'my_class': ['AA', 'Be', 'CC']}) == ['Slices.
   SErviNGSliCes', 'my_class.AA']
>>> for class_name, extensions in classes_with_extensions
Explanation: ValueError: too many values to unpack (expected 2)
```

```

def Strongest_Extension(class_name, extensions):
    """
    Determines the strongest extension for a given class based on the
        strength calculation.

    The strength of an extension is calculated as the number of
       uppercase letters
    minus the number of lowercase letters in the extension's name.

    If multiple extensions have the same strength, the first one in
       the list is chosen.

    Parameters:
    - class_name (str): The name of the class.
```

```
    - extensions (List[str]): A list of extension names.

    Returns:
    - str: A string in the format "ClassName.StrongestExtensionName".

    Example:
    >>> Strongest_Extension('my_class', ['AA', 'Be', 'CC'])
    'my_class.AA'
    """
    if not extensions:
        return f"{class_name}.None"

    def calculate_strength(extension):
        cap = sum(1 for char in extension if char.isupper())
        sm = sum(1 for char in extension if char.islower())
        return cap - sm

    strongest_extension = extensions[0]
    max_strength = calculate_strength(strongest_extension)

    for ext in extensions[1:]:
        strength = calculate_strength(ext)
        if strength > max_strength:
            strongest_extension = ext
            max_strength = strength
    return f"{class_name}.{strongest_extension}"

def Strongest_Extensions(classes_with_extensions):
    """
    Determines the strongest extensions for a list of classes.

    For each class, it finds the strongest extension based on the
       strength calculation.
    If a class has no extensions, it is represented as "ClassName.
       None".

    Parameters:
    - classes_with_extensions (List[Tuple[str, List[str]]]):
      A list where each element is a tuple containing a class name
         and a list of its extensions.

    Returns:
    - List[str]: A list of strings in the format "ClassName.
       StrongestExtensionName".

    Example:
    >>> Strongest_Extensions([
    ...     ('my_class', ['AA', 'Be', 'CC']),
    ...     ('Slices', ['SErviNGSliCes', 'Cheese', 'StuFfed']),
    ...     ('EmptyClass', [])
    ... ])
    ['my_class.AA', 'Slices.SErviNGSliCes', 'EmptyClass.None']
```

```
    """
    result = []
    for class_name, extensions in classes_with_extensions:
        if extensions:
            strongest = Strongest_Extension(class_name, extensions)
        else:
            strongest = f"{class_name}.None"
        result.append(strongest)
    return result

assert Strongest_Extensions({'Slices': ['SErviNGSliCes', 'Cheese', 'StuFfed'], 'my_class': ['AA', 'Be', 'CC']}) == ['Slices.SErviNGSliCes', 'my_class.AA']
assert Strongest_Extensions({'Empty': [], 'Single': ['A']}) == ['Empty.None', 'Single.A']
assert Strongest_Extensions({'Mixed': ['aA', 'Bb', 'Cc'], 'AllCaps': ['AA', 'BB', 'CC']}) == ['Mixed.aA', 'AllCaps.AA']
assert Strongest_Extensions({'NoExtensions': []}) == ['NoExtensions.None']
assert Strongest_Extensions({}) == []  # Check for empty input
```

I.4 Examples of IndexError

Listing 4: IndexError in HumanEval/73-generated by GPT-4o

```
Status: IndexError
Traceback:
>>> assert smallest_change_2d([[1,2,3,5,4,7,9,6], [1,2,3,4,3,2,2], [1,2,3,2,1]]) == 7
>>> column = [arr[row][col] for row in range(rows)]
Explanation: IndexError: list index out of range
```

