BIGCODEBENCH: BENCHMARKING CODE GENERATION WITH DIVERSE FUNCTION CALLS AND COMPLEX INSTRUCTIONS

Terry Yue Zhuo[1,2,3] **Minh Chien Vu**[4] **Jenny Chim**[5] **Han Hu**[1,3] **Wenhao Yu**[12]
Ratnadira Widyasari[3] **Imam Nur Bani Yusuf**[3] **Haolan Zhan**[1] **Junda He**[3]
Indraneil Paul[7] **Simon Brunner**[8] **Chen Gong**[9] **Thong Hoang**[2] **Armel Zebaze**[10]
Xiaoheng Hong[11] **Wen-Ding Li**[13] **Jean Kaddour**[14] **Ming Xu**[19] **Zhihan Zhang**[6]
Prateek Yadav[15] **Naman Jain**[16] **Alex Gu**[17] **Zhoujun Cheng**[18] **Jiawei Liu**[20]
Qian Liu[21] **Zijian Wang**[22] **Binyuan Hui**[8] **Niklas Muennighoff**[23] **David Lo**[3]
Daniel Fried[24] **Xiaoning Du**[†1] **Harm de Vries**[†25] **Leandro von Werra**[†26]

Core contributors, random ordering
Additional contributors, random ordering
Senior contributors, random ordering

[1]Monash University [2]CSIRO's Data61 [3]Singapore Management University
[4]Detomo Inc., Japan [5]Queen Mary University of London [6]University of Notre Dame
[7]TU Darmstadt [8]Independent [9]University of Virginia [10]Inria [11]Intel [12]Tencent AI Lab
[13]Cornell University [14]University College London [15]UNC Chapel Hill [16]UC Berkeley
[17]MIT [18]Shanghai Jiaotong University [19]Uber [20]UIUC [21]Sea AI Lab [22]AWS AI Labs*
[23]Contextual AI [24]Carnegie Mellon University [25]ServiceNow Research [26]Hugging Face

https://bigcode-bench.github.io/
{terry.zhuo, xiaoning.du}@monash.edu
contact@bigcode-project.org

ABSTRACT

Task automation has been greatly empowered by the recent advances in Large Language Models (LLMs) via Python code, where the tasks range from software engineering development to general-purpose reasoning. While current benchmarks have shown that LLMs can solve tasks using programs like human developers, the majority of their evaluations are limited to short and self-contained algorithmic tasks or standalone function calls. Solving challenging and practical tasks requires the capability of utilizing ***diverse function calls as tools*** to efficiently implement functionalities like data analysis and web development. In addition, using multiple tools to solve a task needs compositional reasoning by accurately understanding ***complex instructions***. Fulfilling both of these characteristics can pose a great challenge for LLMs. To assess how well LLMs can solve challenging and practical tasks via programs, we introduce `BigCodeBench`, a benchmark that challenges LLMs to invoke multiple function calls as tools from 139 libraries and 7 domains for 1,140 fine-grained tasks. To evaluate LLMs rigorously, each task encompasses 5.6 test cases with an average branch coverage of 99%. In addition, we propose a natural-language-oriented variant of `BigCodeBench`, `BigCodeBench-Instruct`, that automatically transforms the original docstrings into short instructions only with essential information. Our extensive evaluation of 60 LLMs shows that ***LLMs are not yet capable of following complex instructions to use function calls precisely, with scores up to 60%, significantly lower than the human performance of 97%***. The results underscore the need for further advancements in this area.

*The work does not relate to the author's position at Amazon.
†Corresponding authors.

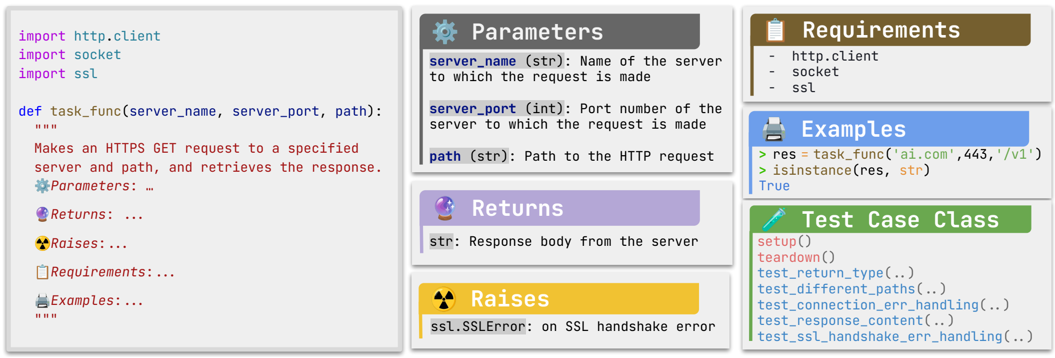


Figure 1: Programming tasks in `BigCodeBench` are structured with complex instructions in the docstrings, annotated by experts. The behavior of the solution is evaluated against a class of rigorous test cases with the proper environment setup.

1 Introduction

Task automation, including competitive programming (Li et al., 2022; Hendrycks et al., 2021; Jain et al., 2024), GitHub issue resolution (Yang et al.), and question answering (Gao et al., 2023; Chen et al.), has attracted significant interest from academia and industry to facilitate the development of advanced models for code, especially in Python (Wang et al.). With recent advances in data-centric deep learning techniques, large language models (LLMs) trained on large-scale corpora have shown superior capabilities of translating textual inputs to syntactically correct and functional programs. However, widely-used benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) only contain short, self-contained, and algorithm-focused programming tasks and have been saturated by recent model releases. In this work, we aim to close the evaluation gap between these isolated coding exercises and real-world programming, asking **Can LLMs solve more challenging and more practical tasks via programs?**

Solving programming tasks in real-world scenarios typically has two main characteristics: (1) **Diverse Function Calls as Tools**[1]. A complex programming task often requires the invocation of diverse function call sequences as tools (Robillard & DeLine, 2011; Hu et al., 2018; Qin et al., 2023). To avoid reinventing the wheel, domain-specific libraries are designed to cover function calls (or APIs) with comprehensive functionalities; and (2) **Complex Instructions**. Due to the complexity to perform various programming tasks, instructions may require compositional reasoning ability to perform a sequence of functionalities in the correct order (e.g., input data manipulation, error message handling, and specific output formatting) (Wiegers & Beatty, 2013; Paetsch et al., 2003; Partsch, 2012). For instance, creating a network application that includes functionality for retrieving responses from an HTTPS server (Figure 1) requires integrating several components with multiple function calls, such as managing SSL contexts, handling socket connections, and ensuring the response is returned in the specific format.

Building a high-quality execution-based benchmark and the environment that simulates aforementioned tasks with practical and challenging constraints is non-trivial. First, it is hard to naturally source self-contained programming tasks with complex instructions, unlike the short code exercises in HumanEval. Although most GitHub repositories contain realistic source code, the functions inside these repositories often require cross-file information (Ding et al., 2024). Second, real-world programming scenarios are extremely diverse (Zhout et al., 2023). Existing benchmarks (Zan et al., 2022b; Lai et al., 2023) only focus on popular scenarios like data science. Third, mainstream programming benchmarks have a significant number of ambiguous or underspecified specifications, resulting in inaccurate evaluation results (Siddiq et al., 2024; Jain et al.). While there have been attempts to improve the data quality with LLMs (Jain et al., 2023), LLMs have their own bias and cannot reliably perform refinement (Zheng et al., 2024a).

[1]In this work, we refer to "tools" as library-oriented but non-object-oriented Code Function Call APIs, which are discussed in `Gorilla OpenFunctions`. The Code Function Calling APIs are typically seen in common external Python packages like Numpy, Sklearn, and Pandas.

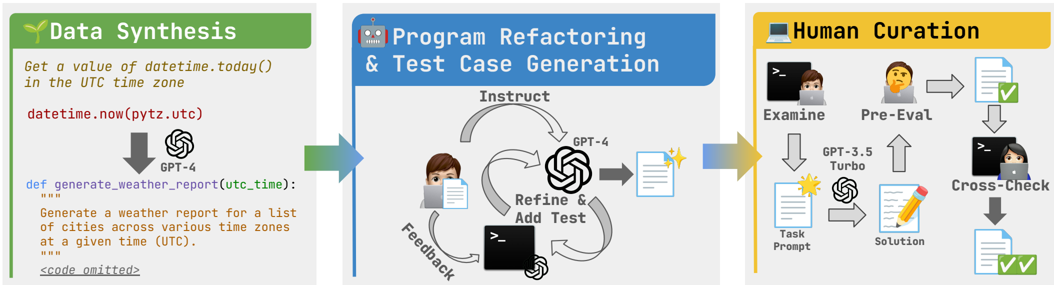


Figure 2: Each programming task in `BigCodeBench` is created through a three-stage construction process. The task quality is controlled by the human-LLM collaboration.

To construct massive, high-quality programming tasks, we propose a novel framework (Figure 2) that uses collaboration between LLMs and human experts to build a rigorous execution-based benchmark, `BigCodeBench`. Particularly, we utilize LLMs to source programming tasks, refactor programs, and add test cases, under constant human supervision. Our benchmark contains 1,140 rich-context and multi-tool-use programming tasks in Python, covering 723 function calls from 139 popular libraries across 7 domains. As we aim to let models reason any suitable function calls to complete the tasks via code, we design the unit test cases in an open-ended manner and examine certain behaviors based on different inputs (Jain et al.). We assess two common programming scenarios: (1) Code Completion (`BigCodeBench-Complete`), which evaluates the capability of code generation based on the structured docstrings; and (2) Instruction to Code (`BigCodeBench-Instruct`), which evaluates the ability to complete programming tasks based on natural-language-oriented (NL-oriented) instructions. While `BigCodeBench-Complete` emphasizes structured docstring prompts, `BigCodeBench-Instruct` challenges LLMs to generate precise code without relying on non-essential details like interactive Python examples.

Through extensive studies on 60 models, we find that LLMs still struggle to invoke multiple function calls from cross-domain libraries and follow complex instructions to solve programming tasks using Python. Specifically, the best performing LLM, GPT-4o, solves merely 60% of tasks on `BigCodeBench-Complete` and less than 50% on `BigCodeBench-Instruct`, indicating that LLMs themselves lack the ability to align with human expectations when instructions are more natural. Interestingly, we find that some instruction-tuned LLMs like GPT-4 constantly refuse to follow long instructions to repeat essential context and thus fail the test cases. Furthermore, LLMs perform differently when using domain-specific function calls as tools. We also demonstrate the strongly positive correlations between mainstream benchmarks and `BigCodeBench`, validating our evaluation results.

2 Benchmark Construction

In this section, we introduce our human-LLM-collaboration framework for the `BigCodeBench` construction. As shown in Figure 2, the process of `BigCodeBench-Complete` encompasses three stages: Data Synthesis, Semi-automatic Program Refactoring and Testing Case Generation, and Human Curation. In addition, we construct `BigCodeBench-Instruct` in Section 2.4, an NL-oriented benchmark for programming-task solving. Further information is available in Appendix B and Appendix J. The construction is progressively contributed by 20 authors as annotators for one year in total, with 75% of them having more than 5 years of experience in Python programming. The progress is continually managed by the lead annotator to control the data quality. Each constructed programming task is expected to: (1) have clear instructions inside PEP-257-structured docstrings; (2) use multiple libraries as tools to solve the problem; and (3) be paired with at least five test cases encapsulated in the `unittest` framework, with a complex test setup (e.g., database connection, and directory creation), to verify program behaviors instead of relying on simple input-output assertions.

2.1 Data Synthesis

One intuitive method would be to rely on source code repositories to construct the function-level programming tasks. However, most repository-level functions require cross-file information, such as customized modules (Ding et al., 2024), which are hard to self-contain and document. We argue that leveraging LLMs to synthesize customized programming tasks can be more viable (Wei et al., 2023; Luo et al., 2023), especially when overseen by humans. Given a code snippet of API usage with a brief human instruction as the seed example (Figure 2), an LLM is instructed to enrich the programming intent and refine the corresponding implementation by using diverse libraries. Specifically, we use seed examples from ODEX (Wang et al., 2023c), a benchmark containing intent-paired code skeletons from Stack Overflow. We use the GPT-4 API[2], the strongest LLM at the time of data sourcing, to synthesize the programming tasks. To help the LLM synthesize self-contained and relevant programming tasks based on the seed example, we instruct the model with a 2-shot in-context demonstration (Appendix J.1) crafted by the lead annotator.

As previous studies have shown that LLMs favor their own generations (Zheng et al., 2024a; Panickssery et al., 2024), such phenomena can make the model evaluation unfair. We mitigate the model biases with an obfuscation and perturbation process. We first replace the semantic-rich program entry points with dummy function names. In addition, we perturb the natural language descriptions in docstrings with the back-translation method of NL-Augmenter (Dhole et al., 2023), inspired by ReCode (Wang et al., 2023a). After validating the post-processed programs with an abstract syntax tree parser, we collected 4,718 function-level programming samples.

2.2 Semi-automatic Program Refactoring and Testing Case Generation

Programs synthesized by LLMs may contain various issues, such as undeclared variables and runtime bugs. Without proper verification, the implementation cannot directly serve as a ground-truth solution. To construct a high-quality execution-based benchmark, we need to add test cases that can rigorously verify the correctness of programs and identify any bugs. However, it takes non-trivial effort for human developers to understand synthesized programs and properly refactor them with thorough testing. To improve the program quality and ease human annotation, we propose a conversion-driven framework for program refactoring and test case generations inspired by (Xia & Zhang, 2023). Specifically, we utilize the Code Interpreter session in the web-based GPT-4, which is back-ended by a Linux-based virtual environment with pre-installed Python packages. We engage 13 authors as human annotators (including the lead annotator) for this step, assigning each annotator 100 randomly sampled programming tasks based on their preferences of code topics (Appendix J.2.1). We detail the design of this human-in-the-loop framework from human and LLM aspects as follows:

Human Aspect Human developers possess varying preferences and levels of familiarity with specific data types and programming scenarios. To aid human annotators in providing more precise feedback for refining programs with GPT-4, we have defined 10 data types (e.g., SQL, CSV, and Python built-in types) and task scenarios (e.g., data analysis, networking, and visualization). GPT-4 API is utilized to automatically classify each program according to these categories, with detailed descriptions available in Appendix J.2.1. The annotators' role is to continually instruct GPT-4 to refactor the programs and to provide continuous feedback to guide the model whenever it fails to self-debug or incorrectly refactor the program.

LLM Aspect To effectively guide GPT-4 in the iterative refinement of programs and test cases, we provide detailed annotation guidelines in Appendix J.2.2 as an initial prompt. These guidelines encompass two high-level instructions: (1) Refine the function, including its docstrings, to enhance realism and reduce ambiguity; and (2) Write unit tests to ensure the functional correctness of the given program description. Specifically, the model is taught to follow a step-by-step refinement process: (1) Remove unused libraries and add necessary libraries if they are missing in the code snippet; (2) Reformat docstrings to adhere to PEP-257 conventions; (3) Align program implementations with the instructions inside docstrings; (4) Write and execute unit tests to ensure they pass; and (5) Add refined programs and test cases to files for downloading.

During interactions with GPT-4, we identify two main drawbacks in the Code Interpreter session. First, GPT-4 struggles to write proper test cases when mocking tests are employed. While the model can generate high-level designs of mocking tests, it often fails to understand how test cases should be constructed based on execution feedback. Second, GPT-4 can become stuck while resolving runtime

[2]We use the model version `gpt-4-0613`.

bugs, leading to iterative refinement until the session times out. Continuous human feedback on viable solutions is essential to address these issues and ensure the model stays on track.

During the post-processing of the annotated data, we observe that a significant number of test cases were incomplete. This issue arises because GPT-4 often omits partial content when writing long contexts into files. After removing all invalid programs via program analysis, we end up with 1,223 refactored programming tasks with paired test cases.

2.3 Human Curation

To enhance the benchmark quality, we implement a three-fold human curation process:

Examination We first perform a rigorous manual examination of the benchmarks, guided by a set of detailed guidelines (Appendix J.3) to add more test cases and resolve a list of runtime issues due to the LLM-generated flaky tests (Luo et al., 2014). In addition, we aim to formalize further the programming tasks based on the following criteria: (1) The task should utilize at least two libraries to enrich the task scopes; (2) Import statements should be limited to those used in the task implementations and be natural based on human understanding; (3) The docstring should follow PEP-257 to ensure consistent styles; (4) Both the task implementation and test cases should be clearly aligned with the task docstrings, based on human understanding; and (5) Required library modules in the docstrings should be aligned with the import statements. For test case construction, the criteria are as follows: (1) Test cases should be encapsulated by the `unittest` framework, making them `pytest`-compatible; and (2) Test cases need to be deterministic, asserting expected program behaviors based on the inputs (Jain et al.).

Pre-Evaluation To enhance the benchmark quality, we perform a dry run to pre-evaluate an LLM other than GPT-4 used in previous steps. We choose GPT-3.5-Turbo API[3] to generate solutions based on the examined task prompts. By understanding how the model fails on the tasks, annotators may add essential details to the docstrings to clarify the task instructions but avoid describing step-by-step implementations.

Cross-Checking To further validate data quality and ensure consistency across all programming tasks in the testing environment, 7 additional annotators refine and finalize the data annotated by others. This round focuses on the utility of docstrings and test cases. We automatically parse the docstrings and ask annotators to manually correct the docstring structures, specifically addressing task descriptions, function parameters, expected returns, exception handling, required modules, and interactive examples. Additionally, we remove unused imported modules via program analysis. For the interactive examples, we ensure their correctness via `doctest`, except for those requiring system and network access. The confirmed programming tasks are finally validated by the automated test workflows in GitHub Container Registry, where the test cases are automatically run against the task implementations in a configured sandbox environment. To ensure the finalized data quality, we randomly assign 33 finalized task prompts to the 11 annotators (the lead annotator was excluded; one annotator was unavailable at the time), to write the solutions. The lead annotator conducts the evaluation of the solutions, finding that 97% (32 out of 33) of sampled tasks can pass all test cases.

2.4 Benchmarking NL-oriented Instructions to Code Generation

When instruction tuning LLMs using NL data, the input is mainly an NL instruction, and the target output is the NL or source code for completion (Muennighoff et al., 2023). This training objective aligns with the downstream applications such as multi-turn dialogue, where users ask questions or provide task-specific instructions to the models. While existing programming benchmarks commonly format the verbose instructions in docstrings (Chen et al., 2021), users may instruct the models to generate code samples with the less verbose NL context. Despite that there have been similar attempts like HumanEvalPack (Muennighoff et al., 2023) addressing this limitation, their inputs still lack some naturalness. Generally, users tend to describe the high-level idea of functionalities and avoid redundant information (e.g., parameters) or too-specific details (e.g., interactive examples). Thus, we create `BigCodeBench-Instruct`, a benchmark variant that prompts LLMs to solve

[3] We use model version `gpt-3.5-turbo-1106`.

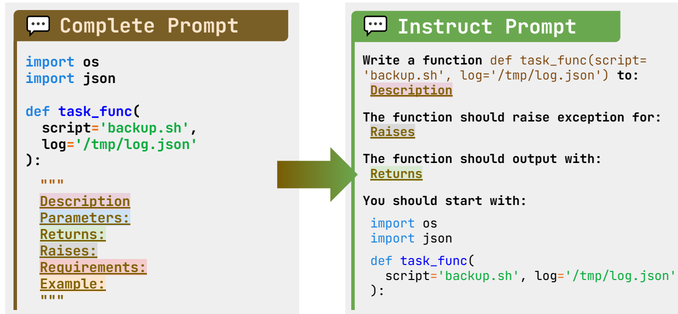


Figure 3: Each task prompt in `BigCodeBench-Instruct` are transformed from `BigCodeBench-Complete` via the pre-defined rules. We omit non-essential details during the transformation.

programming tasks with more NL-oriented instructions, assessing the model's ability to understand human requirements correctly. Based on the task prompts created in `BigCodeBench-Complete`, we design a set of parsing rules and transform them into more natural instructions (Figure 3). For quality control, 5 authors who do not participate in previous stages inspect the randomly sampled prompts and their corresponding ground-truth solutions and reach an agreement on the alignment between the instruction prompts and task implementations.

