🌸 BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions

Terry Yue Zhuo[1,2,3]   Chien Vu[4]   Jenny Chim[5]   Han Hu[1,3]   Wenhao Yu[6]
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[12]   Jean Kaddour[13]   Ming Xu[8]   Zhihan Zhang[6]
Prateek Yadav[14]   Naman Jain[15]   Alex Gu[16]   Zhoujun Cheng[17]   Jiawei Liu[18]
Qian Liu[19]   Zijian Wang[20]   Binyuan Hui[8]   Niklas Muennighoff[21]   David Lo[3]
Daniel Fried[22]   Xiaoning Du[1]   Harm de Vries[23]   Leandro von Werra[24]

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]Cornell University   [13]University College London   [14]UNC Chapel Hill   [15]UC Berkeley
[16]MIT   [17]Shanghai Jiaotong University   [18]UIUC   [19]Sea AI Lab   [20]AWS AI Labs*
[21]Contextual AI   [22]Carnegie Mellon University   [23]ServiceNow Research   [24]Hugging Face

🌸 https://bigcode-bench.github.io/
{terry.zhuo@monash.edu; contact@bigcode-project.org}

Abstract

Automated software engineering has been greatly empowered by the recent advances in Large Language Models (LLMs) for programming. While current benchmarks have shown that LLMs can perform various software engineering tasks like human developers, the majority of their evaluations are limited to short and self-contained algorithmic tasks. Solving challenging and practical programming 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 programming tasks, 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 programming tasks. To evaluate LLMs rigorously, each programming 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.

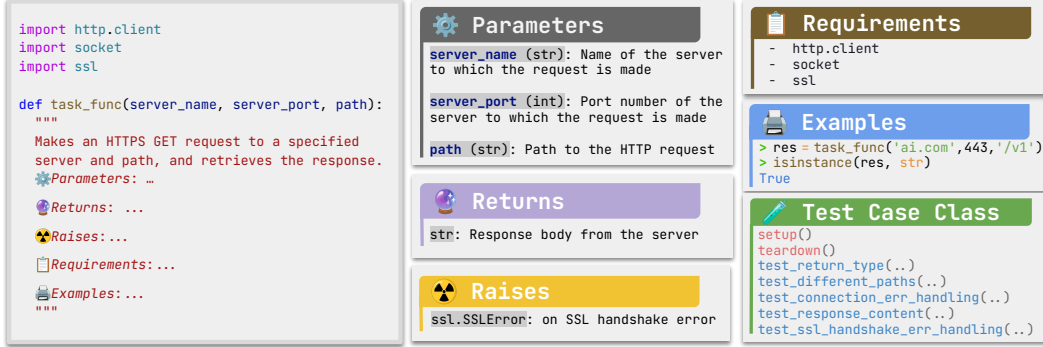*The work does not relate to the author's position at Amazon.

```
import http.client
import socket
import ssl

def task_func(server_name, server_port, path):
    """
    Makes an HTTPS GET request to a specified
    server and path, and retrieves the response.
    ⚙Parameters: …
    🌐Returns: ...
    ☢Raises:...
    📋Requirements:...
    🖨Examples:...
    """
```

⚙ **Parameters**

server_name (str): Name of the server to which the request is made

server_port (int): Port number of the server to which the request is made

path (str): Path to the HTTP request

🌐 **Returns**

str: Response body from the server

☢ **Raises**

ssl.SSLError: on SSL handshake error

📋 **Requirements**
- http.client
- socket
- ssl

🖨 **Examples**
```
> res = task_func('ai.com',443,'/v1')
> isinstance(res, str)
True
```

✏ **Test Case Class**
```
setup()
teardown()
test_return_type(..)
test_different_paths(..)
test_connection_err_handling(..)
test_response_content(..)
test_ssl_handshake_err_handling(..)
```