```

def smallest_change(arr):
    """
    Given an array arr of integers, find the minimum number of elements that
    need to be changed to make the array palindromic. A palindromic array is an array that
    is read the same backwards and forwards. In one change, you can change one element to any other element.
    """
    n = len(arr)
    changes = 0
    for i in range(n // 2):
        if arr[i] != arr[n - i - 1]:
            changes += 1
    return changes

def smallest_change_2d(arr):
    """
```

```
    Given a 2D array of integers, find the minimum number of elements
        that need to be changed to make each row
    and each column of the array palindromic. A palindromic row or
       column is a row or column that is read the same
    backwards and forwards. In one change, you can change one element
        to any other element.
    """
    if not arr or not arr[0]:
        return 0

    rows = len(arr)
    cols = len(arr[0])

    # Calculate changes needed for each row
    row_changes = sum(smallest_change(row) for row in arr)

    # Calculate changes needed for each column
    col_changes = 0
    for col in range(cols):
        column = [arr[row][col] for row in range(rows)]
        col_changes += smallest_change(column)

    return row_changes + col_changes

assert smallest_change_2d([[1,2,3,5,4,7,9,6], [1,2,3,4,3,2,2],
   [1,2,3,2,1]]) == 7
assert smallest_change_2d([[1,2,3,4], [4,3,2,1], [1,2,2,1],
   [1,2,3,4]]) == 8
assert smallest_change_2d([[1,2,3,2,1], [1,2,3,2,1], [1,2,3,2,1],
   [1,2,3,2,1]]) == 0
```

I.5 Examples of TypeError

Listing 5: TypeError in MBPP/164-generated by DeepseekCoder-V2-instruct

```
Status: TypeError
Traceback:
>>> assert total_and_average_volume([{'radius': 3, 'height': 5}, {'
   radius': 4, 'height': 6}]) == (147.6548547187203,
   73.82742735936014)
>>> total_volume += volume_cone(r, h)
>>> return (math.pi * r**2 * h) / 3

Explanation: TypeError: unsupported operand type(s) for ** or pow():
   'str' and 'int'
```

```
# Write a function to find the volume of a cone.
import math
def volume_cone(r,h):
    return (math.pi * r**2 * h) / 3

```

```
# Given a list of cones with their respective radii and heights,
    write a function to calculate the total volume of all cones and
    the average volume per cone.

def total_and_average_volume(cones):
    total_volume = 0
    for cone in cones:
        r, h = cone
        total_volume += volume_cone(r, h)
    average_volume = total_volume / len(cones)
    return total_volume, average_volume

assert total_and_average_volume([{'radius': 3, 'height': 5}, {'radius
    ': 4, 'height': 6}]) == (147.6548547187203, 73.82742735936014)
assert total_and_average_volume([{'radius': 1, 'height': 2}]) ==
    (2.0943951023931953, 2.0943951023931953)
```

I.6 An Example of BigCodeBench-Lite Pro

Listing 6: The outputs of GPT-4o on BigCodeBench-Lite Pro (BigCodeBench/355).

```
Traceback (most recent call last):
  File "evalpro/result/GPT-4o/bigcodebench_lite_pro/log/
     santized_results/case_20/gen_0.py", line 76, in <module>
    new_solution(points)
  File "evalpro/result/GPT-4o/bigcodebench_lite_pro/log/
     santized_results/case_20/gen_0.py", line 65, in new_solution
    voronoi_plot_2d(Voronoi(subset), ax=ax, show_vertices=False,
       line_colors=f'C{i}', point_size=2)
  File "_qhull.pyx", line 2677, in scipy.spatial._qhull.Voronoi.
     __init__
  File "_qhull.pyx", line 352, in scipy.spatial._qhull._Qhull.
     __init__
scipy.spatial._qhull.QhullError: QH6214 qhull input error: not enough
    points(1) to construct initial simplex (need 4)
```

```
import numpy as np
from scipy.spatial import Voronoi, voronoi_plot_2d
import matplotlib.pyplot as plt
def task_func(points, seed=0):
    """
    Calculate the Voronoi diagram for a number of points in 2D and
       plot it.
    Note: this function will raise errors when input is invalid, for
       example wrong type or shape.
    Jittering is applied prior to plotting.