3 Benchmark Statistics

Domain	Library	Function Call
Computation (63%)	pandas, numpy, sklearn, scipy, math, nltk, statistics, cv2, statsmodels, tensorflow, sympy, textblob, skimage…	pandas.DataFrame, numpy.random, numpy.random.seed, numpy.array, numpy.mean, pandas.read_csv, numpy.random.randint, pandas.Series…
General (44%)	random, re, collections, itertools, string, operator, heapq, ast, functools, regex, bisect, inspect, unicodedata…	collections.Counter, random.seed, random.randint, random.choice, re.sub, re.findall, itertools.chain…
Visualization (31%)	matplotlib, seaborn, PIL, folium, wordcloud, turtle, mpl_toolkits	matplotlib.pyplot, matplotlib.pyplot.subplots, matplotlib.pyplot.figure…
System (30%)	os, json, csv, shutil, glob, subprocess, pathlib, sqlite3, io, zipfile, sys, logging, pickle, struct, psutil…	os.path, os.path.join, os.path.exists, os.makedirs, glob.glob, os.listdir, json.load, csv.writer, shutil.move…
Time (10%)	datetime, time, pytz, dateutil, holidays, calendar	datetime.datetime, datetime.datetime.now, time.time, time.sleep, datetime.datetime.strptime…
Network (8%)	requests, urllib, bs4, socket, django, flask, ipaddress, smtplib, http, flask_mail, cgi, ssl, email, mechanize…	arse.urlparse, django.http.HttpResponse, ipaddress.IPv4Network, smtplib.SMTP, requests.post, socket.gaierror…
Cryptography (5%)	hashlib, base64, binascii, codecs, rsa, cryptography, hmac, blake3, secrets, Crypto	cryptography.fernet.Fernet.generate_key, cryptography.hazmat.primitives.padding, cryptography.hazmat.primitives.padding.PKCS7…

Figure 4: Examples of tools in `BigCodeBench` are illustrated. Each function call belongs to a domain-specific library. The distribution of each domain is computed based on the frequency of domain-specific libraries appearing per task. For example, "63%" in "Computation" means that there are 63% tasks in `BigCodeBench` using at least one computation library.

Overall Statistics The first part of Table 1 presents an overview comparison between `BigCodeBench` and other mainstream function-level Python programming benchmarks. We note that the full DS-1000 dataset emphasizes the perturbed tasks to avoid model memorization and thus includes an original subset for reference. Therefore, we also include statistics for non-perturbed problems (DS-1000 Orig.). As the provided statistics suggest, `BigCodeBench` contains much more rigorous execution-based evaluation and has longer task prompts that contain complex instructions. The ground-truth solutions are also longer than in prior benchmarks, indicating that the tasks inside `BigCodeBench` require more complex implementation logic. To illustrate the programming complexity, we measure cyclomatic complexity, which is the number of independent paths through

Table 1: Summarized statistics of representative function-level Python programming benchmarks , partially inspired by (Zan et al., 2023). `BigCodeBench` are more complex regarding the depth of task complexity and breadth of tool-use knowledge. **Cov.**: Branch Coverage. **Char.**: Characters. **C.C.**: Cyclomatic Complexity (starting from 0, higher values indicate more complex code). **Lib.**: Library. **Dom.**: Domain. **Std.**: Python Standard Library. **Ext.**: Python External Library.

Overall Statistics

Benchmark	**# Task**	**Test** (Avg.)		**Prompt** (Avg.)		**Solution** (Avg.)		
		#	Cov.	Char. (Code)	Line	Char.	Line	C.C.
HumanEval	164	7.8	98%	450.6 (450.6)	13.7	180.9	6.8	3.6
DS-1000	1,000	1.6	98%	871.8 (193.9)	29.1	138.1	5.1	1.6
DS-1000 (Orig.)	452	1.5	98%	831.4 (201.2)	26.2	115.5	4.2	1.4
ODEX	945	1.8	96%	87.5 (87.5)	1.0	50.4	1.9	1.4
`BigCodeBench-Complete`	1,140	5.6	99%	1112.5 (1112.5)	33.5	426.0	10.0	3.1
`BigCodeBench-Instruct`				663.2 (124.0)	11.7			

Tool Statistics

Benchmark	**# Dom.**	**# Lib.**	**# Call**	**Tasks** (Avg.)		**Combination**		
		Std. / Ext.	Std. / Ext.	# Lib.	# Call	# Lib.	# Calls	# Dom.
HumanEval (Chen et al., 2021)	3	4 / 0	7 / 0	0.1	0.1	6	8	5
DS-1000 (Lai et al., 2023)	4	5 / 9	7 / 321	0.8	1.1	66	331	24
DS-1000 (Orig.) (Lai et al., 2023)	4	4 / 9	5 / 289	0.9	1.3	59	260	23
ODEX (Wang et al., 2023c)	7	40 / 26	128 / 102	0.6	0.5	105	202	20
`BigCodeBench`	7	77 / 62	281 / 442	2.8	4.7	577	1045	56

the task solution. We notice that `BigCodeBench` has a similar complexity to HumanEval, much higher than DS-1000 and ODEX. The high cyclomatic complexity indicates that solving the tasks in `BigCodeBench` requires non-trivial reasoning ability from the programming perspective.

Tool Statistics In this work, we highlight any libraries and function calls from these libraries as program-based tools for LLMs (Qin et al., 2023). Therefore, we compare the tools used among the mainstream benchmarks in the second part of Table 1. First, `BigCodeBench` covers 281 function calls from 77 standard libraries and 442 from 62 external libraries, which is far more diverse than the other benchmarks. The examples of these tools are shown in Figure 4. Based on the PyPI download statistics, the mean download counts of the recent 30 days are 56.9M+ for the 62 external libraries, suggesting their high popularity in real-world software development. Second, `BigCodeBench` frequently invokes a sequence of function calls from multiple libraries to solve a single task, requiring significant compositional reasoning ability for task-solving. Third, `BigCodeBench` has more diverse combinations among libraries, function calls, and domains in the ground-truth solutions. We visualize the library and function call density used by different benchmarks in Appendix I, showing the even broader tool-use diversity in `BigCodeBench`. To better understand solution complexity (measured by characters) and tool-use diversity (measured by distinct function calls), we compare the tasks in `BigCodeBench` with those in representative benchmarks in Figure 5 (details are provided in Appendix I.2). We find that `BigCodeBench` requires more complex reasoning and problem-solving skills to implement comprehensive functionalities.

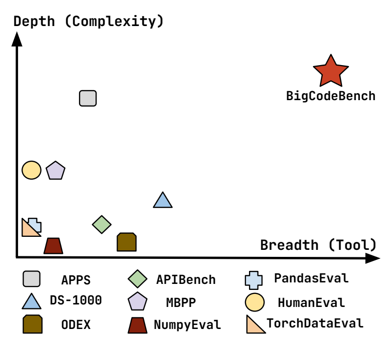


Figure 5: Complexity - tool comparisons with various benchmarks.

4 Evaluation

Our evaluation uses the unbiased version of Pass@K (Chen et al., 2021) to accurately assess the functional correctness of generated code snippets by LLMs. To make general observations, we extensively evaluate 60 state-of-the-art LLMs on `BigCodeBench-Complete` and 35 instruction-tuned LLMs on `BigCodeBench-Instruct`. Specifically, following prior works (Roziere et al., 2023; Liu et al., 2024; Lai et al., 2023), we report Pass@1 with greedy decoding for the main experiments in the zero-shot setting. To investigate more thoroughly, we compute Pass@1 and

Pass@5 results with random sampling to generate N (N=5) samples with a temperature of 0.8 and top-p of 0.95 in Appendix L. While it is encouraged to generate much more ($N \geq K$) samples to avoid bias, we take the lower bound due to limited computational resources. We use the same prompts for code generation from (Liu et al., 2024), given in Appendix K.

4.1 MEASURING TASK-LEVEL PERFORMANCE

We first evaluate the task-solving performance of each LLM and summarize the findings as follows. We show the main results in Figure 6. As models constantly omit the essential code in the generation and hence fail the tests, we calibrate the generation quality by adding the missing setup and calculate Pass@1, which is denoted as **calibrated Pass@1**. The Pearson's r correlation between the model ranks on `BigCodeBench-Complete` and `BigCodeBench-Instruct` is 0.982, indicating a strong alignment. In addition, model rankings show the signs of scaling laws (Kaplan et al., 2020), where bigger models can solve more tasks. We also observe that there is still some performance gap between the top closed models and open ones. Detailed studies can be found in Appendix M. We highlight a few findings as follows:

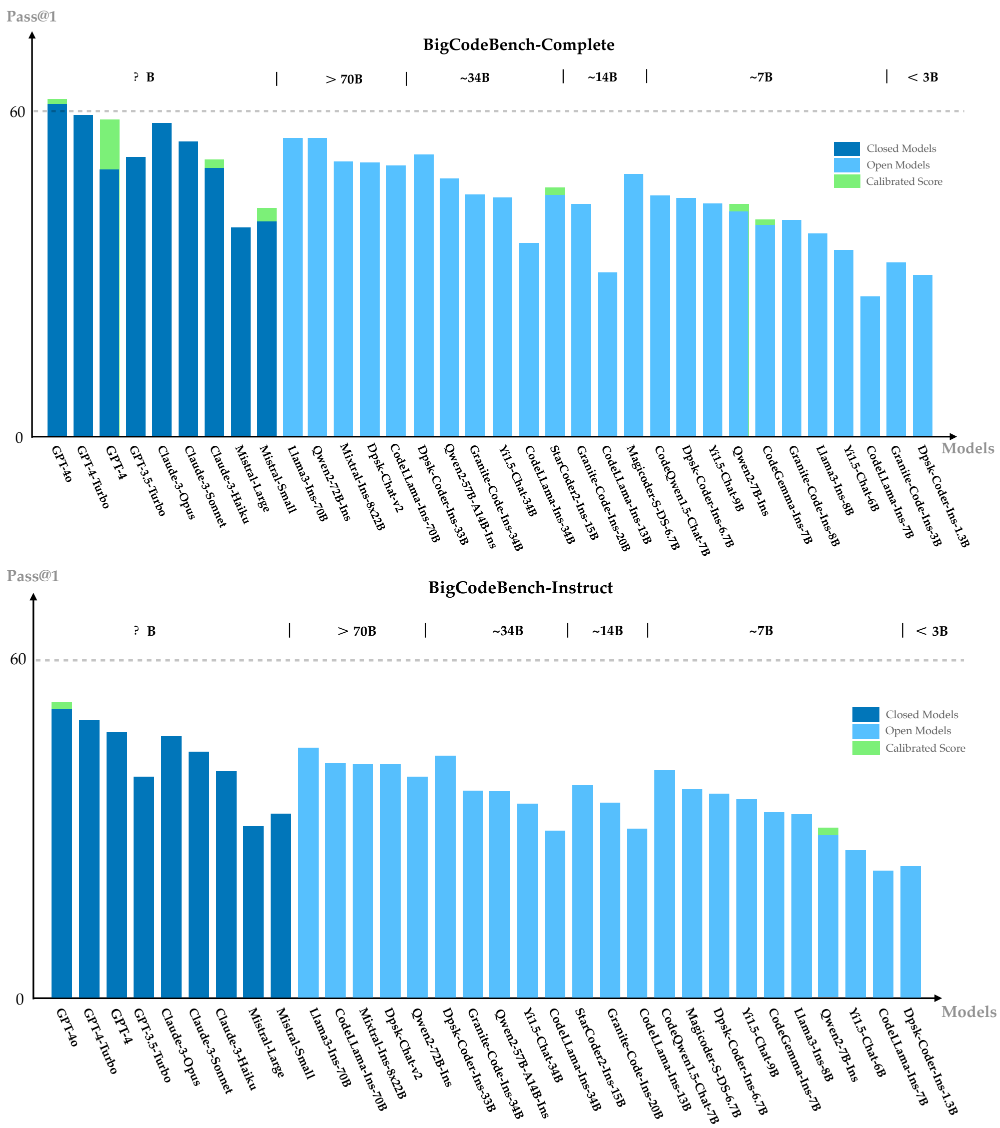


Figure 6: Pass@1 results of instruction-tuned LLMs on `BigCodeBench-Complete` (Top) and `BigCodeBench-Instruct` (Bottom). We only highlight the calibrated results having at least a difference of 1% from the original Pass@1.

Instruction-tuned LLMs omit essential details of long code prompts Interestingly, we observe that instruction-tuned LLMs can omit the essential import statements of the given prompts in `BigCodeBench-Complete`, which can lead to task failure due to the lack of proper module and constant definitions. The omission is likely to happen when models need to repeat the long context in the response. Such behaviors are denoted as "model laziness" in long-context interactions, similar to the observations in Section 2.2. Due to the limited prompt length of existing programming benchmarks (Table 1), there is no quantitative evidence of the laziness phenomenon in prior code generation benchmarks. To understand how laziness can affect the model performance, we calibrate the generation quality by adding the missing setup (e.g., import statements and global constants). When comparing the calibrated Pass@1 and the original ones in the top figure of Figure 6 and Appendix L, we find that GPT-4 tends to omit much more context and perform poorly on `BigCodeBench-Complete`, consistent with the previous community discussion[4] and confirmed by OpenAI (OpenAI, 2024b). While instruction-tuned LLMs have an average performance degradation of 0.8% on `BigCodeBench-Complete`, there is a less than 0.3% difference on `BigCodeBench-Instruct`, which validates the hypothesis that models omit more information of longer inputs.

Instruction tuning helps to follow programming constraints When comparing instruction-tuned LLMs with their base ones (Appendix L), we observe that instruction tuning can improve the capability of following complex constraints of the prompts. The mean calibrated Pass@1 on `BigCodeBench-Complete` for instruction-tuned LLMs and paired base LLMs are 40.7% and 35.7%, respectively, indicating a great performance gap. The performance disparity is further magnified when the instruction-tuned LLMs are calibrated. We also note that the task performance of CodeQwen1.5-7B is not enhanced by instruction tuning, possibly due to the lack of fine-grained data during training.

LLMs are sensitive to the verbosity of programming instructions From the bottom figure of Figure 6, we notice that LLMs perform much worse on `BigCodeBench-Instruct` than `BigCodeBench-Complete` with an average decrease of 8.5% on Pass@1, while maintaining similar rankings. This observation indicates that LLMs still lack the proper understanding of condensed human requirements since the task instructions of `BigCodeBench-Instruct` are less verbose. While it is possible that the lower verbosity may introduce more ambiguity, we make sure that the instructions transformed from `BigCodeBench-Complete` do not lose the key information from the human perspective. In addition, we find that models with lower scores on `BigCodeBench-Complete` degrade less on `BigCodeBench-Instruct`, limited by their programming capability.

4.2 Measuring Tool-Level Performance

Besides evaluating task completion performance, we try to understand how well LLMs can use tools for task-solving. As it is hard to evaluate the accuracy of each independent function call regardless of the correctness of the model solutions, we rely on the calibrated greedy decoding results on `BigCodeBench-Complete` and deem the use of the associated library correct if the task is passed by all test cases.

Jack of all trades, master of most. Figure 7 shows the top 5 calibrated instruction-tuned LLMs ranked on `BigCodeBench-Complete`. The best overall-performing LLMs, like GPT-4o, excel in most domains but still fall short in certain ones. We suggest that domain specialization can result from the training data. Similar findings can be found in the base models, which are visualized in Appendix L.2.

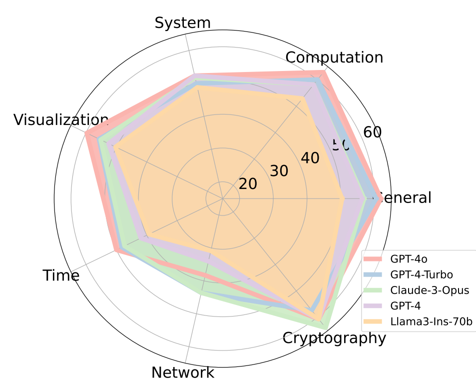


Figure 7: Top 5 instruction-tuned LLMs on `BigCodeBench-Complete`.

[4] https://community.openai.com/t/why-i-think-gpt-is-now-lazy

Table 2: Tool-use comparisons between all generated solutions (**Sol.**) and ground truths (**GT.**) on `BigCodeBench-Complete`.

	Library (%)		**Function Call** (%)	
	$Sol. \subseteq GT.$	$Sol. \nsubseteq GT.$	$Sol. \subseteq GT.$	$Sol. \nsubseteq GT.$
Pass	79.84	20.16	40.46	59.54
Fail	71.78	28.22	22.47	77.53

LLMs use different function calls to solve tasks.
We assess how all 60 calibrated models use tools on `BigCodeBench-Complete` and report the mean statistics in Table 2. By inspecting the library usage, we find that models use imported libraries in more than 70% of tasks. For the remaining 20%, we notice that models tend to import additional libraries to solve the tasks. After some inspection, we notice that most of these added libraries are standard, explaining why they can still pass the tests. When analyzing the function calls, we find that models tend to use different function calls from the ones in ground truths. Due to the open-endedness of the programming tasks, the invocation of function calls is expected to be flexible. However, using more diverse function calls is more likely to introduce task failures. We provide several examples where the model-generated solution contains function calls different from the ground truth in Appendix M, Examples 9 and 10.

5 Related Work

Large Language Models for Code With the rise of LLMs, there have been various models trained on code. Codex (Chen et al., 2021) marks the first base LLM pre-trained on code, which was used as the backbone model for GitHub Copilot. More recently, pre-trained base code models (Nijkamp et al., 2022; Li et al., 2023; Lozhkov et al., 2024; Roziere et al., 2023) have been built to perform accurate code completion. Later, with the advance in instruction tuning (Ouyang et al., 2022), LLMs can generate the code snippet that aligns with the given NL instruction. Instruction-tuned code models (Muennighoff et al., 2023; Luo et al., 2023; Wei et al., 2023), are generally better at programming than their base ones.

Programming Benchmarks To assess the programming capability of LLMs, researchers have proposed many programming benchmarks. Most existing benchmarks focus on short, self-contained, and algorithm-specific programming tasks, such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). Recently, open-domain coding benchmarks (Wang et al., 2023c; Lai et al., 2023) have been built to challenge code LLMs on application-specific scenarios by using specific sets of tools and libraries. However, they focus on simple intents and use limited specific function calls per programming task, making their evaluation less challenging and realistic. Benchmarks like SWE-bench (Jimenez et al., 2023) are built to evaluate the performance of a code agent framework (e.g., iterated prompting, real-time environment interaction, and long-context exploration). Our `BigCodeBench` focuses on evaluating the fundamental code generation capability of LLMs, which is also an essential path toward strong code agents. In addition, SWE-Bench is constructed from GitHub repositories with existing test cases — which limits the diversity of the tasks considered. In contrast, our collaborative LLM-human annotator procedure allows generating tasks from seed queries collected from (Wang et al., 2023c), tackling a broader range of software tasks.

6 Conclusion

We introduce `BigCodeBench`, a new high-quality programming benchmark constructed via the collaboration between human experts and LLMs, assessing the capability of tool use and complex instruction following. Through the extensive evaluation of 60 LLMs, we find that there is a long way for models to achieve perfection on this benchmark and share a few findings that can potentially improve the performance. We urge the community to work on more advanced LLMs for code and continually build upon our benchmark and extended `BigCodeBench-Hard` (Appendix F),as discussed in our long-term roadmap (Appendix G).