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

Automated software engineering, like code generation and repair [19, 24, 72], has attracted significant interest from academia and industry to facilitate the development of advanced models. With recent advances in data-centric deep learning techniques, large language models (LLMs) trained on large-scale corpora have shown superior capabilities of generating syntactically correct and functional programs. However, widely-used benchmarks like HumanEval [14] and MBPP [9] 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 realistic software engineering problems, asking ***Can LLMs solve more challenging and more practical programming tasks?***

Solving programming tasks in real-world scenarios typically has two main characteristics: (1) ***Diverse Function Calls as Tools***.[2] A complex programming task often requires the invocation of diverse function call sequences as tools [51, 25, 50]. 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 of software development, 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) [60, 46, 48]. 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 that simulates aforementioned programming 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 [18]. Second, real-world programming scenarios are extremely diverse [71]. Existing benchmarks [68, 32] only focus on popular scenarios like data science. Third, mainstream programming benchmarks have a significant number of ambiguous or under-specified specifications, resulting in inaccurate evaluation results [54, 27]. While there have been attempts to improve the data quality with LLMs [28], LLMs have their own bias and cannot reliably perform refinement [70].

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

---

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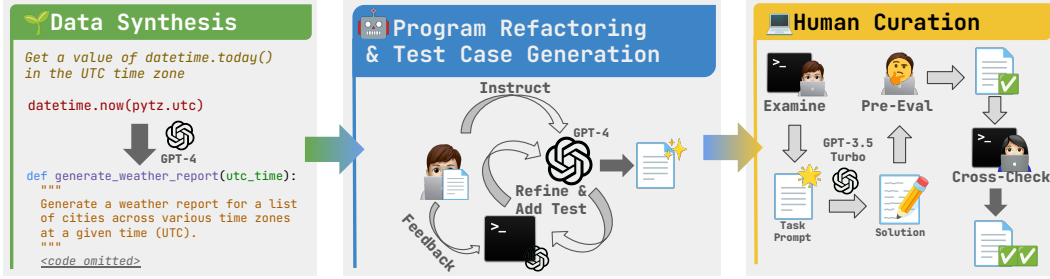
Figure 2: Each programming task in `BigCodeBench` is created through a three-stage construction process. The task quality is controlled by the human-LLM collaboration.

multi-tool-use programming tasks, 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, we design the test cases in an open-ended manner and examine certain behaviors based on different inputs [27]. 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. 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 G. 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 [18], which are hard to self-contain and document. We argue that leveraging LLMs to synthesize customized programming tasks can be more viable [59, 38], 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 [58], a benchmark containing intent-paired code skeletons from Stack Overflow. We use the

GPT-4 API,[3] 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 G.1) crafted by the lead annotator.

As previous studies have shown that LLMs favor their own generations [70, 47], 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 [17], inspired by ReCode [57]. 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 [62]. 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 G.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 G.2.1. The annotators' role is to continually instruct GPT-4 to refractor 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 G.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 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.

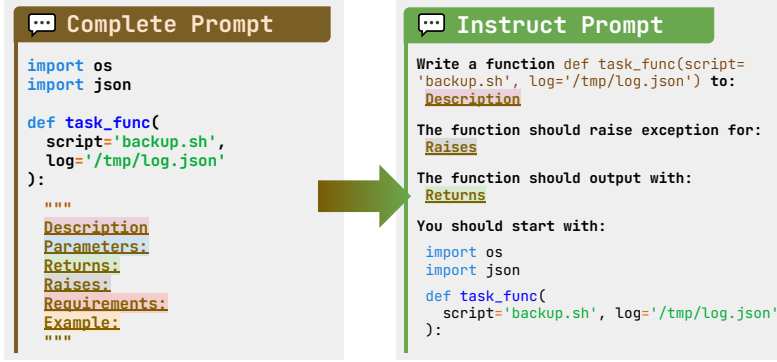[3]We use the model version `gpt-4-0613`.

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.

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 G.3) to add more test cases and resolve a list of runtime issues due to the LLM-generated flaky tests [37]. 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 [27].