    Parameters:
    - points (np.ndarray): A numpy ndarray of shape (n_points, 2)
       with the coordinates of the points.
    - seed (int): Random seed for reproducibility. Defaults to 0.

```

```
    Returns:
    tuple (vor, ax): A tuple containing:
        - vor (Voronoi): A Voronoi object representing the Voronoi
           diagram of the points.
        - ax (Axes): The axes of the plotted Voronoi diagram.
    """
    if points.shape[1] != 2:
        raise ValueError("Input points should have shape (n_points,
           2)")

    np.random.seed(seed)
    jittered_points = points + np.random.normal(scale=0.01, size=
       points.shape)

    vor = Voronoi(jittered_points)

    fig, ax = plt.subplots()
    voronoi_plot_2d(vor, ax=ax, show_vertices=False, line_colors='
       blue', point_size=2)
    plt.show()

    return vor, ax

def new_solution(points):
    """
    Given a set of points in 2D space, calculate and plot the Voronoi
        diagram for each subset of points obtained
    by dividing the original set into three equal parts. Additionally
       , overlay the Voronoi diagrams of these subsets
    on a single plot to visualize their combined effect.

    Parameters:
    - points (np.ndarray): A numpy ndarray of shape (n_points, 2)
       with the coordinates of the points.

    Returns:
    None
    """
    if len(points) < 3:
        raise ValueError("Need at least 3 points to divide into three
           subsets")

    # Sort points array for reproducibility
    points = np.array(sorted(points, key=lambda x: (x[0], x[1])))

    # Divide the points into three equal subsets
    n_points = len(points)
    part_size = n_points // 3

    subsets = [points[i*part_size:(i+1)*part_size] for i in range(3)]

    # Create a plot for overlay
```

```
    fig, ax = plt.subplots()

    # Plot the Voronoi for each subset
    for i, subset in enumerate(subsets):
        # Note: Here `task_func` is assumed to return the `ax`, which will be reused for plotting
        voronoi_plot_2d(Voronoi(subset), ax=ax, show_vertices=False, line_colors=f'C{i}', point_size=2)

    plt.title("Overlay of Voronoi Diagrams for the Three Subsets")
    plt.show()

# Test case 1: Basic test with 9 points
points = np.array([[0, 0], [0, 1], [1, 0], [1, 1], [2, 2], [2, 3], [3, 3], [3, 4], [4, 4]])
new_solution(points)