ACKNOWLEDGEMENTS

We truly appreciate BigCode making many great works happen, including The Stack (Lozhkov et al., 2024), SantaCoder (Allal et al., 2023), StarCoder (Li et al., 2023), OctoPack (Muennighoff et al., 2023), Astraios (Zhuo et al., 2024), StarCoder2 (Lozhkov et al., 2024), and StarCoder2-Instruct (BigCode, 2024). `BigCodeBench` cannot be built without the support of the BigCode community. We thank Sea AI Lab and MASSIVE for providing the computational resources. This research / project is supported by Terry Yue Zhuo's CSIRO's Data61 PhD Scholarships, the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002), and Xiaoning Du's Google Research Scholar Program Award. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. For the benchmark, we particularly thank Xiangru Tang, Dmitry Abulkhanov, Noah Ziems, Chengran Yang, Jiamou Sun, Nickil Maveli, and Lili Bo for their eagerness to participate. We are extremely grateful to the EvalPlus team for their open-source evaluation framework. We also thank Loubna Ben Allal, Zhiruo Wang, Zhensu Sun, Sean Hughes, and Zhou Yang for their valuable comments.

REFERENCES

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

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 2655–2668, 2021.

Mistral AI. Mistral. https://mistral.ai/news/mistral-large/, 2024a.

Mistral AI. Mixtral-8x22b-v0.1. https://huggingface.co/mistralai/Mixtral-8x22B-v0.1, 2024b.

Mistral AI. Mixtral-8x22b-instruct-v0.1. https://huggingface.co/mistralai/Mixtral-8x22B-Instruct-v0.1, 2024c.

AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! In Deep Learning for Code (DL4C) Workshop, 2023.

Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. Journal of systems and software, 86(8):1978–2001, 2013.

AI Anthropic. The claude 3 model family: Opus, sonnet, haiku. Claude-3 Model Card, 2024.

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

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

Emily M Bender and Batya Friedman. Data statements for natural language processing: Toward mitigating system bias and enabling better science. Transactions of the Association for Computational Linguistics, 6:587–604, 2018.

BigCode. starcoder2-15b-instruct-v0.1. https://huggingface.co/bigcode/starcoder2-15b-instruct-v0.1, 2024.

Tom B Brown. Language models are few-shot learners. arXiv preprint arXiv:2005.14165, 2020.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. IEEE Transactions on Software Engineering, 49(7):3675–3691, 2023.

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. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. Transactions on Machine Learning Research.

Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. Teaching large language models to self-debug. In The 61st Annual Meeting Of The Association For Computational Linguistics, 2023.

Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. Pymt5: multi-mode translation of natural language and python code with transformers. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 9052–9065, 2020.

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

Kaustubh Dhole, Varun Gangal, Sebastian Gehrmann, Aadesh Gupta, Zhenhao Li, Saad Mahamood, Abinaya Mahadiran, Simon Mille, Ashish Shrivastava, Samson Tan, et al. Nl-augmenter: A framework for task-sensitive natural language augmentation. Northern European Journal of Language Technology, 9(1), 2023.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. Advances in Neural Information Processing Systems, 36, 2024.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. arXiv preprint arXiv:2308.01861, 2023.

Yanai Elazar, Akshita Bhagia, Ian Magnusson, Abhilasha Ravichander, Dustin Schwenk, Alane Suhr, Pete Walsh, Dirk Groeneveld, Luca Soldaini, Sameer Singh, et al. What's in my big data? arXiv preprint arXiv:2310.20707, 2023.

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

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In International Conference on Machine Learning, pp. 10764–10799. PMLR, 2023.

Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé Iii, and Kate Crawford. Datasheets for datasets. Communications of the ACM, 64(12): 86–92, 2021.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. In International Conference on Learning Representations.

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

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. In Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2), 2021.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. In Proceedings of the 27th International Joint Conference on Artificial Intelligence, pp. 2269–2275, 2018.

Edward Hughes, Michael Dennis, Jack Parker-Holder, Feryal Behbahani, Aditi Mavalankar, Yuge Shi, Tom Schaul, and Tim Rocktaschel. Open-endedness is essential for artificial superhuman intelligence. arXiv preprint arXiv:2406.04268, 2024.

Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent environment. In ICML 2024.

Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E Gonzalez, Koushik Sen, and Ion Stoica. Llm-assisted code cleaning for training accurate code generators. arXiv preprint arXiv:2311.14904, 2023.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. arXiv preprint arXiv:2403.07974, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In The Twelfth International Conference on Learning Representations, 2023.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. arXiv preprint arXiv:2001.08361, 2020.

Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of naacL-HLT, volume 1, pp. 2. Minneapolis, Minnesota, 2019.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. Advances in neural information processing systems, 35: 22199–22213, 2022.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Proceedings of the 29th Symposium on Operating Systems Principles, pp. 611–626, 2023.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In International Conference on Machine Learning, pp. 18319–18345. PMLR, 2023.

Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A systematic review of api evolution literature. ACM Computing Surveys (CSUR), 54(8):1–36, 2021.

Raymond Li, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, LI Jia, Jenny Chim, Qian Liu, et al. Starcoder: may the source be with you! Transactions on Machine Learning Research, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. Science, 378(6624):1092–1097, 2022.

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

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In The Twelfth International Conference on Learning Representations.

Renze Lou, Kai Zhang, and Wenpeng Yin. A comprehensive survey on instruction following. arXiv preprint arXiv:2303.10475, 2023.

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

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 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).

Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp. 643–653, 2014.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In The Twelfth International Conference on Learning Representations.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In The Twelfth International Conference on Learning Representations, 2023.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, et al. Granite code models: A family of open foundation models for code intelligence. arXiv preprint arXiv:2405.04324, 2024.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. In The Twelfth International Conference on Learning Representations, 2023.

Jinjie Ni, Fuzhao Xue, Xiang Yue, Yuntian Deng, Mahir Shah, Kabir Jain, Graham Neubig, and Yang You. Mixeval: Deriving wisdom of the crowd from llm benchmark mixtures. arXiv preprint arXiv:2406.06565, 2024.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In The Eleventh International Conference on Learning Representations, 2022.

OpenAI. Gpt-3.5-turbo. https://openai.com/index/introducing-chatgpt-and-whisper-apis/, 2023.

OpenAI. Gpt-4o. https://openai.com/index/hello-gpt-4o/, 2024a.

OpenAI. Gpt-4-turbo. https://openai.com/index/new-embedding-models-and-api-updates/, 2024b.

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

Frauke Paetsch, Armin Eberlein, and Frank Maurer. Requirements engineering and agile software development. In WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003., pp. 308–313. IEEE, 2003.

Arjun Panickssery, Samuel R Bowman, and Shi Feng. Llm evaluators recognize and favor their own generations. arXiv preprint arXiv:2404.13076, 2024.

Helmut A Partsch. Specification and transformation of programs: a formal approach to software development. Springer Science & Business Media, 2012.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. arXiv preprint arXiv:2305.15334, 2023.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, et al. Tool learning with foundation models. arXiv preprint arXiv:2304.08354, 2023.

Alec Radford. Improving language understanding by generative pre-training. 2018.

N Reimers. Sentence-bert: Sentence embeddings using siamese bert-networks. arXiv preprint arXiv:1908.10084, 2019.

Martin P Robillard and Robert DeLine. A field study of api learning obstacles. Empirical Software Engineering, 16:703–732, 2011.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, YK Li, Yu Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. arXiv preprint arXiv:2402.03300, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems, 36, 2024.

Mohammed Latif Siddiq, Simantika Dristi, Joy Saha, and Joanna Santos. Quality assessment of prompts used in code generation. arXiv preprint arXiv:2404.10155, 2024.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. arXiv preprint arXiv:2403.08295, 2024.

Minyang Tian, Luyu Gao, Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, HAO TONG, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu A Huerta, and Hao Peng. Scicode: A research coding benchmark curated by scientists. In The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track, 2024. URL https://openreview.net/forum?id=ADLaALtdoG.

Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai. Exploring how deprecated python library apis are (not) handled. In Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp. 233–244, 2020.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering, 2024a.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. Recode: Robustness evaluation of code generation models. In The 61st Annual Meeting Of The Association For Computational Linguistics, 2023a.

Shuai Wang, Liang Ding, Li Shen, Yong Luo, Bo Du, and Dacheng Tao. Oop: Object-oriented programming evaluation benchmark for large language models. arXiv preprint arXiv:2401.06628, 2024b.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In Forty-first International Conference on Machine Learning.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pp. 8696–8708, 2021.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. Codet5+: Open code large language models for code understanding and generation. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp. 1069–1088, 2023b.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. In Findings of the Association for Computational Linguistics: EMNLP 2023, pp. 1271–1290, 2023c.

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

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

Karl E Wiegers and Joy Beatty. Software requirements. Pearson Education, 2013.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. arXiv preprint arXiv:2309.07864, 2023.

Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. arXiv preprint arXiv:2304.00385, 2023.

Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Berkeley function calling leaderboard. https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html, 2024.

Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Shuiguang Deng, et al. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. arXiv preprint arXiv:2311.08588, 2023.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering.

Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, et al. Yi: Open foundation models by 01. ai. arXiv preprint arXiv:2403.04652, 2024.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao 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, pp. 1–12, 2024.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. Cert: Continual pre-training on sketches for library-oriented code generation.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Wang Yongji, and Jian-Guang Lou. When language model meets private library. In Findings of the Association for Computational Linguistics: EMNLP 2022, pp. 277–288, 2022a.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. Cert: continual pre-training on sketches for library-oriented code generation. arXiv preprint arXiv:2206.06888, 2022b.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large language models meet nl2code: A survey. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 7443–7464, 2023.

Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. Naturalcodebench: Examining coding performance mismatch on humaneval and natural user prompts. arXiv preprint arXiv:2405.04520, 2024.

Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. How do python framework apis evolve? an exploratory study. In 2020 ieee 27th international conference on software analysis, evolution and reengineering (saner), pp. 81–92. IEEE, 2020.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. Advances in Neural Information Processing Systems, 36, 2024a.

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

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

Xin Zhout, Kisub Kim, Bowen Xu, Jiakun Liu, DongGyun Han, and David Lo. The devil is in the tails: How long-tailed code distributions impact large language models. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 40–52. IEEE, 2023.

Qihao Zhu, Qingyuan Liang, Zeyu Sun, Yingfei Xiong, Lu Zhang, and Shengyu Cheng. Grammart5: Grammar-integrated pretrained encoder-decoder neural model for code. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1–13, 2024.

Terry Yue Zhuo, Armel Zebaze, Nitchakarn Suppattarachai, Leandro von Werra, Harm de Vries, Qian Liu, and Niklas Muennighoff. Astraios: Parameter-efficient instruction tuning code large language models. arXiv preprint arXiv:2401.00788, 2024.

APPENDIX

Contents

A CONTRIBUTIONS

Project Leadership Terry Yue Zhuo

Benchmark Construction Chien Vu, Jenny Chim, Han Hu, Haolan Zhan, Xiaoheng Hong, Wenhao Yu, Niklas Muennighoff, Jean Kaddor, Wen-Ding Li, Junda He, Ming Xu, Zhihan Zhang, Ratnadira Widyasari, Indraneil Paul, Simon Brunner, Imam Nur Bani Yusuf, Thong Hoang, Chen Gong, Armel Zebaze, Prateek Yadav, Terry Yue Zhuo

Experiment Terry Yue Zhuo, Binyuan Hui

Evaluation Framework Terry Yue Zhuo, Indraneil Paul, Jiawei Liu, Zhoujun Cheng

Analysis Terry Yue Zhuo, Binyuan Hui, Zhoujun Cheng, Alex Gu, Naman Jain

Paper Writing Terry Yue Zhuo

Presentation Editing Niklas Muennighoff, Indraneil Paul, David Lo, Zijian Wang, Daniel Fried, Binyuan Hui, Qian Liu, Jean Kaddor, Jiawei Liu, Imam Nur Bani Yusuf, Chen Gong

Long-term Advice & Support Daniel Fried, Niklas Muennighoff, Qian Liu, Zijian Wang, Binyuan Hui, Xiaoning Du, David Lo, Jiawei Liu, Harm de Vries, Leandro von Werra

B DATACARD

We follow (Bender & Friedman, 2018) to create the datacard for `BigCodeBench`, where we tend to summarize and centralize all information that might be relevant for the benchmark analysis.

Curation Rationale This is detailed in Section 2 and Appendix J.

Language Variety Information about our annotators' nationality will not be provided, as the constructed benchmark is hardly related to regional or social dialects. However, we confirm that all communications during the annotation process are in mainstream English (en-US). We note that the first language of some annotators is not English, which can introduce some inaccurate expressions to the task prompts in `BigCodeBench`.

Curators Demographic The benchmark construction requires the great annotation effort of Curators, who are involved in the process detailed in Section 2. They come from the following population:

- **Age**:
  - 18-25: 25% (5/20)
  - 26-35: 70% (14/20)
  - 36-45: 5% (1/20)
- **Experience in Python Programming (Years)**:
  - 1-3: 5% (1/20)
  - 3-5: 20% (4/20)
  - 5+: 75% (15/20)
- **Academic Background**:
  - Bachelor: 5% (1/20)

  - Master: 20% (4/20)
  - PhD: 75% (15/20)

Text Characteristics This is detailed in Section 3.

C Data Sheet

Besides the provided Datacard, we follow the documentation frameworks provided by (Gebru et al., 2021).

C.1 Motivation

C.1.1 For what purpose was the dataset created?

Our dataset aims to provide a thorough assessment of the capability of solving programming tasks. Particularly, we focus on the challenges and practicability of the tasks, and pinpoint two main characteristics that few benchmarks highlight: (1) Diverse Function Calling; and (2) Complex Instruction Following. This dataset will help stakeholders better understand the fundamental abilities and limitations associated with deploying LLMs.

We believe that there are three main expectations of a good execution-based programming benchmark:

- The benchmark should be easy to use and efficient in evaluating the fundamental capabilities of LLMs. Repository-level benchmarks (e.g., SWE-bench (Yang et al.)) are not suitable for this purpose.
- The benchmark should be practical, covering various programming scenarios. Algorithm-specific benchmarks (e.g., HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021)) are unsuitable. Domain-specific benchmarks (e.g., DS-1000 (Lai et al., 2023)) are also unsuitable for this purpose.
- The benchmark should be challenging, where the tasks require LLMs' strong compositional reasoning capabilities and instruction-following capabilities. The benchmarks with simple tasks (e.g., ODEX (Wang et al., 2023c)) are unsuitable.

`BigCodeBench` is the first benchmark that meets all three expectations. It is an easy-to-use benchmark that evaluates LLMs with practical and challenging programming tasks, accompanied by an end-to-end evaluation framework `BigCodeBench`. We aim to assess how well LLMs can solve programming tasks in an open-ended setting.

C.2 Composition/Collection Process/Preprocessing/Cleaning/Labeling and Use

The answers are described in our paper as well as the GitHub repository: https://github.com/bigcode-project/bigcodebench-annotation.

C.3 Distribution

C.3.1 Will the dataset be distributed to third parties outside of the entity (e.g., company, institution, organization) on behalf of which the dataset was created?

No. Our dataset will be managed and maintained by the BigCode community (https://www.bigcode-project.org/).

C.3.2 How will the dataset be distributed (e.g., tarball on website, API, GitHub)?

The evaluation dataset will be released to the public, and hosted on Hugging Face.

C.3.3 When will the dataset be distributed?

It has been released now.

C.3.4 Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)?

Our dataset is distributed under the Apache-2.0 license.

C.4 Maintenance

C.4.1 How can the owner/curator/manager of the dataset be contacted (e.g., email address)?

Please contact Terry Yue Zhuo (`terry.zhuo@monash.edu`) and the BigCode Project (`contact@bigcode-project.org`), who are responsible for maintenance.

C.4.2 Will the dataset be updated (e.g., to correct labeling errors, add new instances, delete instances)?

Yes. If we include more tasks or find any errors, we will correct the dataset hosted on Hugging Face and update the results in the leaderboard accordingly. It will be updated on our website.

C.4.3 If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so?

For dataset contributions and evaluation modifications, the most efficient way to reach us is via GitHub pull requests. For more questions, contact Terry Yue Zhuo (`terry.zhuo@monash.edu`) and the BigCode Project (`contact@bigcode-project.org`), who are responsible for maintenance.

D Data Contamination

While `BigCodeBench` tasks are constructed from scratch, there are still some concerns regarding the potential data contamination. Therefore, we conduct N-gram contamination experiments on ODEX intents (English) and an anonymized archive of Stack Overflow used by StarCoder2 (Lozhkov et al., 2024), which may be correlated to `BigCodeBench` instructions. We also evaluated StarCoderData (Python) (Li et al., 2023), which has been widely used as the code training data for various LLMs. We focus on the overlaps between the `BigCodeBench` instructions and the queries contained by these data sources, using 10-gram and 13-gram setups (Brown, 2020; Elazar et al., 2023; Shao et al., 2024; Guo et al., 2024; Bai et al., 2023) to indicate potential data contamination. Due to the significant computational resources required, the 10-gram overlap on StarCoderData is timeout and thus omitted.

Table 3: Comparison of N-Gram data across different datasets.

N-Gram	Overlap Percentage (%)		
	ODEX	**Stack Overflow**	**StarCoderData**
13	0.00	0.18	—
10	0.09	1.49	2.49

As shown in Table 3, the likelihood of our task descriptions being contaminated by existing data is extremely low. With a stricter 10-gram configuration, no more than 2.5% of `BigCodeBench` tasks overlapped with the tested data sources.

In addition, we have carefully considered potential future data contamination issues before releasing `BigCodeBench`. To mitigate this, we have decided to release the data on Hugging Face rather than directly on GitHub. Based on past experiences, most contamination stems from the unintentional inclusion of GitHub source code, as seen with datasets like HumanEval and MBPP. Unlike GitHub, Hugging Face does not support the kind of automated scraping that typically leads to contamination, making BigCodeBench relatively safer from this issue. That said, we acknowledge that it is impossible to ensure complete privacy of benchmark data. For instance, when closed-source model APIs are used for inference, companies may collect and use the data for training if it is deemed high-quality. Preventing this would require access to their model weights and the ability to run them locally, which is not feasible in most cases.

E Extended Related Work

Large Language Models for Code With the rise of LLMs, various models have been trained on code. The first series of models, like CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al.), are built based on the BERT (Kenton & Toutanova, 2019) encode-only architecture, which specializes in code-understanding tasks such as defect detection and clone detection (Lu et al.). As there is an increasing need for automated code completion, the T5-based encoder-decoder architecture has been applied to train code-specific models like PyMT5 (Clement et al., 2020), PLBART (Ahmad et al., 2021), CodeT5 (Wang et al., 2021), CodeT5+ (Wang et al., 2023b), and GrammarT5 (Zhu et al., 2024). While these models are better at generating code and comments, they are not fully optimized. More LLMs for code have become decoder-only, following the design of GPT (Radford, 2018). The early decoder-only LLMs for code, trained with left-to-right token prediction, cannot assist developers in inserting code between programs. To fulfill this need, InCoder (Radford, 2018) has been designed to allow code infilling. Subsequently, the combination of left-to-right and fill-in-the-middle code generation has been widely adopted in later model development, such as StarCoder (Li et al., 2023), CodeLlama (Roziere et al., 2023), and DeepSeek-Coder (Guo et al., 2024). However, these models are pre-trained only and are denoted as base models. With advances in instruction tuning (Ouyang et al., 2022), more LLMs for code are further fine-tuned on natural language instructions to better understand user queries. For example, WizardCoder (Luo et al.) and MagiCoder (Wei et al., 2024) are instruction-tuned with synthetic data containing diverse instruction-code pairs. Later, OpenCodeInterpreter (Zheng et al., 2024b) developed a multi-turn instruction dataset and achieved better coding performance. More recently, there has been a growing interest in agentic programming (Yang et al.), developing prompting systems to enhance the capabilities of performing software engineering tasks, like GitHub issue resolution and code search. However, they are agnostic regarding the backbone models. In this work, we aim to assess the fundamental capabilities of LLMs in practical code generation without any customized prompting systems.