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[4] 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 [40]. This training objective aligns with the

---

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

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 [14], users may instruct the models to generate code samples with the less verbose NL context. Despite that there have been similar attempts like HumanEvalPack [40] 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 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 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.
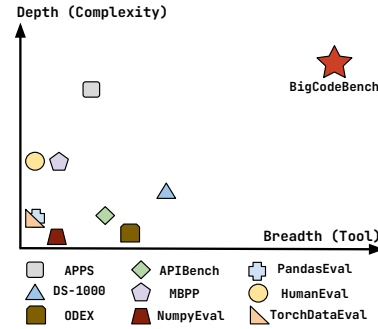
Table 1: Summarized statistics of representative function-level Python programming benchmarks , partially inspired by [69]. `BigCodeBench` are more complex regarding the depth of task complexity and breadth of tool-use knowledge. **PL.**: Programming Language. **Cov.**: Branch Coverage. **Char.**: Characters. **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 [14]	3	4 / 0	7 / 0	0.1	0.1	6	8	5	
DS-1000 [32]	4	5 / 9	7 / 321	0.8	1.1	66	331	24	
DS-1000 (Orig.) [32]	4	4 / 9	5 / 289	0.9	1.3	59	260	23	
ODEX [58]	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	

Tool Statistics In this work, we highlight any libraries and function calls from these libraries as program-based tools for LLMs [50]. 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,

Figure 5: Complexity - tool comparisons with various benchmarks.

and domains in the ground-truth solutions. We visualize the library and function call density used by different benchmarks in Appendix F.1, 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 F.2). We find that BigCodeBench requires more complex reasoning and problem-solving skills to implement comprehensive functionalities.

4 Evaluation

Our evaluation uses the unbiased version of Pass@K [14] 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 [52, 35, 32], 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 I. 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 [35], given in Appendix H.

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. 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 [30], 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 J. We highlight a few findings as follows:
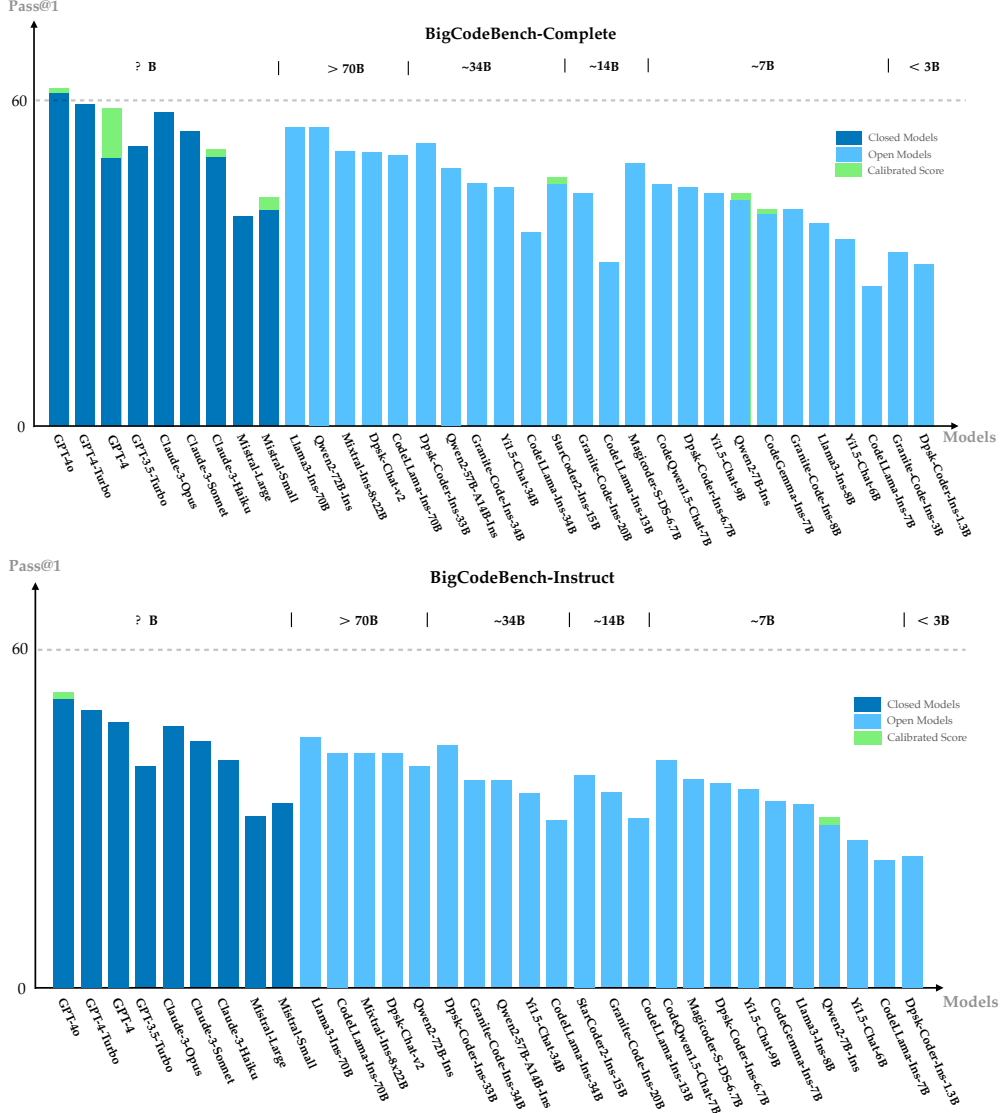