# Test case 2: Test with exactly 3 points
points = np.array([[0, 0], [1, 1], [2, 2]])
new_solution(points)

# Test case 3: Test with random points ensuring at least 9 points
points = np.random.rand(9, 2)
new_solution(points)
```

J Error Statistics across Different Models

Model	Dataset	Error type						All
		AssertionError	NameError	ValueError	IndexError	TypeError	OtherError	
O1-mini	HumanEval Pro	27	8	3	1	0	0	39
	MBPP Pro	89	15	6	2	4	4	120
	All	**116**	**23**	**9**	**3**	**4**	**4**	**159**
GPT-4o	HumanEval Pro	28	11	2	1	0	0	41
	MBPP Pro	82	17	4	1	5	1	110
	All	**110**	**28**	**6**	**2**	**5**	**1**	**151**
DeepseekCoder-V2-instruct	HumanEval Pro	26	7	1	1	1	1	37
	MBPP Pro	79	12	4	3	7	3	108
	All	**105**	**19**	**5**	**4**	**8**	**4**	**145**
DeepseekV2.5	HumanEval Pro	30	8	2	1	2	0	43
	MBPP Pro	82	18	1	3	4	1	109
	All	**112**	**26**	**3**	**4**	**6**	**1**	**152**
Qwen2.5-Coder-32B-instruct	HumanEval Pro	32	12	2	2	1	1	50
	MBPP Pro	89	16	3	1	4	1	114
	All	**121**	**28**	**5**	**3**	**5**	**2**	**164**
Qwen2.5-Coder-7B-instruct	HumanEval Pro	36	8	3	2	6	1	56
	MBPP Pro	93	14	3	3	18	2	133
	All	**129**	**22**	**6**	**5**	**24**	**3**	**189**
Claude-3.5-sonnet	HumanEval Pro	30	11	1	1	0	2	45
	MBPP Pro	87	28	3	1	6	2	127
	All	**117**	**39**	**4**	**2**	**6**	**4**	**172**
LLaMa-3-70B-instruct	HumanEval Pro	44	10	3	2	2	4	65
	MBPP Pro	100	12	2	2	14	8	138
	All	**144**	**22**	**5**	**4**	**16**	**12**	**203**
Codestral-22B	HumanEval Pro	45	13	3	3	2	1	67
	MBPP Pro	102	16	3	1	12	3	137
	All	**147**	**29**	**6**	**4**	**14**	**4**	**204**
OpenCoder-8B-base	HumanEval Pro	47	43	0	3	5	2	100
	MBPP Pro	114	43	2	2	14	6	181
	All	**161**	**86**	**2**	**5**	**19**	**8**	**281**
OpenCoder-8B-instruct	HumanEval Pro	42	15	2	1	5	2	67
	MBPP Pro	118	22	3	1	11	4	159
	All	**160**	**37**	**5**	**2**	**16**	**6**	**226**
Qwen2.5Coder-1.5B-base	HumanEval Pro	56	25	7	1	9	5	103
	MBPP Pro	117	37	3	4	14	21	196
	All	**173**	**62**	**10**	**5**	**23**	**26**	**299**
Qwen2.5Coder-7B-base	HumanEval Pro	45	15	3	4	5	2	74
	MBPP Pro	99	21	1	3	16	6	146
	All	**144**	**36**	**4**	**7**	**21**	**8**	**220**
Qwen2.5Coder-32B-base	HumanEval Pro	39	15	3	3	1	2	63
	MBPP Pro	90	17	2	2	7	4	122
	All	**129**	**32**	**5**	**5**	**8**	**6**	**185**
Yi-Coder-9B	HumanEval Pro	48	31	2	5	3	5	94
	MBPP Pro	92	37	1	3	12	5	150
	All	**140**	**68**	**3**	**8**	**15**	**10**	**244**
Yi-Coder-9B-Chat	HumanEval Pro	47	12	1	3	3	0	66
	MBPP Pro	96	19	1	2	11	4	133
	All	**143**	**31**	**2**	**5**	**14**	**4**	**199**
GPT-4-Turbo	HumanEval Pro	33	8	3	1	1	0	46
	MBPP Pro	91	18	1	1	5	0	116
	All	**124**	**26**	**4**	**2**	**6**	**0**	**162**
DeepseekCoder-33B-base	HumanEval Pro	55	16	2	2	3	5	83
	MBPP Pro	108	23	5	1	8	10	155
	All	**163**	**39**	**7**	**3**	**11**	**15**	**238**
DeepseekCoder-33B-instruct	HumanEval Pro	49	14	2	2	4	0	71
	MBPP Pro	101	16	2	1	10	6	136
	All	**150**	**30**	**4**	**3**	**14**	**6**	**207**
DeepseekCoder-6.7B-base	HumanEval Pro	59	24	4	4	6	9	106
	MBPP Pro	128	25	3	3	14	14	187
	All	**187**	**49**	**7**	**7**	**20**	**23**	**293**
DeepseekCoder-6.7B-instruct	HumanEval Pro	46	15	4	4	2	2	73
	MBPP Pro	107	30	4	2	17	2	162
	All	**153**	**45**	**8**	**6**	**19**	**4**	**235**
Magicoder-S-DS	HumanEval Pro	49	11	6	4	5	0	75
	MBPP Pro	107	21	2	2	20	4	156
	All	**156**	**32**	**8**	**6**	**25**	**4**	**231**
WaveCoder-Ultra-6.7B	HumanEval Pro	51	12	2	3	4	2	74
	MBPP Pro	113	20	2	4	8	4	151
	All	**164**	**32**	**4**	**7**	**12**	**6**	**225**

Table 11: Error type of Different Models on HumanEval Pro and MBPP Pro.