Programming Benchmarks To assess the programming capability of LLMs, researchers have proposed various programming benchmarks. Most existing benchmarks focus on short, self-contained, algorithm-specific programming tasks, such as HumanEval (Chen et al., 2021) and MBPP (Hendrycks et al., 2021). While there are other programming benchmarks like APPS (Hendrycks et al., 2021) and CodeContests (Li et al., 2022), they are not frequently used due to the lack of an easy-to-use evaluation infrastructure. Additionally, some benchmarks focus on real-world program-based tool use, such as DS-1000 (Lai et al., 2023), ODEX (Wang et al., 2023c), NumpyEval (Zan et al.), and TorchDataEval (Zan et al., 2022a). These benchmarks are designed to evaluate how well LLMs can handle specific libraries and programming scenarios, simulating more realistic and complex coding tasks. However, these benchmarks often focus on simple intents and use a limited number of specific function calls per programming task, making their evaluations less challenging and realistic. Meanwhile, there has been another line of work on multi-programming-language code generation. For example, HumanEval-X (Zheng et al., 2023), MultiPL-E (Cassano et al., 2023), and CodeScope (Yan et al., 2023) extend Python-only programming benchmarks via code translation, without considering programming-language-centric downstream scenarios. While there are some benchmarks addressing language-specific limitations like ClassEval (Du et al., 2023) and OOP (Wang et al., 2024b), their tasks are limited in terms of quantity, complexity, and diversity. Later, benchmarks like RepoBench (Liu et al.), CrossCodeEval (Ding et al., 2024), CoderEval (Yu et al., 2024), and SWE-bench (Yang et al.) are designed to evaluate the performance of a code agent framework, which includes iterated prompting, real-time environment interaction, and long-context exploration. Our benchmark focuses on evaluating the fundamental code generation capability of LLMs, which is also an essential path toward strong code agents. Additionally, SWE-bench is constructed from GitHub repositories with existing test cases, which limits the diversity of the tasks considered. In contrast, our collaborative LLM-human annotator procedure allows for generating tasks driven by real-world software engineering requirements, similar to the queries from StackOverflow, tackling a broader range of software tasks. Furthermore, our benchmark emphasizes some important aspects that have not been well discussed in the programming domain, like open-endedness (Hughes et al., 2024), multi-tool use (Qin et al., 2023), and instruction-following (Lou et al., 2023).

F BigCodeBench-Hard

Running the full set of `BigCodeBench` will be burdensome for common users, especially when evaluating a large model on both `BigCodeBench-Complete` and `BigCodeBench-Instruct`. In order to save budgets, we release a minimal high-quality subset of `BigCodeBench-Hard`, serving as a proxy for the full set.

As illustrated in Figure 8, the workflow to construct `BigCodeBench-Hard` is mainly inspired by MixEval (Ni et al., 2024), which utilizes a small number of benchmark samples to align user-facing evaluation. While MixEval focuses on general-domain evaluation and considers only code generation tasks with minimal samples from MBPP and HumanEval, we extend the idea to make code generation evaluation more user-centric. Specifically, we follow these steps to create `BigCodeBench-Hard`:

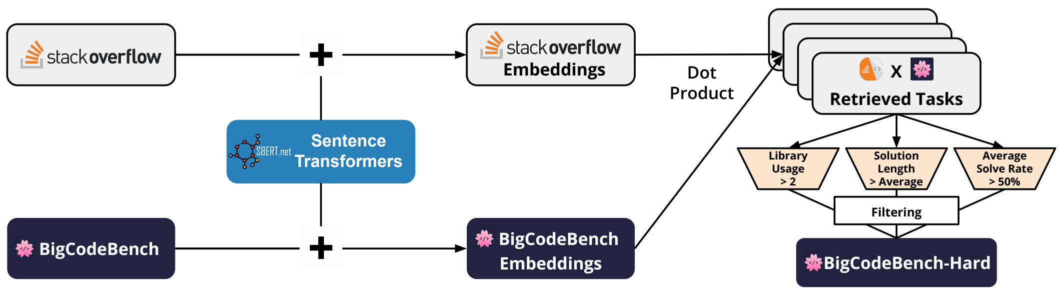


Figure 8: `BigCodeBench-Hard` construct workflow.

First, we choose an anonymized archive of Stack Overflow that has been preprocessed by the BigCode community. Details of the preprocessing can be found in the StarCoder2 (Lozhkov et al., 2024). The archive contains 10.4 million questions and answers, covering diverse programming languages and topics, making it a good source of user queries.

To bridge the query source and `BigCodeBench`, we leverage `all-mpnet-base-v2`, a pre-trained sentence embedding model recommended by the Sentence Transformers documentation (Reimers, 2019). This model, trained on a mixture of text and code data, is suitable for identifying similarities between programming queries and `BigCodeBench` tasks.

We use the model to retrieve the most similar tasks for each query in the Stack Overflow archive, ranking them by the dot product between normalized embeddings. Based on manual inspection of the retrieved tasks, we conclude that a similarity score above 0.7 is a good threshold for task selection. By applying this threshold, we obtain 6,895 queries and 626 `BigCodeBench` tasks after deduplication.

We illustrate the alignment between the Stack Overflow queries (Figure 9) and the `BigCodeBench` tasks (Figure 10). As shown in the figures, both the query and the task prompt revolve around web scraping to extract hyperlinks from web pages, using Python libraries to handle HTTP requests and parse HTML. Both involve interaction with CSV files to either read input URLs or write output data. While the specific implementation details differ, the core objective of extracting and handling hyperlink data from web pages is a shared aspect, aligning their overall scope closely.

However, the retrieved 626 tasks are still infeasible for evaluation. To improve evaluation efficiency, we further filter the tasks by difficulty. Unlike the construction of MixEval-Hard, we define the following more explainable criteria: (1) **Library Usage**: Each task in `BigCodeBench` emphasizes the compositional reasoning ability for coding and requires the use of at least two libraries. For `BigCodeBench-Hard`, we keep only the tasks that require more than two libraries, challenging the models to choose more diverse function calls as tools to solve the tasks; (2) **Solution Length**: We set the threshold at 426 tokens, which is the average solution length of the tasks in `BigCodeBench`. The ground-truth solution provides a reference for the task complexity, and tasks with longer solutions are more challenging to solve; and (3) **Solve Rate**: We compute the solve rate per task based on all

````
I have a Python script that imports a list of url's from a CSV named list.csv, scrapes them and outputs
any anchor text and href links found on each url from the csv:

(For reference the list of urls in the csv are all in column A)

```
from requests_html import HTMLSession
from urllib.request import urlopen
from bs4 import BeautifulSoup
import requests
import pandas
import csv

contents = []
with open('list.csv','r') as csvf: # Open file in read mode
    urls = csv.reader(csvf)
    for url in urls:
        contents.append(url) # Add each url to list contents

for url in contents:
    page = urlopen(url[0]).read()
    soup = BeautifulSoup(page, "lxml")

    for link in soup.find_all('a'):
        if len(link.text)>0:
            print(url, link.text, '-', link.get('href'))

```

The output results look something like this where <https:\/\/www.example.com\/csv-url-one\/> and
<https:\/\/www.example.com\/csv-url-two\/> are the url's in column A in the csv:

```
['https:\/\/www.example.com\/csv-url-one\/'] Creative - https:\/\/www.example.com\/creative\/
['https:\/\/www.example.com\/csv-url-one\/'] Web Design - https:\/\/www.example.com\/web-design\/
['https:\/\/www.example.com\/csv-url-two\/'] PPC - https:\/\/www.example.com\/ppc\/
['https:\/\/www.example.com\/csv-url-two\/'] SEO - https:\/\/www.example.com\/seo\/

```

The issue is i want the output results to look more like this i.e not repeatedly print the url in the CSV
before each result AND have a break after each line from the CSV:

```
['https:\/\/www.example.com\/csv-url-one\/']
Creative - https:\/\/www.example.com\/creative\/
Web Design - https:\/\/www.example.com\/web-design\/

['https:\/\/www.example.com\/csv-url-two\/']
PPC - https:\/\/www.example.com\/ppc\/
SEO - https:\/\/www.example.com\/seo\/

```

Is this possible?

Thanks
````

Figure 9: StackOverflow user query.

```
import requests
from urllib.parse import urljoin
from bs4 import BeautifulSoup
import csv


def task_func(
    url: str,
    base_url: str = "https:\/\/www.example.com",
    csv_file: str = "scraped_data.csv",
) -> int:
    """
    This function scrapes a webpage for all hyperlinks and saves them as absolute URLs to a CSV file.

    Parameters:
    - url (str): The relative URL of the webpage to scrape.
    - base_url (str, optional): The base URL of the website to prepend to relative links. Defaults to
'https:\/\/www.example.com'.
    - csv_file (str, optional): The filename for the CSV file where the links will be saved. Defaults to
'scraped_data.csv'.

    Returns:
    - int: The number of unique absolute links scraped from the webpage.

    Requirements:
    - requests
    - urllib.parse.urljoin
    - bs4.BeautifulSoup
    - csv

    Examples:
    >>> task_func('\/mywebpage')
    5
    >>> task_func('\/anotherpage', base_url='https:\/\/www.different.com', csv_file='other_links.csv')
    8
    """
```

Figure 10: `BigCodeBench` task prompt.

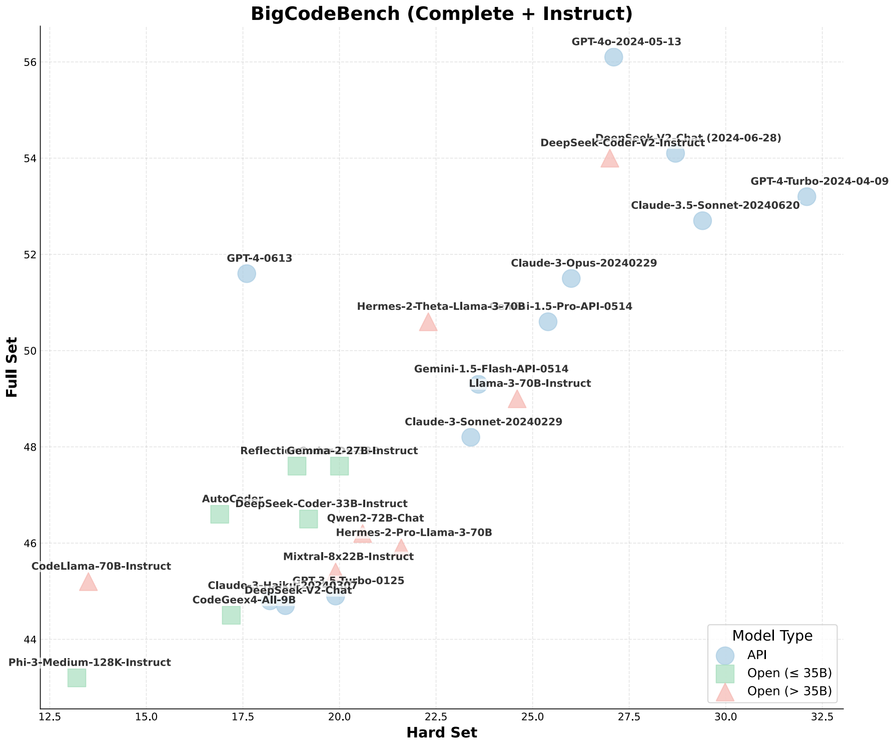


Figure 11: `BigCodeBench` rankings v.s. `BigCodeBench-Hard` rankings

the evaluated models on the leaderboard. The solve rate is defined as the number of models that can solve the task divided by the total number of models. Specifically, we deem tasks with a solve rate below 50% as hard tasks.

Through comparison, we notice that the model performance on `BigCodeBench-Hard` differs significantly from the one on the full set of `BigCodeBench`. We suggest that these differences arise from the imbalanced distribution of target domains and a large number of easy tasks in `BigCodeBench`, resulting in a slight misalignment between the evaluation and user-facing tasks. For example, GPT-4o-2024-05-13 and GPT-4-0613 may be overfitting to the easy tasks in `BigCodeBench`, leading to low performance on `BigCodeBench-Hard`.

To validate the effectiveness of `BigCodeBench-Hard`, we use a private leaderboard, SEAL-Coding curated by Scale AI, as a reference. The SEAL-Coding leaderboard is designed to evaluate models on a set of user-facing tasks across various application domains and programming languages. Specifically, SEAL-Coding compares four of the best closed LLMs on Python, with the following rankings: (1) **GPT-4-Turbo Preview**, (2) **Claude 3.5 Sonnet**, (3) **GPT-4o**, and (4) **Gemini 1.5 Pro (May 2024)**. These rankings align with our results based on the average score of the Complete and Instruct splits of `BigCodeBench-Hard`, indicating that `BigCodeBench-Hard` is more user-centric and challenging for model evaluation.

We encourage the community to use `BigCodeBench-Hard` when the budget is limited, and the evaluation needs to be more user-centric. Additionally, we note that `BigCodeBench-Hard` can be dynamic by design, depending on user queries and the evaluated models. We can periodically update `BigCodeBench-Hard` to keep the evaluation challenging and user-centric.

For dataset contributions and evaluation modifications, the most efficient way to reach us is via GitHub pull requests. For more questions, please contact `terry.zhuo@monash.edu` and **contact@bigcode-project.org** Project, who are responsible for maintenance.

G LONG-TERM ROADMAP AND CALL FOR CONTRIBUTIONS

In this section, we share a long-term roadmap to address the limitations of BigCodeBench, and sustainably build with the community. We believe that program-aided language models (Gao et al., 2023) for task completion and reasoning provide the possibility towards artificial general intelligence. Our goal is to provide the community with the most open, reliable, and scalable evaluations to truly understand the fundamental capabilities of LLMs for programming, pinpointing the ways to unleash their power.

G.1 LIMITATIONS

Given the limited time and budget we have to develop the initial benchmark, we have foreseen several limitations and aim to address them step-by-step.

Multilingualism One of the main limitations is that BigCodeBench is Python-only and cannot be easily extended to other programming languages. As the function calls are mostly language-specific, it is hard to find a package or library with exactly the same functionalities other than Python. However, given the fact that Python is the most flexible and popular programming language to automate various tasks, BigCodeBench may fulfill most of the community needs. In the meantime, we still seek efficient approaches to construct BigCodeBench-like programming tasks with tools in other languages, without much human effort.

Saturation Another potential criticism is that some LLMs can still perform reasonably well BigCodeBench, considering that the best models can only resolve no more than 30% of real-world GitHub issues on SWE-bench. One might indicate that our benchmark is not challenging enough. However, we note that the low performance on SWE-bench is likely due to the under-specified instructions and misaligned test cases[5]. Compared to SWE-bench, we tend to make the programming tasks much less ambiguous and ensure that the authors can pass their own solutions during annotation.

Reliability During the execution-based evaluation, we notice that some test cases are flaky (Luo et al., 2014), which results in uncertainty across multiple test runs without any code changes. We have progressively resolved some identified issues, such as missing setup of random states and improper removal of non-existent files. However, the remaining cases are trickier. For example, the socket query can be timed out or refused due to the unstable connection. With that being said, we try our best to make the uncontrollable changes of Pass@1 under 0.6%. We plan to continually enhance the reliability of all test cases, with the help of the community. To maintain the high reproducibility, we host a real-time code execution sandbox in the Hugging Face space.

Rigorousness While we achieve high test coverage for the ground-truth solutions in BigCodeBench, it does not guarantee that any code generated by LLMs will be correctly assessed against existing test cases. Previous works like EvalPlus (Liu et al., 2024) have attempted to extend the limited test cases by augmenting the input-output pairs via LLM- and mutation-based strategies. However, it is challenging to adapt EvaPlus to the test harness in BigCodeBench, as the harness only examines the expected program behaviors during the runtime (e.g., mocking tests). Furthermore, the function calls used to pass test cases by LLMs are more nondeterministic, making traditional test generation (Anand et al., 2013) methods cover all possible scenarios. Therefore, we still consider LLM-based test generation (Wang et al., 2024a) promising, but with proper designs. Specifically, a possible approach is to collect all the generated solutions that pass the current test cases and make a capable LLM (e.g., GPT-4o) harness with self-refinement in a sandbox environment.

Generalization One intuitive question is "How well do the models generalize to the unseen tools and tasks?" Current BigCodeBench only covers the common libraries and daily programming tasks. It will be more interesting to benchmark models on the programming tasks that use emerging libraries like transformers and langchain. Crafting new high-quality programming tasks requires huge effort, as demonstrated in this paper. There are two efficient approaches for building a programming benchmark for model generalization. First, we can instruction tune an LLM on

[5] https://github.com/princeton-nlp/SWE-bench/issues/72

BigCodeBench data with additional information of libraries and function calls. The trained model is expected to generate programming tasks with proper test cases based on the given library or function call details. However, the quality of such data synthesis is unknown, making the practicability questionable. Another way is to replace the function calls and libraries in BigCodeBench with the synthetic names, simulating the unseen ones. A similar approach (Zan et al., 2022b) has been used for code generation on unknown libraries. Although this method may lose the naturalness of software development, the construction process is more controllable and practical.

Evolution Naturally, the libraries can be obsolete or updated (Lamothe et al., 2021), which means that the source code data for model training will constantly evolve. Thus, the models may not memorize function calls from a deprecated library version. This poses a challenge for any tool-dependent programming benchmarks to correctly examine the model capability without periodic updates. Another related concern is the test set contamination issue due to the evolving training data. It is suggested to have both a public set and a private set for better evaluation in a recent blog[6]. For future releases, we aim to perform the benchmark evolution both publicly and privately, and host the private test set internally.

Interaction Recent interests are around the concept of *LLMs as Agents* (Xi et al., 2023), which is deemed as a way towards artificial general intelligence. Specifically, LLMs will be grounded in a less constrained sandbox environment, where they can interact with any given applications, such as the web browser and terminal. The environment can help unlock the capabilities like self-debugging (Chen et al., 2023) and self-reflection (Shinn et al., 2024). We tend to work in this direction and see how well *LLMs as Agents* can perform on BigCodeBench.

Diversity There might be some concerns regarding the diversity of Python libraries covered by BigCodeBench. While we agree that BigCodeBench can be extended with more domain-specific libraries like the ones in SciCode (Tian et al., 2024), we believe that the usage of popular libraries can be more aligned with real-world software development and maximally benefit the community. In addition, due to the limited number of domain experts participating in our data annotation, they may not have enough knowledge of the latest or domain-specific libraries. Annotating the programming tasks involved with such libraries may result in errors and ambiguities. Furthermore, the emerging libraries are immature and usually propose breaking changes, posing challenges to constructing programming tasks with stable APIs.

G.2 BigCodeBench-OOD

We will construct BigCodeBench-OOD, a variant of BigCodeBench that assesses the generalization ability on out-of-distribution (OOD) libraries and programming tasks (Zhout et al., 2023). We note that any features that do not frequently appear can be considered OOD. We tend to build BigCodeBench-OOD with two different types of OOD: (1) Completely synthetic features; and (2) Real but long-tail features. The first type can represent the private libraries that are unknown to the models, and the latter one may represent the ones trending in the recent programming practice but have not yet played major roles in the training data.

G.3 BigCodeBench-Interact

To examine the agent-centric programming ability, we will adapt BigCodeBench to an interactive sandbox environment, with a list of applications that can help models program and repair iteratively as humans. Unlike existing programming benchmarks such as HumanEval and MBPP, BigCodeBench covers a wide range of tasks that require profound programming knowledge. Without the proper understanding, models are likely to fail in debugging and repair. There are potentially two options for the evaluation. One naive option is to allow models to see the backtrace of how they fail in the test cases, which can guide them to repair the bugs with a clear goal. A more challenging setup is to make models fix bugs independently without seeing any backtrace. Capable models are expected to resolve issues by generating their own test cases and inspecting the execution outcomes.