Figure 6: Pass@1 results of instruction-tuned LLMs on `BigCodeBench-Complete` (Top) and `BigCodeBench-Instruct` (Bottom). We only highlight the calibrated results having at least a difference of 1% from the original Pass@1.

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 program-

ming 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 I, we find that GPT-4 tends to omit much more context and perform poorly on `BigCodeBench-Complete`, consistent with the previous community discussion[5] and confirmed by OpenAI [43]. 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 I), 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 the 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.

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

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 I.2.

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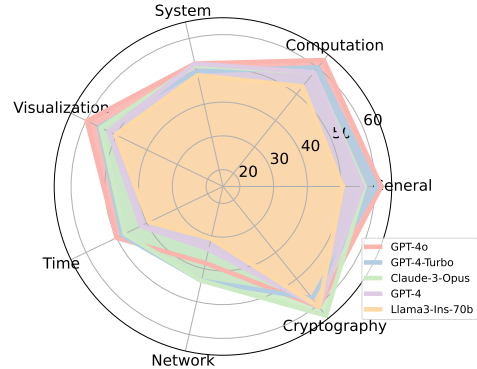
Table 2: Tool-use comparisons between all generated solutions (**Sol.**) and ground truths (**GT.**) on `BigCodeBench-Complete`.

	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

---

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

9

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 J, Examples 9 and 10.

5   Comparison to Existing Programming Benchmarks

Table 3: Correlation coefficients measured by Pearson's r and Spearman's p. against HumanEval+ and LiveCodeBench.

	BigCodeBench-Complete		BigCodeBench-Instruct	
	r	p	r	p
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+ [35] and LiveCodeBench [26] 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 HumanEval+ and Live-CodeBench. From Table 3, we observe the strong correlations for both coefficients, suggesting that `BigCodeBench` is well aligned with the mainstream evaluation trend. However, as `BigCodeBench` measures the different aspects from HumanEval+ and LiveCodeBench, some models are expected to have various ranks among these benchmarks. In addition, we note that models which have saturated HumanEval like GPT-4o, still have room for improvement on `BigCodeBench` in comparison to human performance.

6   Related Work

Large Language Models for Code   With the rise of LLMs, there have been various models trained on code. Codex [14] 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 [41, 34, 36, 52] have been built to perform accurate code completion. Later, with the advance in instruction tuning [45], LLMs can generate the code snippet that aligns with the given NL instruction. Instruction-tuned code models [40, 38, 59], 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 [14] and MBPP [9]. Recently, open-domain coding benchmarks [58, 32] 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 [29] 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 from StackOverflow (collected in [58]), tackling a broader range of software tasks.