[6] https://www.jasonwei.net/blog/evals

G.4 BIGCODEBENCH-EVOLVED

To mitigate the effects of API evolution, we consider LLM-based API updates as a promising approach to explore. However, several challenges need to be addressed from the software engineering aspect. First, there are many types of API evolution in Python. According to the study on 288 releases of six popular Python libraries (Zhang et al., 2020), there are 14 different types of API evolution (e.g., class removal, method removal, parameter reordering, and field addition). Among them, 11 types are breaking changes, meaning they will directly affect the program's behavior. Resolving the API removal requires non-trivial effort, as the evolved APIs may not have any replacement in the future library version. Second, Wang et al. (2020) mention that a significant number of deprecated APIs are not documented properly, making it hard for library users to mitigate the usage of deprecated APIs. To potentially mitigate all these issues, we suggest combining static program analysis for library versions and multi-turn LLM interaction. Specifically, we can utilize static program analysis to compare APIs across versions and make LLMs to identify the potential changes, even if they are not officially documented. Then, we will apply rule-based methods to analyze whether the target programming tasks inside BigCodeBench have deprecated APIs. For the ones to be updated, we will further provide LLMs with the new library documentation and identified API changes, let them propose new implementations for the ground-truth solutions, and revise the deprecated APIs inside test cases. Regarding the cases of API removal, LLMs may need more turns to generate the valid implementation, as there is no reference for the replacement. The revision To automatically validate the updated solutions and test cases, we will ground LLMs inside the code sandbox for multi-turn execution.

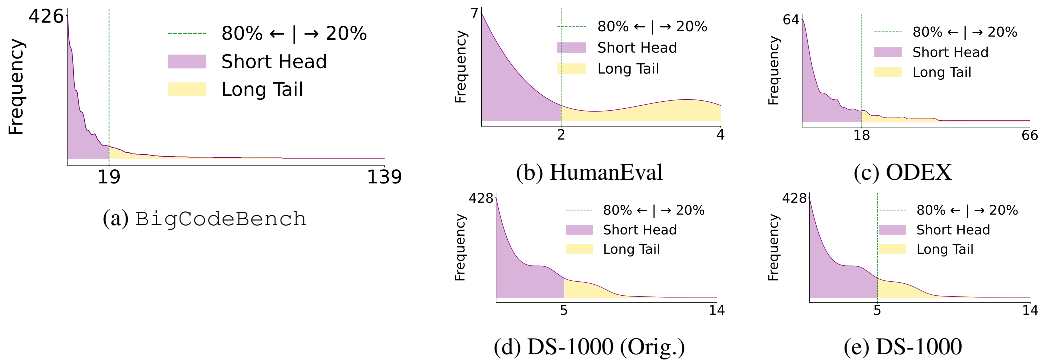
H ARTIFACTS

Table 4: Artifacts for reproducibility.

Name	**Public Link or Endpoint**
	`BigCodeBench` *(v0.2.4)*
GitHub	https://github.com/bigcode-project/bigcodebench/releases/tag/v0.2.4
Hugging Face	https://huggingface.co/datasets/bigcode/bigcodebench
Croissant	https://huggingface.co/api/datasets/bigcode/bigcodebench/croissant
	Annotation Framework
GitHub	https://github.com/bigcode-project/bigcodebench-annotation
	Evaluation Framework
GitHub	https://github.com/bigcode-project/bigcodebench
PyPI	https://pypi.org/project/bigcodebench/
	Datasets for Comparisons
HumanEval	https://github.com/openai/human-eval/blob/master/data/HumanEval.jsonl.gz
ODEX	https://github.com/zorazrw/odex/tree/master/data
DS-1000	https://github.com/xlang-ai/DS-1000/blob/main/data/ds1000.jsonl.gz
APPS	https://github.com/hendrycks/apps
APIBench	https://github.com/ShishirPatil/gorilla/tree/main/data/apibench
MBPP	https://github.com/google-research/google-research/tree/master/mbpp
NumpyEval	https://github.com/microsoft/PyCodeGPT/tree/main/cert/pandas-numpy-eval
PandasEval	https://github.com/microsoft/PyCodeGPT/tree/main/cert/pandas-numpy-eval
TorchDataEval	https://github.com/microsoft/PyCodeGPT/blob/main/apicoder/private-eval/data/real_torchdata_eval_v3.jsonl.gz
	Models for Evaluations
GPT-4o	gpt-4-turbo-2024-04-09
GPT-4-Turbo	gpt-4-0125-preview
GPT-4	gpt-4-0613
GPT-3.5-Turbo	gpt-3.5-turbo-0125
Claude-3-Opus	claude-3-opus-20240229
Claude-3-Sonnet	claude-3-sonnet-20240229
Claude-3-Haiku	claude-3-haiku-20240307
DeepSeek-V2-Chat	deepseek-chat (API, 32K)
Mistral-Large	mistral-large-2402
Mistral-Small	mistral-small-2402
DeepSeek-Coder-33b-Instruct	https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct
DeepSeek-Coder-6.7b-Instruct	https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct
DeepSeek-Coder-1.3b-Instruct	https://huggingface.co/deepseek-ai/deepseek-coder-1.3b-instruct
CodeLlama-70B-Instruct	https://huggingface.co/codellama/CodeLlama-70b-Instruct-hf
CodeLlama–34B-Instruct	https://huggingface.co/codellama/CodeLlama-34b-Instruct-hf
CodeLlama-13B-Instruct	https://huggingface.co/codellama/CodeLlama-13b-Instruct-hf
CodeLlama-7B-Instruct	https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf
Llama-3-70B-Instruct	https://huggingface.co/meta-llama/Meta-Llama-3-70B-Instruct
Llama-3-8B-Instruct	https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct
Qwen2-72B-Instruct	https://huggingface.co/Qwen/Qwen2-72B-Instruct
Qwen2-7B-Instruct	https://huggingface.co/Qwen/Qwen2-7B-Instruct
Qwen2-57B-A14B	https://huggingface.co/Qwen/Qwen2-57B-A14B
CodeQwen1.5-7B-Chat	https://huggingface.co/Qwen/CodeQwen1.5-7B-Chat
Qwen1.5-110B-Chat	https://huggingface.co/Qwen/Qwen1.5-110B-Chat
Qwen1.5-72B-Chat	https://huggingface.co/Qwen/Qwen1.5-72B-Chat
Qwen1.5-32B-Chat	https://huggingface.co/Qwen/Qwen1.5-32B-Chat
Yi-1.5-34B-Chat	https://huggingface.co/01-ai/Yi-1.5-34B-Chat
Yi-1.5-9B-Chat	https://huggingface.co/01-ai/Yi-1.5-9B-Chat
Yi-1.5-6B-Chat	https://huggingface.co/01-ai/Yi-1.5-6B-Chat
Mixtral-8x22B-Instruct	https://huggingface.co/mistralai/Mixtral-8x22B-Instruct-v0.1
Magicoder-S-DS	https://huggingface.co/ise-uiuc/Magicoder-S-DS-6.7B
CodeGemma-7B-Instruct	https://huggingface.co/google/codegemma-7b-it
StarCoder2-Instruct	https://huggingface.co/bigcode/starcoder2-15b-instruct-v0.1
Granite-34B-Code-Instruct	https://huggingface.co/ibm-granite/granite-34b-code-instruct
Granite-20B-Code-Instruct	https://huggingface.co/ibm-granite/granite-20b-code-instruct
Granite-8B-Code-Instruct	https://huggingface.co/ibm-granite/granite-8b-code-instruct
Granite-3B-Code-Instruct	https://huggingface.co/ibm-granite/granite-3b-code-instruct
CodeLlama-70B-Base	https://huggingface.co/codellama/CodeLlama-70b-hf
CodeLlama-34B-Base	https://huggingface.co/codellama/CodeLlama-34b-hf
CodeLlama-13B-Base	https://huggingface.co/codellama/CodeLlama-13b-hf
CodeLlama-7B-Base	https://huggingface.co/codellama/CodeLlama-7b-hf
Llama-3-70B-Base	https://huggingface.co/meta-llama/Meta-Llama-3-70B
Llama-3-8B-Base	https://huggingface.co/meta-llama/Meta-Llama-3-8B
DeepSeek-Coder-base-33B	https://huggingface.co/deepseek-ai/deepseek-coder-33b-base
DeepSeek-Coder-base-6.7B	https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-base
DeepSeek-Coder-base-1.3B	https://huggingface.co/deepseek-ai/deepseek-coder-1.3b-base
CodeQwen1.5-7b	https://huggingface.co/Qwen/CodeQwen1.5-7B
Yi-1.5-34B	https://huggingface.co/01-ai/Yi-1.5-34B
Yi-1.5-9B	https://huggingface.co/01-ai/Yi-1.5-9B
Yi-1.5-6B	https://huggingface.co/01-ai/Yi-1.5-6B
Mixtral-8x22B-Base	https://huggingface.co/mistralai/Mixtral-8x22B-v0.1
CodeGemma-7B	https://huggingface.co/google/codegemma-7b
CodeGemma-2B	https://huggingface.co/google/codegemma-2b
StarCoder2-15B	https://huggingface.co/bigcode/starcoder2-15b
StarCoder2-7B	https://huggingface.co/bigcode/starcoder2-7b
StarCoder2-3B	https://huggingface.co/bigcode/starcoder2-3b
Granite-34B-Code-Base	https://huggingface.co/ibm-granite/granite-34b-code-base
Granite-20B-Code-Base	https://huggingface.co/ibm-granite/granite-20b-code-base
Granite-8B-Code-Base	https://huggingface.co/ibm-granite/granite-8b-code-base
Granite-3B-Code-Base	https://huggingface.co/ibm-granite/granite-3b-code-base

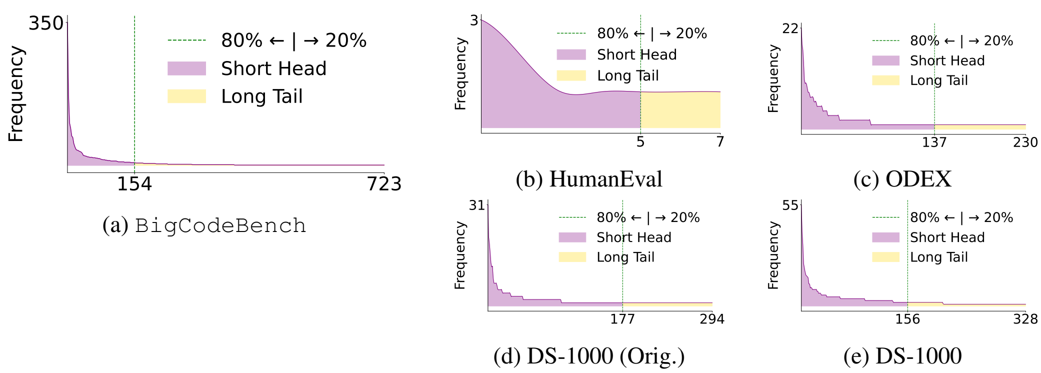
I Tool Statistics

I.1 Analysis



(a) BigCodeBench (b) HumanEval (c) ODEX (d) DS-1000 (Orig.) (e) DS-1000

Figure 12: Library density comparisons. We sort the libraries by frequency count, showcasing the long-tail distribution to highlight the broad diversity within `BigCodeBench`.



(a) BigCodeBench (b) HumanEval (c) ODEX (d) DS-1000 (Orig.) (e) DS-1000

Figure 13: Function call density comparisons. We sort function calls by frequency count, showcasing the long-tail distribution to highlight the broad diversity within `BigCodeBench`.

I.2 Comparison to Existing Programming Benchmarks

Table 5: Depth (Complexity — Solution Characters) and breath (Diversity – Function Calls) comparisons to existing programming benchmarks in Python.

Benchmark	**Complexity**	**Diversity**
APPS (Hendrycks et al., 2021)	352.9	137
DS-1000 (Lai et al., 2023)	138.1	328
ODEX (Wang et al., 2023c)	50.4	230
APIBench (Patil et al., 2023)	77.5	171
MBPP (Austin et al., 2021)	181.1	51
NumpyEval (Zan et al., 2022b)	30.1	52
PandasEval (Zan et al., 2022b)	45.8	12
HumanEval (Chen et al., 2021)	180.9	7
TorchDataEval (Zan et al., 2022a)	52.8	6
`BigCodeBench`	426.0	723

I.3 Version Control

```
pandas==2.0.3
scikit-learn==1.3.1
requests==2.31.0
matplotlib==3.7.0
seaborn==0.13.2
numpy==1.21.2
numba==0.55.0
cryptography==38.0.0
scipy==1.7.2
nltk==3.8
pytz==2023.3.post1
networkx==2.6.3
statsmodels==0.14.0
lxml==4.9.3
psutil==5.9.5
Django==4.2.7
selenium==4.15.
Pillow==10.3.0
beautifulsoup4==4.8.2
datetime==5.5
python-docx==1.1.0
openpyxl==3.1.2
Levenshtein==0.25.0
PyYAML==6.0.1
wordninja==2.0.0
Faker==20.1.0
tensorflow==2.11.1
wordcloud==1.9.3
pytesseract==0.3.10
chardet==5.2.0
python-dateutil==2.9.0
blake3==0.4.1
dnspython==2.6.1
flask==3.0.3
Flask-Mail==0.9.1
flask_login==0.6.3
flask_restful==0.3.10
flask_wtf==1.2.1
folium==0.16.0
geopy==2.4.1
keras==2.11.0
librosa==0.10.1
mechanize==0.4.9
prettytable==3.10.0
pycryptodome==3.14.1
python_http_client==3.3.7
Requests==2.31.0
requests_mock==1.11.0
rsa==4.9
sendgrid==6.11.0
soundfile==0.12.1
texttable==1.7.0
Werkzeug==3.0.1
WTForms==3.1.2
xlrd==2.0.1
xlwt==1.3.0
xmltodict==0.13.0
python-Levenshtein-wheels
gensim==4.3.2
sympy==1.12
pyfakefs==5.4.1
textblob==0.18.0
docxtpl==0.11.5
statsmodels==0.14.0
pyquery==1.4.3
holidays==0.29
scikit-image==0.18.0
natsort==7.1.1
shapely==2.0.4
geopandas==0.13.2
opencv-python-headless==4.9.0.80
xlrd==2.0.1
pytest==8.2.0
wikipedia==1.4.0
```

I.4 Domain Classification


```
{
    "Crypto": "Cryptography",
    "PIL": "Visualization",
    "array": "General",
    "base64": "Cryptography",
    "binascii": "Cryptography",
    "bisect": "General",
    "blake3": "Cryptography",
    "bs4": "Network",
```

```
    "calendar": "Time",
    "cgi": "Network",
    "chardet": "Network",
    "cmath": "Computation",
    "codecs": "Cryptography",
    "collections": "General",
    "cryptography": "Cryptography",
    "csv": "System",
    "ctypes": "System",
    "datetime": "Time",
    "dateutil": "Time",
    "difflib": "General",
    "django": "Network",
    "docx": "System",
    "email": "Network",
    "faker": "General",
    "flask": "Network",
    "flask_login": "Network",
    "flask_mail": "Network",
    "flask_restful": "Network",
    "fnmatch": "General",
    "folium": "Visualization",
    "functools": "General",
    "geopy": "Network",
    "getpass": "System",
    "glob": "System",
    "gzip": "System",
    "hashlib": "Cryptography",
    "heapq": "General",
    "hmac": "Cryptography",
    "html": "Network",
    "http": "Network",
    "importlib": "General",
    "inspect": "General",
    "io": "System",
    "ipaddress": "Network",
    "itertools": "General",
    "json": "System",
    "keras": "Computation",
    "librosa": "Computation",
    "logging": "System",
    "lxml": "Network",
    "math": "Computation",
    "matplotlib": "Visualization",
    "mechanize": "Network",
    "mimetypes": "Network",
    "multiprocessing": "System",
    "nltk": "Computation",
    "numpy": "Computation",
    "openpyxl": "System",
    "operator": "General",
    "os": "System",
    "pandas": "Computation",
    "pathlib": "System",
    "pickle": "System",
    "pkgutil": "General",
    "platform": "System",
    "prettytable": "General",
    "psutil": "System",
    "pytesseract": "Computation",
    "pytz": "Time",
    "queue": "General",
    "random": "General",
    "re": "General",
    "requests": "Network",
    "rsa": "Cryptography",
    "scipy": "Computation",
    "seaborn": "Visualization",
    "secrets": "Cryptography",
    "select": "System",
    "sendgrid": "Network",
    "shutil": "System",
    "sklearn": "Computation",
    "smtplib": "Network",
    "socket": "Network",
    "soundfile": "Computation",
    "sqlite3": "System",
    "ssl": "Network",
    "statistics": "Computation",
    "statsmodels": "Computation",
    "string": "General",
    "struct": "System",
    "subprocess": "System",
    "sys": "System",
    "tarfile": "System",
    "tensorflow": "Computation",
    "texttable": "General",
    "textwrap": "General",
    "threading": "System",
    "time": "Time",
    "turtle": "Visualization",
    "types": "General",
```

```
    "unicodedata": "General",
    "urllib": "Network",
    "uuid": "General",
    "warnings": "General",
    "werkzeug": "Network",
    "wordninja": "Computation",
    "wtforms": "Network",
    "xlwt": "System",
    "xml": "Network",
    "xmltodict": "Network",
    "yaml": "System",
    "zipfile": "System",
    "Levenshtein": "Computation",
    "ast": "General",
    "configparser": "System",
    "cv2": "Computation",
    "decimal": "General",
    "enum": "General",
    "errno": "System",
    "flask_wtf": "Network",
    "ftplib": "Network",
    "gensim": "Computation",
    "geopandas": "Computation",
    "holidays": "Time",
    "mpl_toolkits": "Visualization",
    "natsort": "General",
    "pyquery": "Network",
    "python_http_client": "Network",
    "regex": "General",
    "shapely": "Computation",
    "shlex": "System",
    "signal": "System",
    "skimage": "Computation",
    "sympy": "Computation",
    "textblob": "Computation",
    "typing": "General",
    "wikipedia": "Network",
    "wordcloud": "Visualization",
    "zlib": "System",
    "aspose": "System",
    "builtins": "General",
    "locale": "System",
    "imp": "System",
    "docxtpl": "System",
    "selenium": "Network",
    "IPython": "Computation",
    "filecmp": "System",
    "multidict": "General",
    "sqlalchemy": "System",
    "obspy": "Computation",
    "pprint": "General",
    "xlrd": "System",
    "argparse": "General",
    "torch": "Computation",
    "copy": "General"
}
```

J Detailed Benchmark Construction

J.1 Data Synthesis Prompt

```
Based on the following simple example, write more complex scenarios and invoke multiple Python libraries to solve each problem.
The written intent should align with a more specific and practical scenario, but should still be easy to do functional correctness assertion.
For each scenario, write a single Python function with the rewritten intent.
Please include requirements and terminal-based input-output examples in the function docstring.
The function should contain complex logic like if-else statements and loops.
You have to use more than three Python libraries for a scenario. Write imports and variable definitions outside the function.
Try to avoid using web APIs if possible.
If there are any constants (e.g. strings and numeric values) used in the functions, you need to declare them before the function.
If data is used, you need to provide sample data in the comment.
Try to return values for correctness assertion.
Each programming scenario and intent should be separated by the special token `GPT_ODEX_BREAK`.

Generate two examples with two scenarios:
{"task_id": 4530069, "prompt": "def f_4530069():\n\treturn ", "suffix": "", "canonical_solution": "datetime.now(pytz.utc)", "test_start": "\nimport pytz\nimport time\nfrom datetime import datetime, timezone\n\ndef check(candidate):", "test": ["\n    assert (candidate() - datetime(1970, 1, 1).replace(tzinfo=timezone.utc)).total_seconds() - time.time() <= 1\n"], "entry_point": "f_4530069", "intent": "get a value of datetime.today() in the UTC time zone", "library": ["datetime", "pytz", "time"]}

Scenario 1:
pandas, pytz, datetime, random, matplotlib
```

````
```python
import pandas as pd
import pytz
from datetime import datetime
from random import randint
import matplotlib.pyplot as plt

# Constants
CITIES = ['New York', 'London', 'Beijing', 'Tokyo', 'Sydney']
WEATHER_CONDITIONS = ['Sunny', 'Cloudy', 'Rainy', 'Snowy', 'Stormy']

# Time zones for the cities
TIMEZONES = {
    'New York': 'America/New_York',
    'London': 'Europe/London',
    'Beijing': 'Asia/Shanghai',
    'Tokyo': 'Asia/Tokyo',
    'Sydney': 'Australia/Sydney'
}

def generate_weather_report(utc_datetime):
    """
    Generate a report of weather conditions for a list of cities across various
    time zones at a given time (UTC).