7   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 as discussed in our long-term roadmap (Appendix D).

Acknowledgements

References

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

[2] M. AI. Mistral. https://mistral.ai/news/mistral-large/, 2024.

[3] M. AI. Mixtral-8x22b-instruct-v0.1. https://huggingface.co/mistralai/Mixtral-8x22B-Instruct-v0.1, 2024.

[4] M. AI. Mixtral-8x22b-v0.1. https://huggingface.co/mistralai/Mixtral-8x22B-v0.1, 2024.

[5] AI@Meta. Llama 3 model card. 2024.

[6] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, et al. Santacoder: don't reach for the stars! In *Deep Learning for Code (DL4C) Workshop*, 2023.

[7] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of systems and software*, 86(8):1978–2001, 2013.

[8] A. Anthropic. The claude 3 model family: Opus, sonnet, haiku. *Claude-3 Model Card*, 2024.

[9] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[10] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, B. Hui, L. Ji, M. Li, J. Lin, R. Lin, D. Liu, G. Liu, C. Lu, K. Lu, J. Ma, R. Men, X. Ren, X. Ren, C. Tan, S. Tan, J. Tu, P. Wang, S. Wang, W. Wang, S. Wu, B. Xu, J. Xu, A. Yang, H. Yang, J. Yang, S. Yang, Y. Yao, B. Yu, H. Yuan, Z. Yuan, J. Zhang, X. Zhang, Y. Zhang, Z. Zhang, C. Zhou, J. Zhou, X. Zhou, and T. Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[11] E. M. Bender and B. 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.

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

[13] A. Z. Broder. Identifying and filtering near-duplicate documents. In *Annual symposium on combinatorial pattern matching*, pages 1–10. Springer, 2000.

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[15] X. Chen, M. Lin, N. Schaerli, and D. Zhou. Teaching large language models to self-debug. In *The 61st Annual Meeting Of The Association For Computational Linguistics*, 2023.

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

[17] K. Dhole, V. Gangal, S. Gehrmann, A. Gupta, Z. Li, S. Mahamood, A. Mahadiran, S. Mille, A. Shrivastava, S. Tan, et al. Nl-augmenter: A framework for task-sensitive natural language augmentation. *Northern European Journal of Language Technology*, 9(1), 2023.

[18] Y. Ding, Z. Wang, W. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36, 2024.

[19] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*, 2023.

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

[21] T. Gebru, J. Morgenstern, B. Vecchione, J. W. Vaughan, H. Wallach, H. D. Iii, and K. Crawford. Datasheets for datasets. *Communications of the ACM*, 64(12):86–92, 2021.

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

[23] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. 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.

[24] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. C. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review. *ArXiv*, abs/2308.10620, 2023.

[25] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin. Summarizing source code with transferred api knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 2269–2275, 2018.

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

[27] N. Jain, M. Shetty, T. Zhang, K. Han, K. Sen, and I. Stoica. R2e: Turning any github repository into a programming agent environment. In *ICML 2024*.

[28] N. Jain, T. Zhang, W.-L. Chiang, J. E. Gonzalez, K. Sen, and I. Stoica. Llm-assisted code cleaning for training accurate code generators. *arXiv preprint arXiv:2311.14904*, 2023.

[29] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2023.

[30] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

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

[32] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR, 2023.

[33] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang. A systematic review of api evolution literature. *ACM Computing Surveys (CSUR)*, 54(8):1–36, 2021.

[34] R. Li, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, L. Jia, J. Chim, Q. Liu, et al. Starcoder: may the source be with you! *Transactions on Machine Learning Research*, 2023.

[35] J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

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

[37] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 643–653, 2014.

[38] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*, 2023.