    Parameters:
    utc_datetime (datetime): The datetime in UTC.

    Returns:
    DataFrame: A pandas DataFrame with weather conditions for the cities.

    Requirements:
    - pandas
    - pytz
    - datetime
    - random
    - matplotlib.pyplot

    Example:
    >>> utc_time = datetime(2023, 6, 15, 12, 0, 0, tzinfo=pytz.UTC)
    >>> report = generate_weather_report(utc_time)
    >>> print(report)
    >>> report['Weather Condition'].value_counts().plot(kind='bar')
    """
    report_data = []

    for city in CITIES:
        city_tz = pytz.timezone(TIMEZONES[city])
        city_time = utc_datetime.astimezone(city_tz)
        weather = WEATHER_CONDITIONS[randint(0, len(WEATHER_CONDITIONS)-1)]
        report_data.append([city, city_time, weather])

    report_df = pd.DataFrame(report_data, columns=['City', 'Local Time', 'Weather Condition'])

    return report_df
```
`GPT_ODEX_BREAK`

Scenario 2:
pytz, datetime, numpy, dateutil
```python
import pytz
from datetime import datetime
import numpy as np
from dateutil.parser import parse

# Constants
LEAP_SECONDS = np.array([1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979, 1980,
                         1981, 1982, 1983, 1985, 1988, 1990, 1993, 1994, 1997,
                         1999, 2006, 2009, 2012, 2015, 2016, 2020])

def total_seconds_since_date(date_str, from_tz, to_tz):
    """
    Calculate the total seconds that have passed since a given datetime from the current time
    in different timezones considering the leap seconds.

    Parameters:
    date_str (str): The date string in "yyyy-mm-dd hh:mm:ss" format.
    from_tz (str): The timezone of the given date string.
    to_tz (str): The timezone to which the current time should be converted.

    Returns:
    int: The total seconds.

    Requirements:
    - datetime
    - pytz
    - numpy
    - dateutil.parser

    Example:
    >>> total_seconds_since_date('1970-01-01 00:00:00', 'UTC', 'America/New_York')
````
````

```
    """
    from_tz = pytz.timezone(from_tz)
    to_tz = pytz.timezone(to_tz)
    given_date = parse(date_str).replace(tzinfo=from_tz)
    current_date = datetime.now().astimezone(to_tz)

    total_seconds = (current_date - given_date).total_seconds()

    leap_years = LEAP_SECONDS[np.logical_and(LEAP_SECONDS >= given_date.year, LEAP_SECONDS <= current_date
      .year)]
    leap_seconds = len(leap_years)

    total_seconds += leap_seconds

    return int(total_seconds)
```
```
Above is the illustration.

Generate five complex scenarios based on the following simple example:
```

J.2 Semi-automatic Program Refactoring and Testing Case Generation

J.2.1 Programming Task Classification Prompt

```
# Choose the most suitable labels for the given program:
SQL
CSV
DataFrames
Time
JSON
XML
HTML
Image
Text
Built-in Data Structure
Analysis
Networking
Processing
Visualization
File Storage
Encryption

# You should output the suitable labels in a list format, such as ["CSV", "DataFrames"].
```

J.2.2 Guidelines

```
## Annotation Guideline:

You are given a function inside "function.py". The goal is to:
1) refine the function including its docstrings in order to make the function more realistic and less
      ambiguous. This means when you see the function stub and docstring, you should be able to implement
      with exactly the same functionality with the given function body;
2) write blackbox unit tests to ensure the functional correctness of the given function. You should also
      make the function easy to test.

### Step1:Check Library Imports

#### Import Statement

- Remove the library imports that are not used in the code.
- Import libraries before the function declaration.

#### Library Usage

- Check if the usage of these libraries is reasonable. For example, if the description asks to complete a
      functionality that can be implemented without any of these libraries, then the usage of these
      libraries APIs is not reasonable. You need to check Step 2 for more details to modify the description
       so that it can make use of the imported libraries.

### Step2: Check Docstring

#### Description

- Check if the expression of the description is clear and concise. If not, you need to modify the
      description to make it clear and concise.
- The description must mention the following five things:
  - Functionality
  - Input
  - Output to be returned
  - Requirements of the imported libraries/modules to be used
  - 1 or 2 examples of the input and output of the function
  - Mention the necessary data structure if the function requires data manipulation.
- You must not change the overall functionality of the function, and remove any libraries/modules that are
       imported in the function to accommodate the blackbox testing.
```

````
#### Input Parameters

- Check if the function takes input parameters. If not, you need to modify the description and the
      function to make it take input parameters.

#### Example

- Provide 1 or 2 examples of the input and output of the function.
  - ```bash
    >>> f_0("hello")
    "hello world"
    >>> f_0("I")
    "love you"
    ```
  - ```bash
    >>> f_0("image_1.jpg")
    <module 'matplotlib.pyplot'>
    >>> f_0("image_2.jpg")
    <module 'matplotlib.pyplot'>

    ```

### Step 3: Check Function Implementation

- Check if the function implementation is correct. If not, you need to repair the function implementation
      to make it correct.
- Check if the function uses any constants.
  - If yes and the description has not mentioned any of them, you need to either leave off the argument so
       it takes the default value, or modify the description to mention them.
    - For example, for the `plt.hist(counts, bins=np.arange(len(counts)+1)-0.5, rwidth=0.8)` in `f_0`, the
       description should mention the specific way to compute with `len(counts)+1)-0.5` and use `rwidth
      =0.8`.
- Check if the function has the return values. If not, you need to modify the function to make it return
      the values for the test cases to check.
- If the function requires to write or show some data, you shall either return the data or make the
      function take a path for file storage or a variable for data visualization. For example, if the
      function requires to show a plot, you must return the plot (via Axes).
- If the function requires checking some properties and uses `print()` to show the results, you need to
      modify this function to return these results. The revised function should amalgamate these properties
       with the preexisting return values, thereby facilitating more versatile utilization of the outputs
      in the rest of your code.
  - Consider this original function:

def check_properties(original_list):
    is_empty = not bool(original_list)
    print(f"Is the list empty? {is_empty}")

    length = len(original_list)
    print(f"Length of the list is {length}")

check_properties([1, 2, 3, 4])

This function checks two properties of a list: whether it's empty and its length. It then prints these
      results. However, these results can't be used elsewhere in your program.

Now, let's modify the function to return these results:

def check_properties(original_list):
    is_empty = not bool(original_list)
    length = len(original_list)

    return is_empty, length

list_empty, list_length = check_properties([1, 2, 3, 4])

print(f"Is the list empty? {list_empty}")
print(f"Length of the list is {list_length}")

In this modified version, the function returns the two properties instead of printing them. This allows
      you to capture the returned values in list_empty and list_length variables and use them elsewhere in
      your program.

- If you return any formats of values(e.g. `string`), make sure that you mention the format in the
      description. It is better for assessing the correctness of the function implementation.

### Step4: Run The Function and Write Blackbox Test Cases

- The function is contained in a file named `function.py`, and you are required to write a blackbox
      testing function named `run_tests()` that contains assertion-based blackbox test cases in `test.py`.
- If any of the following data types are used for manipulation, you need to manually design the data or
      utilize 3rd party libraries and websites (e.g. `Faker` and `unittest.mock`) to generate or mock the
      test data. You can use the "file://" protocol to access the local HTML files, and any url request
      APIs should work correctly with this protocol. (See https://chat.openai.com/share/84ba0dc9-067d-4eb0-
      a4d4-d8f4a77ff1a5)

  - html (webpage, page link)
  - csv
  - json
  - xml
  - sql
  - image
````

```
- You should test the possible attributes of that written data. For example, if you return a plot, you ↩
      need to test if the plot contains the correct title, x-axis, y-axis and data points.

- To formalize the test case writing, you need to write with the following function:

```python
def run_tests():
    suite = unittest.TestSuite()
    suite.addTest(unittest.makeSuite(TestCases))
    runner = unittest.TextTestRunner()
    runner.run(suite)

class TestCases(unittest.TestCase):
    def test_case_1(self):
        # Input 1: write your test case here.
        # provide a series of unit tests to test all attributes of the returned values
        pass

    def test_case_2(self):
        # Input 2: write your test case here.
        # Provide a series of unit tests to test all attributes of the returned values
        pass

    def test_case_3(self):
        # Input 3: write your test case here.
        # Provide a series of unit tests to test all attributes of the returned values
        pass

    # write more tests here
```

- Each blackbox test method should be tested with a unique input value and asserted for the return values.

Working

After reading the guideline, refine 'function.py' and write a single test program named 'run_tests()' in '↩
      test.py', which should contain at least *five* different input cases.

When testing with data, you need to generate the data by yourself or use 3rd party libraries and websites ↩
      (e.g. 'Faker' and 'unittest.mock') to generate or mock the test data. For example, you should design ↩
      the html webpage to meet the functionality and test scenarios. For any numeric values, you need to ↩
      represent them in the data.

Make the test data as complex as possible to mimic the real-world data. For example, if the function ↩
      requires reading a csv file, you need to design the csv file to meet the functionality and test ↩
      scenarios.

You can not remove any libraries or modules used in the original function. However, you can add new ↩
      libraries or modules to the function.

Make sure you have tested all return values and the possible attributes of the written data.

Keep testing the function until you are satisfied with the function implementation and the test cases.

If any tested properties are not mentioned in the description, you need to modify the description to ↩
      mention them.

As we will provide the function stub for programmers to implement by their own, please make sure there is ↩
      no ambiguity in the description and the function implementation. If there is any ambiguity, you need ↩
      to modify the description and the function implementation to make it clear and concise. Think about ↩
      the possible questions that programmers may ask and try to answer them in the description.

Execute to make sure 'function.py' will pass all blackbox test cases. Otherwise, you need to modify the ↩
      function implementation to make it pass all blackbox test cases.

Refine 'function.py' and write blackbox tests in 'test.py'

Note that 'Faker' library has already been installed in the environment and you can use it freely.

Download refined 'function.py', written 'test.py' and created 'test_data.zip' if there exists.
```

J.3 HUMAN CURATION GUIDELINES

```
# Annotation Guidelines

## Environment Setup

You are given a file named 'requirements.txt', please set up a Python environment with version 3.8.10. You↩
       can use Anaconda or Python Virtual Environment for this purpose.

Use the following command to install all required libraries:
```sh
pip install -U -r requirements.txt
```

Please note that this environment will be the same as the one used by OpenAI GPT-4 Advanced Data Analysis ↩
      (or Code Interpreter). Although it is expected that most APIs are stable, it is safer to use ↩
```

```
      consistent library versions. You are encouraged to use more libraries covered in the requirements.txt
       to enrich each sample.

## Annotation Goal

The goal is to:
Refine the function, including its docstrings in order to make the function more realistic and less
      ambiguous. This means when you see the function stub and docstring, you should be able to implement
      exactly the same functionality with the given function body. Add more library APIs if necessary;
 Write additional black box unit tests to ensure the functional correctness of the given function. You
      should consider as many corner cases as possible.

## Expected Annotation

You are given a Python script to execute and annotate.

We define the following terms:
 `Programming Problem` contains three parts:
[Standalone] `Import Statement`
The imported libraries, submodules or functions should be used in the following function implementation.
add missing libraries, submodules or functions if one is used but not imported.
`Problem Function`
`Function Signature` and its corresponding `Docstring`
The `Function Name` should be obfuscated in the format of `f_[NUMBER]` to ensure anonymity. [NUMBER]
      should be the one inside the file name.
The docstrings (example can be found at Google Python Style Guide) should contain:
A `Functionality Description`.
`Function Parameters` and their `Function Parameters Descriptions`.
2-3 `Running Examples` in Python Interpreter and their expected outputs.
`Solution`
The function implementation to fulfil the functionality described in the `Docstring`.
`Test Suite` contains three parts:
[Standalone] `Import Statement`
The imported libraries, submodules or functions should be used in the following tests.
[Standalone] `TestCases` class
The class should contain at least five distinct test cases.
The test cases should be aligned with the docstring description. Test cases should not assert any
      attributes which are not specifically mentioned.
The test cases should cover as many branches in the `Problem Function` as possible. In order to get the
      complete coverage, you should use the command `coverage run -m unittest f_[NUMBER]_[NAME].py &&
      coverage report -m`. Replace the `f_[NUMBER]_[NAME].py` with the file name you are testing with.
      Ignore the missing lines in the `Test Suite`.
`Programming Problem` should be able to pass all these test cases. This means the scripts should run
      successfully without any failed test cases when you run `python XXX.py` in the terminal.
[Standalone] `run_tests` function
The function should only contain the code helping to execute the test cases.

## Issues To Be Addressed

You may notice the existence of the following issues in the given Python script:
`Function Name` has not been obfuscated.
Replace the name with `f`.
`Docstring` is unclear, ambiguous,  impractical or not well aligned with `Solution`.
`Function Description` should describe a practical functionality.
You should either refine the `Function Description` or  `Solution`. Choose the one more feasible to be
      done.
Make sure at least 2 correct `Running Examples` are included.
`Solution` does not use all imported libraries or APIs.
Try to refine the `Programming Problem` so that the `Function Description` implies the corresponding API
      usage and such APIs are correctly invoked in `Solution`.
If (a) is difficult to complete, remove the unused import statements.
`Solution` uses APIs that are not included in `Import Statement`.
Add the corresponding import statements if these APIs are necessary to complete the functionality.
      Otherwise, refine `Function Description` so that the functionality will require such APIs.
`Solution` uses less than 2 libraries.
You should refine `Programming Problem` so that the functionality must require the API usage and invoke
      APIs from at least 2 distinct libraries. You can use ChatGPT (GPT-4) in Advanced Data Analysis for
      inspiration.
`Solution` uses APIs in `random` or the random functionality.
Initialize the random seed for each `TestCases` to control the behavior.
`Solution` contains dummy code.
Based on your understanding, replace the dummy code with the actual implementation of each part.
`Solution` contains the display functionality, such as `print()` and `matplotlib.pyplt.show()`.
If the function requires you to write or show some data, you shall either return the data or make the
      function take a path for file storage or a variable for data visualization. For example, if the
      function requires you to show a plot, you must return the plot (via Axes). If there is a specific
      attribute inside the object, you should mention it in the `Docstring` and test it inside `TestCases`.
       For example, the plot contains the specific title or label names. You should make sure that these
      attributes are either stated in `Docstring` or implied by `Docstring`.
If the function requires checking some properties and uses `print()` to show the results, you need to
      modify this function to return these results. The revised function should amalgamate these properties
       with the preexisting return values, thereby facilitating more versatile utilization of the outputs
      in the rest of your code.
Refer to Step3 in the guidelines of the previous stage.

Global constants before `Problem Function`.
If the global constants are used as sample inputs in the `Solution`, remove them and write your own test
      input in `TestCases`.
If the global constants are unused, remove them directly.
 Test cases inside `TestCases` only check the range of returned results or fail to test in a specific way.
```

```
It assumes that you now have full control of `Programming Problem`. Write the test cases to validate if
      the returned results are equal to certain values. For the returned objects, validate if the
      attributes are equal to certain values.
Test cases do not use any deterministic values as expected outputs.
Come up with the expected outputs after testing.
`TestCases` uses libraries or APIs that are not included in `Import Statement`.
Add the corresponding import statements if these APIs are necessary to complete the testing. Otherwise,
      remove such APIs.
`TestCases` contains test cases that do not work for `Solution`.
Repair these test cases or replace them with better cases.
 `TestCases` does not test all attributes of the returned object, where these attributes are implied by `
      Function Description` or completed by `Solution`.
Add lines of code to test these attributes.
If these attributes are not mentioned or implied by `Function Description`, try to describe them in `
      Function Description`.
`TestCases` does not test the files that result in `Solution`.
Some files are created during the execution of `Programming Problem`. Add necessary lines of code to test
      the attributes of these files in each test case.
 `TestCases` is wrapped in `run_tests`.
Separate these two.
 Test cases in `TestCases` are duplicated or used to test the same behavior.
Remove them if there is a huge overlap. Replace them with more complex test cases. Make sure that at least
       five test cases are included.
 Test data used in `TestCases` is missing.
You need to manually design the data or utilize 3rd party libraries and websites (e.g. `Faker` and `
      unittest.mock`) to generate or mock the test data. Refer to Step4 in the guidelines of previous stage
      .

 Lack of return value.
Functions should have clear return values to indicate the result, and these should be tested in the `
      TestCases`.
 Lack of corner cases.
Corner cases should be considered in the function or in the test cases.
 Lack of error handling:
You should add necessary checks for null inputs, incorrect data types, or values out of expected ranges to
       deal with incorrect input format.

## Further Explanation of Each Issue

1. `Function Name` has not been obfuscated:
     - The given function should have a generic name such as `f` to ensure anonymity. This prevents the
       user from inferring the function's purpose based on its name.
     - Example:
       Before: `def calculate_average(nums):`
       After: `def f(nums):`

2. `Docstring` is unclear, ambiguous, impractical or not well aligned with `Solution`:
     - The function's docstring should provide a clear and concise description of its purpose, expected
       inputs, outputs, and examples of usage. If the description is vague or doesn't match the function's
       behavior, it can lead to confusion.
     - Example:
       Before: `"""Calculates something."""`
       After: `"""Calculates the average of a list of numbers."""`

3. `Solution` does not use all imported libraries or APIs:
     - If libraries are imported but not used in the function, it indicates redundant code or a mismatch
       between the problem description and the solution.
     - Example:
       Before: `import math` (but no usage of `math` in the function)
       After: Remove `import math` or ensure it's used in the function.

4. `Solution` uses APIs that are not included in `Import Statement`:
     - All external libraries or functions used in the solution should be imported at the beginning of the
       script to ensure the code runs without errors.
     - Example:
       If using `sqrt` from `math` library in the function, ensure `from math import sqrt` is present at
       the beginning.

5. `Solution` does not use any library APIs:
     - The problem should be designed in a way that requires the usage of library APIs to solve it,
       ensuring the challenge of integrating external tools.
     - Example:
       If the problem is to calculate the square root, the solution should leverage the `math.sqrt`
       function.

6. `Solution` uses APIs in `random`, but does not pass a random seed to `Function Parameters`:
     - When using random functionalities, for reproducibility, it's good practice to allow the user to set
       a seed.
     - Example:
       Before: `random.randint(1,10)`
       After: `random.seed(seed); random.randint(1,10)`

7. `Solution` contains dummy code:
     - Placeholder or dummy code should be replaced with actual implementation to ensure the function works
       as expected.
     - Example:
       Before: `# TODO: Implement this`
       After: Actual implementation of the required logic.

8. Unused global constants before `Problem Function`:
     - Any constants or variables that are not used in the solution should be removed to clean up the code.
```

```
9. `TestCases` uses libraries or APIs that are not included in `Import Statement`:
    - Similar to the solution, all external libraries or functions used in the test cases should be imported.

10. `TestCases` contains test cases that do not work for `Solution`:
    - All test cases should be aligned with the function's behavior to ensure they test the function correctly.

11. `TestCases` does not test all attributes of the returned object:
    - If the function returns an object with multiple attributes or methods, the test cases should validate all of them to ensure complete coverage. For example, when plotting data on a graph, you might get an `AxesSubplot` object in return. This object has various attributes, like the title, x-label, y-label, and the data points themselves. You should test all of these attributes if they are required in the functionality.

12. `TestCases` does not test the files that result in `Solution`:
    - If the function creates or modifies files, the test cases should validate these files to ensure the function works as expected.

13. `TestCases` is wrapped in `run_tests`:
    - The test cases and the function to run them should be separated for clarity.

14. Test cases in `TestCases` are duplicated or used to test the same behavior:
    - Redundant test cases should be removed to keep the test suite concise and focused.

15. Test data used in `TestCases` is missing:
    - All required data for testing should be provided or generated to ensure the test cases can run without issues.
```

K Evaluation Setup

K.1 Inference

We perform all the model inference on A100 GPUs, except for the closed ones. For the closed models, we rely on their official APIs provided in the documents.

K.2 Execution

We conduct the execution mainly on the Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz, composed of 2 sockets, with 18 cores per socket.

K.3 Prompt Template

Please provide a self-contained Python script that solves the following problem in a markdown code block:
`{prompt}`

Figure 14: Prompt template for models supported by vLLM (Kwon et al., 2023).

Please generate self-contained code to complete the following problem:
`{prompt}`

Figure 15: Prompt template for OpenAI/DeepSeek APIs.

Please generate self-contained code to solve the following problem in a Python markdown block:
`{prompt}`

Figure 16: Prompt template for Mistral model APIs.

Please generate self-contained code to complete the following problem wrapped in a Python markdown block:
`{prompt}`

Figure 17: Prompt template for Anthropic model APIs.

L Detailed Benchmarking Results and Analysis

L.1 Detailed Results

Table 6: Evaluating LLMs on `BigCodeBench-Complete`. Due to the training flaws in StarCoder2 and Granite-Code, we additionally strip the trailing newlines for model inference.

Model	**Size / Checkpoint**	**Greedy Decoding** Original	**Greedy Decoding** Calibrated	**Random Sampling** $Pass@1$	**Random Sampling** $Pass@5$
Instruction-tuned LLMs					
GPT-4o (OpenAI, 2024a)	2024-05-13	0.602	0.611	0.557	0.711
GPT-4-Turbo (OpenAI, 2024b)	2024-04-09	0.582	0.582	0.563	0.699
GPT-4 (Achiam et al., 2023)	0613	0.484	0.572	0.407	0.682
GPT-3.5-Turbo (OpenAI, 2023)	0125	0.506	0.506	0.504	0.657
Claude-3 (Anthropic, 2024)	Opus	0.568	0.574	—	—
	Sonnet	0.534	0.538	—	—
	Haiku	0.486	0.501	—	—
Mistral (AI, 2024a)	Large	0.379	0.383	0.365	0.539
	Small	0.390	0.413	0.398	0.601
DeepSeek-Chat (DeepSeek-AI, 2024)	V2	0.494	0.494	0.482	0.596
DeepSeek-Coder-instruct (Guo et al., 2024)	33B	0.511	0.511	0.491	0.687
	6.7B	0.432	0.438	0.426	0.624
	1.3B	0.293	0.296	0.270	0.468
CodeLlama-instruct (Roziere et al., 2023)	70B	0.489	0.496	0.429	0.681
	34B	0.351	0.356	0.266	0.518
	13B	0.297	0.317	0.247	0.470
	7B	0.254	0.257	0.193	0.403
Llama3-instruct (AI@Meta, 2024)	70B	0.539	0.545	0.522	0.650
	8B	0.368	0.369	0.338	0.562
Qwen2-Instruct (Bai et al., 2023)	72B	0.539	0.540	0.489	0.682
	57B-A14B	0.465	0.468	0.399	0.648
	7B	0.407	0.421	0.329	0.577
CodeQwen1.5-Chat (Bai et al., 2023)	7B	0.437	0.443	0.408	0.632
Qwen1.5-Chat (Bai et al., 2023)	110B	0.432	0.444	0.425	0.590
	72B	0.398	0.403	0.373	0.569
	32B	0.410	0.420	0.375	0.557
Yi-1.5-Chat (Young et al., 2024)	34B	0.433	0.428	0.411	0.622
	9B	0.422	0.424	0.379	0.601
	6B	0.338	0.339	0.290	0.496
Mixtral-instruct (AI, 2024c)	8x22B	0.496	0.502	0.458	0.677
Magicoder-S-DS (Wei et al., 2023)	6.7B	0.475	0.476	0.424	0.643
CodeGemma-instruct (Team et al., 2024)	7B	0.383	0.393	0.334	0.568
StarCoder2-Instruct (BigCode, 2024)	15B	0.437	0.451	0.404	0.610
Granite-Code-Instruct (Mishra et al., 2024)	34B	0.438	0.444	0.399	0.613
	20B	0.421	0.421	0.375	0.600
	8B	0.392	0.397	0.367	0.581
	3B	0.315	0.315	0.250	0.466
Base LLMs					
CodeLlama (Roziere et al., 2023)	70B	0.443	—	0.364	0.639
	34B	0.371	—	0.297	0.570
	13B	0.320	—	0.243	0.527
	7B	0.287	—	0.207	0.457
Llama3-base (AI@Meta, 2024)	70B	0.433	—	0.375	0.625
	8B	0.288	—	0.223	0.466
DeepSeek-Coder-base (Guo et al., 2024)	33B	0.466	—	0.401	0.661
	6.7B	0.418	—	0.343	0.599
	1.3B	0.223	—	0.174	0.412
CodeQwen1.5 (Bai et al., 2023)	7B	0.456	—	0.411	0.650
Yi-1.5 (Young et al., 2024)	34B	0.400	—	0.302	0.575
	9B	0.355	—	0.295	0.563
	6B	0.274	—	0.201	0.435
Mixtral-base (AI, 2024b)	8x22B	0.455	—	0.355	0.633
CodeGemma (Team et al., 2024)	7B	0.373	—	0.295	0.557
	2B	0.239	—	0.153	0.375
StarCoder2 (Lozhkov et al., 2024)	15B	0.383	—	0.332	0.609
	7B	0.285	—	0.232	0.514
	3B	0.214	—	0.178	0.416
Granite-Code (Mishra et al., 2024)	34B	0.385	—	0.333	0.582
	20B	0.258	—	0.286	0.552
	8B	0.356	—	0.287	0.536
	3B	0.202	—	0.177	0.406

Table 7: Evaluating instruction-tuned LLMs on `BigCodeBench-Instruct`. We report the results of greedy decoding, paired with calibrated results. To understand the performance difference between `BigCodeBench-Complete` and `BigCodeBench-Instruct`, we compute $\Delta(NL2C - C2C)$ for each model. We do not include the results of Granite-Code-Instruct 8B & 3B as they constantly have empty outputs.

Model	**Size / Checkpoint**	**Greedy Decoding**		$\Delta(NL2C - C2C)$	
		Original	Calibrated	Original	Calibrated
GPT-4o	2024-05-13	0.499	0.511	-0.103	-0.100
GPT-4-Turbo	2024-04-09	0.480	0.482	-0.102	-0.100
GPT-4	0613	0.459	0.460	-0.025	-0.112
GPT-3.5-Turbo	0125	0.382	0.391	-0.124	-0.115
Claude-3	Opus	0.452	0.455	-0.116	-0.119
	Sonnet	0.425	0.427	-0.109	-0.111
	Haiku	0.392	0.394	-0.094	-0.107
DeepSeek-Chat	V2	0.404	0.404	-0.090	-0.090
DeepSeek-Coder-instruct	33B	0.418	0.420	-0.093	-0.091
	6.7B	0.353	0.355	-0.079	-0.083
	1.3B	0.227	0.228	-0.066	-0.068
CodeLlama-instruct	70B	0.405	0.407	-0.084	-0.089
	34B	0.289	0.290	-0.062	-0.066
	13B	0.282	0.285	-0.015	-0.032
	7B	0.218	0.219	-0.036	-0.038
Llama3-instruct	70B	0.432	0.436	0.107	-0.109
	8B	0.317	0.319	-0.051	-0.050
Qwen2-Instruct	72B	0.382	0.385	-0.157	-0.155
	57B-A14B	0.357	0.361	-0.108	-0.060
	7B	0.280	0.291	-0.127	-0.130
CodeQwen1.5-Chat	7B	0.393	0.396	-0.044	-0.047
Qwen1.5-Chat	110B	0.336	0.350	-0.096	-0.094
	72B	0.326	0.332	-0.072	-0.071
	32B	0.318	0.323	-0.092	-0.097
Yi-1.5-Chat	34B	0.335	0.339	-0.098	-0.089
	9B	0.343	0.345	-0.079	-0.079
	6B	0.255	0.256	-0.083	-0.083
Mistral	Large	0.296	0.300	-0.083	-0.083
	Small	0.318	0.321	-0.072	-0.092
Mixtral	8x22B	0.404	0.406	-0.092	-0.096
Magicoder-S-DS	6.7B	0.360	0.362	-0.115	-0.114
CodeGemma-instruct	7B	0.321	0.323	-0.062	-0.070
StarCoder2-Instruct	15B	0.367	0.376	-0.070	-0.075
Granite-Code-Instruct	34B	0.358	0.361	-0.080	-0.083
	20B	0.337	0.340	-0.084	-0.081

L.2 Further Analysis

Bigger models yield better programming performance The performance on `BigCodeBench` displays signs of scaling laws, where the models with more parameters generally solve more programming tasks. While the scaling laws hold for most instruction-tuned and base LLMs, Mistral-Large appears less capable than Mistral-Small on both `BigCodeBench-Complete` and `BigCodeBench-Instruct`, implying that Mistral-Large may be under-fitting. Our findings are consistent with (Yan et al., 2024).

Closed LLMs perform better than open LLMs We notice that most strong LLMs on `BigCodeBench` are non-permissive, led by the models from OpenAI and Anthropic. Among the open LLMs, the best model, Llama3-instruct-70B, slightly outperforms Claude-3-Sonnet and ranks 5th on both `BigCodeBench-Complete` and `BigCodeBench-Instruct`. The most permissive LLMs are relatively small and, hence, remain in a gap from the non-permissive ones.

Jack of all trades, master of most Figure 7 shows the top 5 for instruction-tuned ranked on `BigCodeBench-Complete`. Best overall base LLMs like Dpsk-Coder-Base-6.7B excel in most domains but still fall short in certain ones. We suggest that the domain-specific specialty will likely result from the training data.

LLMs are better at computation, cryptography, and general domains We further investigate the performance among different domains and find that models tend to be more capable of the tools in computation, cryptography, and general domains. On the other hand, models frequently fail when the tasks involve network tools. Therefore, we encourage the development of models specializing in such low-performing domains. A few examples of failures in the network domain can be found in Appendix M Examples 6, 7, and 8.

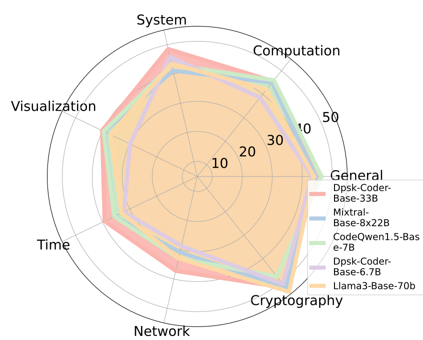


Figure 18: Top 5 base LLMs

L.3 Prompting Techniques

Table 8: Calibrated Pass@1 comparison of plain prompting and zero-shot CoT prompting on both `BigCodeBench` and `BigCodeBench-Hard`.

Split-Subset	GPT-4o-2024-05-13 (%)	Gemini-1.5-Pro-API-0514 (%)
Full-Complete	61.1 → 59.4	57.5 → 55.9
Full-Instruct	51.1 → 49.5	43.8 → 44.2 (↑)
Hard-Complete	29.1 → 34.5 (↑)	31.1 → 27.7
Hard-Instruct	25.0 → 23.0	19.6 → 18.2

We provide some preliminary studies on the effectiveness of the zero-shot chain-of-thought (CoT) (Kojima et al., 2022) on GPT-4o and Gemini-1.5-Pro by appending "\nLet's think step by step." to the end of the original prompt. From Table 8, we observe that there is no significant advantage to using zero-shot CoT for performing `BigCodeBench` tasks, indicating that the prompting technique is not sufficiently reliable for complex compositional reasoning. Therefore, we conclude that prompt design is very tricky and should not be recommended for the official evaluation setup. For our prompt design, we followed the same approach used in EvalPlus for consistency.

M QUALITATIVE STUDIES

Our benchmark exposes instances where even the strongest models struggle to use APIs, which may not be exposed by simple benchmarks like HumanEval. Here, we show some examples with GPT-4o.

Example 1: In the following example, the model fails to use `itertools.cycle` correctly, producing an infinite loop.

```
from collections import Counter
import random
from itertools import cycle

# Constants
ELEMENTS = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']

def task_func(l):
    """
    Create a counter from a list "l" and move the first 3 elements to the end of the list.

    Parameters:
    - l (list): A list of elements that the function will process.

    Returns:
    - counter (collections.Counter): A frequency counter that maps elements from the input list to their
      frequencies in the first 30 elements of the cycled, shuffled list.

    Requirements:
    - collections
    - random
    - itertools

    Example:
    >>> random.seed(42)
    >>> task_func(ELEMENTS)
    Counter({'I': 3, 'F': 3, 'G': 3, 'J': 3, 'E': 3, 'A': 3, 'B': 3, 'H': 3, 'D': 3, 'C': 3})
    """
    # Move the first 3 elements to the end of the list
    l = l[3:] + l[:3]

    # Create a cycled, shuffled list
    cycled_list = list(cycle(l))
    ...
```

Example 2: In this example, the model uses the semantics of `instanceof` incorrectly. Here, `isinstance(j, (int, float))` does not actually check if j is an int or float, but rather the type `(int, float)`.

```
import pandas as pd
import seaborn as sns

def task_func(data):
    """
    Analyze a dataset by calculating the average of values across each row and visualizing the correlation
       matrix as a
    heatmap.

    Parameters:
    data (numpy.array): 2D array where each row represents a record and each column represents a feature

    Returns:
    tuple: A tuple containing:
        - DataFrame: A pandas DataFrame enhanced with an 'Average' column that represents the mean across
      each row.
        - Axes: The matplotlib Axes object showing the heatmap of the correlations.

    Requirements:
    - pandas
    - numpy
    - seaborn

    Raises:
    ValueError: If the input data is not a 2D array or if it contains non-numeric data.

    Example:
    >>> data = np.array([[1, 2, 3, 4, 5, 6, 7, 8], [8, 7, 6, 5, 4, 3, 2, 1]])
    >>> df, ax = task_func(data)
    >>> print(df['Average'].to_string(index=False))
    4.5
    4.5
    """
    # Check if data is a 2D array
    if not isinstance(data, (list, np.ndarray)) or not all(isinstance(i, (list, np.ndarray)) for i in data
      ):
        raise ValueError("Input data is not a 2D array")
    # Check if data contains only numeric values
    if not all(all(isinstance(j, (int, float)) for j in i) for i in data):
```

```
        raise ValueError("Input data contains non-numeric data")
    ...
```

Example 3: Here, the model mishandles the case where the directory does not exist with incorrect use of `os.path.dirname`.

```
import pickle
import os

def task_func(filename, data):
    """
    Serialize an object using pickle and overwrite the specified file with this serialized data.
    Before writing, checks if the directory exists, creating it if necessary.

    Parameters:
    - filename (str): The path of the file to be overwritten with serialized data.
    - data (object): The object to serialize and write to the file.

    Returns:
    - bool: True if the operation is successful, False otherwise.

    Requirements:
    - pickle
    - os

    Example:
    >>> result = task_func('data.pkl', {'key': 'value'})
    >>> print(result)
    True
    """
    try:
        # Check if the directory exists
        dir_name = os.path.dirname(filename)
        if not os.path.exists(dir_name):
            # Create directory if it does not exist
            os.makedirs(dir_name)

        # Open the file in write mode
        with open(filename, 'wb') as file:
            # Serialize the object and write it to the file
            pickle.dump(data, file)

        return True
    except Exception as e:
        print(f"An error occurred: {e}")
        return False
```

Example 4: In this example, the model misunderstands how to use the `itertools.zip_longest` method. The arguments must be iterable, but the written function feeds single values.

```
import numpy as np
import itertools

def task_func(data_list):
    """
    Unzips a list of tuples and calculates the mean of the numeric values for
    each position.

    The function accepts a list of tuples, where each tuple consists of
    alphanumeric values. It unzips the tuples, and calculates the mean of
    numeric values at each position using numpy, where non numeric values are
    ignores. If all values at a position are non numeric, the mean at this
    position is set to be np.nan.
    If the provided tuples have different number of entries, missing values are
    treated as zeros.

    Parameters:
    - data_list (list of tuples): The data to process, structured as a list of tuples. Each tuple can
      contain alphanumeric values.

    Returns:
    - list: A list of mean values for each numeric position across the tuples. Non-numeric positions are
      ignored.
            An empty list is returned if the input list (data_list) is empty.

    Requirements:
    - numpy
    - itertools

    Example:
    >>> task_func([('a', 1, 2), ('b', 2, 3), ('c', 3, 4), ('d', 4, 5), ('e', 5, 6)])
    [nan, 3.0, 4.0]
    >>> task_func([(1, 'a', 2), ('a', 3, 5), ('c', 1, -2)])
    [1.0, 2.0, 1.6666666666666667]
    """
    if not data_list:
```

```
        return []

    max_len = max(len(t) for t in data_list)
    data_list = [list(itertools.zip_longest(*t, fillvalue=0)) for t in data_list]
    ...
```

Example 5: In the following example, the model outputs code that has a different encoding for `password_hash` and `expected_password_hash` so that even if the passwords are the same, the response will indicate a failed one.

```
def task_func(data):
    """
    This method is designed to handle the authentication process in a web application context.
    It expects input in the form of a dictionary with 'username' and 'password' keys. The password
    is expected to be a base64-encoded SHA-256 hash. The method decodes and authenticates these
      credentials
    against predefined values (for demonstration purposes, it checks if the username is 'admin' and the
    password hash matches the hash of 'password'). Based on the authentication result, it returns an
      appropriate
    HTTP response.

    Parameters:
    data (dict): A dictionary with 'username' and 'password' keys.

    Returns:
    django.http.HttpResponse: An HttpResponse indicating the login result.
                              HttpResponseBadRequest if the data is invalid.

    Raises:
    KeyError, UnicodeDecodeError, binascii.Error, ValueError if the input dictionary is invalid.

    Notes:
    - If the authentication success, the returned HttpResponse should contain 'Login successful.' with
      status 400.
    - If the authentication fails, the returned HttpResponse should contain 'Login failed.' with status
      401.
    - If the input data is invalid (i.e., password is a non-base64, missing keys), the function return
      HttpResponseBadRequest and it contains 'Bad Request.'

    Examples:
    >>> from django.conf import settings
    >>> if not settings.configured:
    ...     settings.configure()
    >>> data = {'username': 'admin', 'password': base64.b64encode(hashlib.sha256('password'.encode()).
      digest()).decode()}
    >>> response = task_func(data)
    >>> response.status_code == 200 and 'Login successful.' in response.content.decode()
    False

    >>> data = {'username': 'admin', 'password': base64.b64encode(hashlib.sha256('wrongpassword'.encode())
      .digest()).decode()}
    >>> response = task_func(data)
    >>> response.status_code == 401 and 'Login failed.' in response.content.decode()
    False

    Requirements:
    - django.http
    - django.conf
    - base64
    - hashlib
    - binascii
    """
    try:
        # Check if the required keys are present
        if 'username' not in data or 'password' not in data:
            raise KeyError('Missing keys in input data')

        username = data['username']
        password_base64 = data['password']

        # Decode the base64-encoded password
        try:
            password_hash = base64.b64decode(password_base64)
        except (binascii.Error, ValueError) as e:
            raise ValueError('Invalid base64 encoding')

        # Predefined credentials for demonstration
        expected_username = 'admin'
        expected_password_hash = hashlib.sha256('password'.encode()).digest()

        # Authenticate the credentials
        if username == expected_username and password_hash == expected_password_hash:
            return HttpResponse('Login successful.', status=200)
        else:
            return HttpResponse('Login failed.', status=401)

    except (KeyError, UnicodeDecodeError, binascii.Error, ValueError) as e:
        return HttpResponseBadRequest('Bad Request.')
```