[39] M. Mishra, M. Stallone, G. Zhang, Y. Shen, A. Prasad, A. M. Soria, M. Merler, P. Selvam, S. Surendran, S. Singh, et al. Granite code models: A family of open foundation models for code intelligence. *arXiv preprint arXiv:2405.04324*, 2024.

[40] N. Muennighoff, Q. Liu, A. R. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. Von Werra, and S. Longpre. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations*, 2023.

[41] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2022.

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

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

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

[45] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[46] F. Paetsch, A. Eberlein, and F. Maurer. Requirements engineering and agile software development. In *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.*, pages 308–313. IEEE, 2003.

[47] A. Panickssery, S. R. Bowman, and S. Feng. Llm evaluators recognize and favor their own generations. *arXiv preprint arXiv:2404.13076*, 2024.

[48] H. A. Partsch. *Specification and transformation of programs: a formal approach to software development*. Springer Science & Business Media, 2012.

[49] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

[50] Y. Qin, S. Hu, Y. Lin, W. Chen, N. Ding, G. Cui, Z. Zeng, Y. Huang, C. Xiao, C. Han, et al. Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*, 2023.

[51] M. P. Robillard and R. DeLine. A field study of api learning obstacles. *Empirical Software Engineering*, 16:703–732, 2011.

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

[53] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

[54] M. L. Siddiq, S. Dristi, J. Saha, and J. Santos. Quality assessment of prompts used in code generation. *arXiv preprint arXiv:2404.10155*, 2024.

[55] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

[56] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 2024.

[57] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, et al. Recode: Robustness evaluation of code generation models. In *The 61st Annual Meeting Of The Association For Computational Linguistics*, 2023.

[58] Z. Wang, S. Zhou, D. Fried, and G. Neubig. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1271–1290, 2023.

[59] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2023.

[60] K. E. Wiegers and J. Beatty. *Software requirements*. Pearson Education, 2013.

[61] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

[62] C. S. Xia and L. Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

[63] F. Yan, H. Mao, C. C.-J. Ji, T. Zhang, S. G. Patil, I. Stoica, and J. E. Gonzalez. Berkeley function calling leaderboard. https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html, 2024.

[64] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. Swe-agent: Agent-computer interfaces enable automated software engineering.

[65] S. Yang, W.-L. Chiang, L. Zheng, J. E. Gonzalez, and I. Stoica. Rethinking benchmark and contamination for language models with rephrased samples. *arXiv preprint arXiv:2311.04850*, 2023.

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

[67] D. Zan, B. Chen, Z. Lin, B. Guan, W. Yongji, and J.-G. Lou. When language model meets private library. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 277–288, 2022.

[68] D. Zan, B. Chen, D. Yang, Z. Lin, M. Kim, B. Guan, Y. Wang, W. Chen, and J.-G. Lou. Cert: continual pre-training on sketches for library-oriented code generation. *arXiv preprint arXiv:2206.06888*, 2022.

[69] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. 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)*, pages 7443–7464, 2023.

[70] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36, 2024.

[71] X. Zhout, K. Kim, B. Xu, J. Liu, D. Han, and D. 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)*, pages 40–52. IEEE, 2023.

[72] T. Y. Zhuo, Z. Yang, Z. Sun, Y. Wang, L. Li, X. Du, Z. Xing, and D. Lo. Data augmentation approaches for source code models: A survey. *arXiv preprint arXiv:2305.19915*, 2023.

[73] T. Y. Zhuo, A. Zebaze, N. Suppattarachai, L. von Werra, H. de Vries, Q. Liu, and N. 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 [11] 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 G.

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 [21].

C.1   Motivation

C.1.1   For what purpose was the dataset created?

Our dataset aims at providing 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. Repo-level benchmarks (e.g., SWE-bench [64]) are not suitable for this purpose.