Example 6: This example showcases a failure to understand the nuances of the `wikipedia` library. When `page_title` is not found in Wikipedia, verbatim, the `wikipedia.page` method returns a `wikipedia.exceptions.DisambiguationError` rather than a `wikipedia.exceptions.PageError`.

```
import wikipedia
from wordcloud import WordCloud
import matplotlib.pyplot as plt

def task_func(page_title):
    """
    Create a word cloud from the text of a Wikipedia page.

    Parameters:
    page_title (str): The title of the Wikipedia page.

    Returns:
    matplotlib.axes.Axes: The Axes object of the plotted data. Is None if there is no wikipedia page with ↩
      the title given as input.

    Requirements:
    - wikipedia
    - wordcloud.WordCloud
    - matplotlib.pyplot

    Example:
    >>> ax = task_func('Python (programming language)')
    """
    try:
        # Fetch the Wikipedia page content
        page_content = wikipedia.page(page_title).content

        # Generate the word cloud
        wordcloud = WordCloud(width=800, height=400, background_color='white').generate(page_content)

        # Plot the word cloud
        fig, ax = plt.subplots(figsize=(10, 5))
        ax.imshow(wordcloud, interpolation='bilinear')
        ax.axis('off')  # Hide the axes

        # Show the plot
        plt.show()

        return ax
    except wikipedia.exceptions.PageError:
        # Return None if the page does not exist
        return None
```

Example 7: In this *Network* example, the error actually originates from the regex. For the first example, instead of `www.google.com`, the regex actually extracts `www.google.com,`, which leads to errors down the line.

```
import re
import socket
import urllib.parse
def task_func(myString):
    """
    Extracts all URLs from a given string, analyzes each URL to extract the domain, and retrieves the IP ↩
      address of each domain.

    Parameters:
    myString (str): The string from which URLs are extracted. The string should contain valid URLs ↩
      starting with http or https.

    Returns:
    dict: A dictionary with domains as keys and their respective IP addresses (IPv4) as values. If a ↩
      domain cannot be resolved, the IP address will be None.

    Requirements:
    - re
    - urllib.parse
    - socket

    Raises:
    socket.gaierror if the domain cannot be resolved

    Example:
    >>> task_func("Check these links: http://www.google.com, https://www.python.org")
    {'www.google.com': '172.217.12.142', 'www.python.org': '151.101.193.223'}
    """

    # Regular expression to find URLs
    url_pattern = re.compile(r'https?://[^\s/$.?#].[^\s]*')
    urls = url_pattern.findall(myString)

    # Dictionary to store domain and IP address
    domain_ip_map = {}
```

```python
    for url in urls:
        # Parse the URL to extract the domain
        parsed_url = urllib.parse.urlparse(url)
        domain = parsed_url.netloc

        # Get the IP address of the domain
        try:
            ip_address = socket.gethostbyname(domain)
        except socket.gaierror:
            ip_address = None

        # Add the domain and IP address to the dictionary
        domain_ip_map[domain] = ip_address

    return domain_ip_map
```

Example 8: In this example, the model misunderstands the host naming structures. The issue is in the line `response = requests.get(f"https://ipinfo.io/ip_address/json")`, where the `/json` should not be present. This leads to a `hostname not found` error.

```python
import socket
import requests
def task_func(host):
    """
    This function resolves the IP address of the given host and then uses the IP address
    to fetch geolocation information from the ipinfo.io API. The function is robust against
    various common errors, such as invalid hostnames, network issues, or problems with the
    geolocation service.

    Parameters:
    host (str): The hostname to be resolved.

    Returns:
    dict: A dictionary containing the IP address and geolocation information if successful.

    Raises:
    ValueError: If 'host' is None or an empty string.
    ConnectionError: If there is a problem connecting to the geolocation service.

    Example:
    >>> result = task_func('google.com')
    >>> 'ip_address' in result and 'geolocation' in result
    True
    >>> task_func('')
    Traceback (most recent call last):
       ...
    ValueError: Host must be a non-empty string.

    Requirements:
    - socket
    - requests
    """
    if not host:
        raise ValueError("Host must be a non-empty string.")

    try:
        ip_address = socket.gethostbyname(host)
    except socket.gaierror:
        raise ValueError("Invalid hostname.")

    try:
        response = requests.get(f"https://ipinfo.io/{ip_address}/json")
        response.raise_for_status()
        geolocation = response.json()
    except requests.RequestException as e:
        raise ConnectionError(f"Error connecting to the geolocation service: {e}")

    return {
        "ip_address": ip_address,
        "geolocation": geolocation
    }
```

Example 9: This example shows an example where there is a simple solution using `flatten`, but the generated solution does not use this function and therefore gets the function incorrect.

```python
def task_func(dimension, seed=42):
    """
    Create a 2D numeric array (matrix) of a given dimension with random integers between 1 and 100,
    and a flat list of all elements in the matrix.

    Parameters:
    - dimension (int): The dimension of the square matrix to be created. It must be a positive integer.

    Returns:
    tuple: A tuple containing:
        - A 2D numpy array of the given dimension with random integers between 1 and 100.
        - A flat list of all elements in the matrix.
```

```
    Requirements:
    - numpy
    - itertools

    Example:
    >>> matrix, flat_list = task_func(3)
    >>> print(matrix)
    [[52 93 15]
     [72 61 21]
     [83 87 75]]
    >>> print(flat_list)
    [52, 93, 15, 72, 61, 21, 83, 87, 75]
    """
    # Set the random seed for reproducibility
    np.random.seed(seed)

    # Generate a 2D numpy array with random integers between 1 and 100
    matrix = np.random.randint(1, 101, size=(dimension, dimension))

    # Flatten the matrix to a list
    flat_list = list(itertools.chain.from_iterable(matrix))

    return matrix, flat_list

# Ground Truth
def task_func(dimension, seed=42):
    np.random.seed(seed)  # Ensure reproducible results
    if dimension <= 0:
        raise ValueError("The dimension must be a positive integer")
    matrix = np.random.randint(1, 101, size=(dimension, dimension))
    flat_list = matrix.flatten().tolist()
    return matrix, flat_list
```

Example 10: This is another example where the model chooses to use the wrong function. Here, it uses `np.mean` instead of `np.nanmean` (which the ground truth solution uses). This leads to the failure of a test with incomplete data.

```
import pandas as pd
import numpy as np
import itertools
def task_func(data_list=[('a', 1, 2.1), ('b', 2, 3.2), ('c', 3, 4.3), ('d', 4, 5.4), ('e', 5, 6.5)]):
    """
    Calculate the mean of numerical values in each position across tuples in a list.
    Non-numeric values are ignored, and means are computed only from available data.
    That means that missing data in some of the tuples is simply ignored.

    A DataFrame with one columns named 'Mean Value' which contains the mean values for all tuple positions
    .
    The index is according to this scheme: 'Position i' where i is the current position.
    If an empty list is passed, then an empty DataFrame is returned.

    Parameters:
    data_list (list of tuples): A list containing tuples of mixed data types (string, int, float, etc.).
        Defaults to [('a', 1, 2.1), ('b', 2, 3.2), ('c', 3, 4.3), ('d', 4, 5.4), ('e', 5, 6.5)]

    Returns:
    DataFrame: A pandas DataFrame with the mean values of the numerical data at each position.

    Requirements:
    - pandas
    - numpy
    - itertools

    Example:
    >>> df = task_func()
    >>> print(df)
                Mean Value
    Position 0         NaN
    Position 1         3.0
    Position 2         4.3

    >>> data = [('a', '1', 2.1), ('b', 21, 'c'), (12, 3, 4.3), (['d'], 4, 5.4), ('e', 5, 6.5)]
    >>> df = task_func(data)
    >>> print(df)
                Mean Value
    Position 0         NaN
    Position 1         8.25
    Position 2         4.3
    """
    if not data_list:
        return pd.DataFrame(columns=['Mean Value'])

    # Transpose the list of tuples to get columns
    transposed = list(itertools.zip_longest(*data_list, fillvalue=np.nan))

    means = []
    for col in transposed:
        # Filter out non-numeric values
        numeric_values = [x for x in col if isinstance(x, (int, float))]
```

```python
        if numeric_values:
            mean_value = np.mean(numeric_values)
        else:
            mean_value = np.nan
        means.append(mean_value)

    # Create the DataFrame
    index = [f'Position {i}' for i in range(len(means))]
    df = pd.DataFrame(means, index=index, columns=['Mean Value'])

    return df

# Ground Truth Solution
def task_func(data_list=[('a', 1, 2.1), ('b', 2, 3.2), ('c', 3, 4.3), ('d', 4, 5.4), ('e', 5, 6.5)]):
    unzipped_data = list(itertools.zip_longest(*data_list, fillvalue=np.nan))
    mean_values = []
    for column in unzipped_data[:]:
        numeric_values = [val for val in column if isinstance(val, (int, float))]
        if numeric_values:
            mean_values.append(np.nanmean(numeric_values))
        else:
            mean_values.append(np.nan)
    df = pd.DataFrame(mean_values, columns=['Mean Value'],
                      index=['Position {}'.format(i) for i in range(len(mean_values))])
    return df
```

N UNIT TEST DESIGN

HumanEval Test: The HumanEval tests only consider input-output assertions, which only work for simple programs without configuration and environment setup. We use several tests below as an example.

```python
METADATA = { 'author': 'jt', 'dataset': 'test' }

def check(candidate):
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False
```

`BigCodeBench` Unit Test: We demonstrate the design of `BigCodeBench` unit tests as follows, where we mock various scenarios for the network connection and validate the program behaviours. Compared to the ones in HumanEval and other benchmarks like APPS using input-output assertions, our unit tests require great human effort to design and cover various settings.

```python
# Requirements SetUp
import unittest
from unittest.mock import patch
import http.client
import ssl
import socket

# Start the test
class TestCases(unittest.TestCase):

    # Mock the successful connection and assess the response content
    @patch('http.client.HTTPSConnection')
    def test_response_content(self, mock_conn):
        """ Test the content of the response. """
        mock_conn.return_value.getresponse.return_value.read.return_value = b'Expected Content'
        result = task_func('www.example.com', 443, '/content/path')
        self.assertEqual(result, 'Expected Content')

    # Mock the failed connection and assess the error handling
    @patch('socket.create_connection')
    @patch('http.client.HTTPSConnection')
    def test_ssl_handshake_error_handling(self, mock_conn, mock_socket):
        """ Test handling of SSL handshake errors. """
        mock_socket.side_effect = ssl.SSLError('SSL handshake failed')
        with self.assertRaises(ssl.SSLError):
            task_func('badssl.com', 443, '/test/path')

    # More test cases...
```

We further illustrate how the unit test is constructed for the programming tasks related to data visualization. In the following example, the function `task_func` returns both a DataFrame and a bar chart Axes object for validation. The DataFrame's `Category` and `Value` columns are tested to ensure they correctly represent the input data. The Axes object is validated for its title (`"Category vs Value"`) and bar properties, including heights and x-axis labels, using a helper method, `is_bar`. The test cases cover diverse inputs with varying data sizes and values, ensuring the function reliably handles both data transformation and visualization requirements. We note that the test case design for the visualization tasks is similar to DS-1000 (Lai et al., 2023) but with more detailed validation.

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns


def task_func(list_of_pairs):
    """
    Create a Pandas DataFrame from a list of pairs and visualize the data using a bar chart.
    - The title of the barplot should be set to 'Category vs Value'`.

    Parameters:
    list_of_pairs (list of tuple): Each tuple contains:
        - str: Category name.
        - int: Associated value.

    Returns:
    tuple:
        - DataFrame: A pandas DataFrame with columns 'Category' and 'Value'.
        - Axes: A matplotlib Axes displaying a bar chart of categories vs. values.

    Requirements:
    - pandas
    - matplotlib.pyplot
    - seaborn

    Example:
    >>> list_of_pairs = [('Fruits', 5), ('Vegetables', 9)]
    >>> df, ax = task_func(list_of_pairs)
    >>> print(df)
         Category  Value
    0      Fruits      5
    1  Vegetables      9
    """
    pass


import unittest
class TestCases(unittest.TestCase):
    """Test cases for the task_func function."""
    @staticmethod
    def is_bar(ax, expected_values, expected_categories):
        extracted_values = [
            bar.get_height() for bar in ax.patches
        ]  # extract bar height
        extracted_categories = [
            tick.get_text() for tick in ax.get_xticklabels()
        ]  # extract category label
        for actual_value, expected_value in zip(extracted_values, expected_values):
            assert (
                actual_value == expected_value
            ), f"Expected value '{expected_value}', but got '{actual_value}'"
        for actual_category, expected_category in zip(
            extracted_categories, expected_categories
        ):
            assert (
                actual_category == expected_category
            ), f"Expected category '{expected_category}', but got '{actual_category}'"
    def test_case_1(self):
        df, ax = task_func(
            [
                ("Allison", 49),
                ("Cassidy", 72),
                ("Jamie", -74),
                ("Randy", -25),
                ("Joshua", -85),
            ]
        )
        # Testing the DataFrame
        self.assertEqual(
            df["Category"].tolist(), ["Allison", "Cassidy", "Jamie", "Randy", "Joshua"]
        )
        self.assertEqual(df["Value"].tolist(), [49, 72, -74, -25, -85])
        # Testing the plot title
        self.assertEqual(ax.get_title(), "Category vs Value")
        self.is_bar(
            ax=ax,
            expected_categories=["Allison", "Cassidy", "Jamie", "Randy", "Joshua"],
            expected_values=[49, 72, -74, -25, -85],
        )

    # More test cases...
```

O Comparison to Existing Programming Benchmarks

Table 9: Correlation coefficients measured by Pearson's r and Spearman's p against existing benchmarks.

	BigCodeBench-Complete		BigCodeBench-Instruct	
	r	p	r	p
MBPP+	0.963	0.937	0.926	0.881
NaturalCodeBench	0.757	0.803	0.913	0.857
HumanEval+	0.849	0.861	0.864	0.894
LiveCodeBench	0.853	0.898	0.778	0.815

We compare model performances on `BigCodeBench-Complete` against existing benchmarks like HumanEval+ (Liu et al., 2024), LiveCodeBench (Jain et al., 2024), MBPP+ (Liu et al., 2024), and NaturalCodeBench (Python-English) (Zhang et al., 2024) used for evaluating the coding capabilities of LLMs. We compute the Pearson and Spearman correlation coefficients for the calibrated model ranks on `BigCodeBench-Complete` and `BigCodeBench-Instruct` against these benchmarks. From Table 9, we observe strong correlations for both coefficients, particularly with MBPP+ showing the highest correlation with `BigCodeBench`. The correlation between NaturalCodeBench and BigCodeBench-Complete is slightly lower, which is expected as NaturalCodeBench prompts are more similar to `BigCodeBench-Instruct`. These results suggest that `BigCodeBench` is well aligned with the mainstream evaluation trend. However, as `BigCodeBench` measures different aspects from these benchmarks, some models are expected to have various ranks among them. Additionally, we note that models which have saturated HumanEval like GPT-4o still have room for improvement on `BigCodeBench` in comparison to human performance.

P Evaluation on Less-Structured Instructions

Although `BigCodeBench-Instruct` targets the conversational setup, the rule-based prompt structure may not fully reflect how the user prompts look. In real-world scenarios, the user prompts can be less structured and more ambiguous. For instance, in the following case shown in the first text block, the user may prompt the model differently, with a less-specified task description (as shown in the second block). The user no longer describes the title name and the return type. To simulate such cases, we preliminarily study several representative models on the rephrased `BigCodeBench-Hard`, where the NL part was rephrased by Llama-3.1-8B-Instruct. For the rephrasing prompt, we use "Please rephrase the following programming description in clear and concise plain text. Keep the core meaning intact but use different wording. Write in a casual style from a user's perspective, without including any task completion markers.\n\n# Programming Description\n". We set the temperature to 0.2 and top_k as 0.95.

````
Train a random forest classifier to perform the classification of the rows in a dataframe with respect to
     the column of interest plot the bar plot of feature importance of each column in the dataframe.
- The xlabel of the bar plot should be 'Feature Importance Score', the ylabel 'Features' and the title '
     Visualizing Important Features'.
- Sort the feature importances in a descending order.
- Use the feature importances on the x-axis and the feature names on the y-axis.
The function should output with:
sklearn.model.RandomForestClassifier : The random forest classifier trained on the input data.
matplotlib.axes.Axes: The Axes object of the plotted data.
You should write self-contained code starting with:

```
from sklearn.ensemble import RandomForestClassifier
import seaborn as sns
import matplotlib.pyplot as plt
def task_func(df, target_column):
```
````

````
I want to use a random forest classifier to predict the class of each row in a dataset based on a specific
      column. I also want to create a bar chart that shows how important each feature is in making those
     predictions. The chart should have a title, labels for the x and y axes, and the features should be
     listed on the y-axis in order of how important they are. The output should include the trained random
      forest classifier and the chart itself. Please start with:

```
from sklearn.ensemble import RandomForestClassifier
import seaborn as sns
````
````

```
import matplotlib.pyplot as plt
def task_func(df, target_column):
```
```

Table 10: Performance comparison across models for original `BigCodeBench-Instruct` (**Original**), rephrased, and the differences.

| Original Model | Original | Rephrased | Δ ↓ |
|---|---|---|---|
| Qwen2.5-Coder-32B-Instruct | 27.7 | 8.8 | -18.9 |
| Llama-3.1-70B-Instruct | 23.6 | 11.5 | -12.1 |
| GPT-4o-mini-2024-07-18 | 23.6 | 10.1 | -13.5 |
| GPT-4o-2024-05-13 | 25.0 | 12.8 | -12.2 |

From Table 10, it is evident that Qwen2.5-Coder-32B-Instruct exhibits a more significant performance drop compared to other models, despite its stronger performance on the original BigCodeBench-Hard. While it is true that rephrased instructions may introduce additional ambiguity, the results indicate that some models may still lack sufficient robustness to handle less structured and more casual inputs. This highlights a potential area for future improvement in LLM capabilities, particularly in managing variations in natural language.

# Q BIGCODEBENCH: EVALUATION INFRASTRUCTURE

In this section, we document the usage of `bigcodebench`, the evaluation infrastructure for `BigCodeBench`. We note that the prototype of `bigcodebench` is based on EvalPlus (Liu et al., 2024).

```
bigcodebench.evaluate \
  --model meta-llama/Meta-Llama-3.1-8B-Instruct \
  --split [complete|instruct] \
  --subset [full|hard] \
  --backend [vllm|openai|anthropic|google|mistral|hf]
```

```
bigcodebench.inspect --dataset [bigcodebench] --eval-results samples-sanitized_eval_results.json [--in-
     place]
```

R DEVELOPMENT TIMELINE

04/2023 - 05/2023 Project Initiation.

06/2023 - 07/2023 Benchmark Construction — Data Synthesis.

07/2023 - 11/2023 Benchmark Construction — Semi-automatic Program Refactoring and Testing Case Generation.

12/2023 - 04/2024 Benchmark Construction — Human Curation; `BigCodeBench` Evaluation Tool Development.

04/2024 - 05/2024 Benchmark Finalization; Experiment; Analysis; Drafting; Code-Eval Development.

06/2024 Initial `BigCodeBench` Release.
```