• The benchmark should be practical, covering various programming scenarios. Algorithm-specific benchmarks (e.g., HumanEval [14] and MBPP [9]) are unsuitable. Domain-specific benchmarks (e.g., DS-1000 [32]) 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 [58]) 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 is released to the public, and hosted on GitHub and 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 GitHub 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, please contact Terry Yue Zhuo (terry.zhuo@monash.edu) and BigCode Project (contact@bigcode-project.org), who are responsible for maintenance.

D 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 [20] 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.

D.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, we may not be able 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 used by software developers, 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[6]. 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 [37], 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.

Efficiency   Currently, our execution-based evaluation requires non-trivial computational resources on CPUs. Our machines with 36 CPU cores typically execute the tests for 1,140 samples under 10 minutes, with the default time limits. For machines with few cores, it may take 1 - 2 hours to finish, which is quite costly. The main reason behind the long-time execution is due to the heavy dependency on external libraries and the complex computation required in specific programming tasks. We will make an effort to improve the efficiency of the evaluation to ease the burden.

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 [35] 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 [7] methods cover all possible scenarios. Therefore, we still consider LLM-based test generation [56] 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 `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 [68] 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 [33], 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

---

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

both a public set and a private set for better evaluation in a recent blog[7]. 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* [61], 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 [15] and self-reflection [53]. We tend to work in this direction and see how well *LLMs as Agents* can perform on BigCodeBench.

D.2   BigCodeBench-**Mini**

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 plan to release a minimal high-quality subset of BigCodeBench, serving as a proxy for the full set. To conclude the best practice, We will explore the evaluation results with multiple approaches like manual selection, hash-based deduplication [13] and LLM-based decontamination [65].

D.3   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 [71]. 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.

D.4   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.

E   Artifacts

---

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

Table 4: Artifacts for reproducibility.

Name	Public Link or Endpoint
BigCodeBench (v0.1.0)	
GitHub	https://github.com/bigcode-project/bigcodebench-annotation/releases/tag/v0.1.0
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
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

F   Tool Statistics

F.1   Analysis
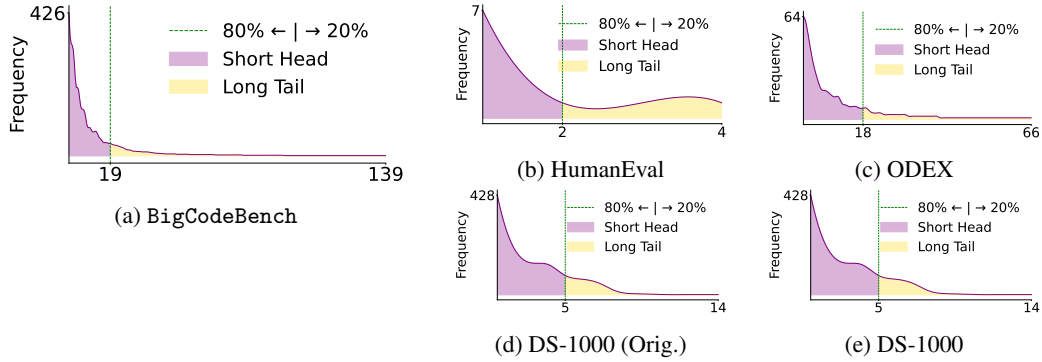
Figure 8: Library density comparisons. We sort the libraries by frequency count, showcasing the long-tail distribution to highlight the broad diversity within `BigCodeBench`.
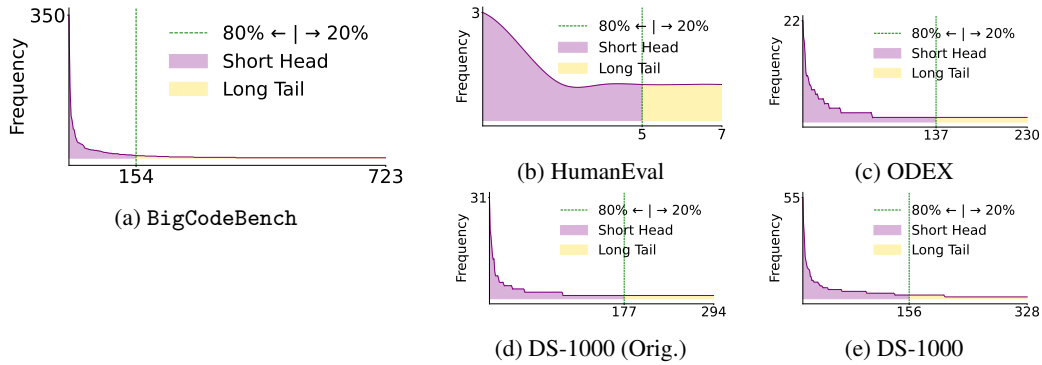
Figure 9: Function call density comparisons. We sort function calls by frequency count, showcasing the long-tail distribution to highlight the broad diversity within `BigCodeBench`.

F.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 [23]	352.9	137
DS-1000 [32]	138.1	328
ODEX [58]	50.4	230
APIBench [49]	77.5	171
MBPP [9]	181.1	51
NumpyEval [68]	30.1	52
PandasEval [68]	45.8	12
HumanEval	180.9	7
TorchDataEval [67]	52.8	6
BigCodeBench	426.0	723

F.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
```

F.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",
```

```json
        "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"
}
```

G Detailed Benchmark Construction

G.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
```

```python
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:
```

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

G.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"].
```

G.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 .
```

G.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.
```

H   Evaluation Setup

H.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.

H.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.

H.3   Prompt Template

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

Figure 10: Prompt template for models supported by vLLM [31].

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

Figure 11: Prompt template for OpenAI/DeepSeek APIs.

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

Figure 12: Prompt template for Mistral model APIs.

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

Figure 13: Prompt template for Anthropic model APIs.

I Detailed Benchmarking Results and Analysis

I.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		Random Sampling	
		Original	Calibrated	$Pass@1$	$Pass@5$
Instruction-tuned LLMs					
GPT-4o [44]	2024-05-13	0.602	0.611	0.557	0.711
GPT-4-Turbo [43]	2024-04-09	0.582	0.582	0.563	0.699
GPT-4 [1]	0613	0.484	0.572	0.407	0.682
GPT-3.5-Turbo [42]	0125	0.506	0.506	0.504	0.657
Claude-3 [8]	Opus	0.568	0.574	—	—
	Sonnet	0.534	0.538	—	—
	Haiku	0.486	0.501	—	—
Mistral [2]	Large	0.379	0.383	0.365	0.539
	Small	0.390	0.413	0.398	0.601
DeepSeek-Chat [16]	V2	0.494	0.494	0.482	0.596
DeepSeek-Coder-instruct [22]	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 [52]	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 [5]	70B	0.539	0.545	0.522	0.650
	8B	0.368	0.369	0.338	0.562
Qwen2-Instruct [10]	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 [10]	7B	0.437	0.443	0.408	0.632
Qwen1.5-Chat [10]	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 [66]	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 [3]	8x22B	0.496	0.502	0.458	0.677
Magicoder-S-DS [59]	6.7B	0.475	0.476	0.424	0.643
CodeGemma-instruct [55]	7B	0.383	0.393	0.334	0.568
StarCoder2-Instruct [12]	15B	0.437	0.451	0.404	0.610
Granite-Code-Instruct [39]	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 [52]	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 [5]	70B	0.433	—	0.375	0.625
	8B	0.288	—	0.223	0.466
DeepSeek-Coder-base [22]	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 [10]	7B	0.456	—	0.411	0.650
Yi-1.5 [66]	34B	0.400	—	0.302	0.575
	9B	0.355	—	0.295	0.563
	6B	0.274	—	0.201	0.435
Mixtral-base [4]	8x22B	0.455	—	0.355	0.633
CodeGemma [55]	7B	0.373	—	0.295	0.557
	2B	0.239	—	0.153	0.375
StarCoder2 [36]	15B	0.383	—	0.332	0.609
	7B	0.285	—	0.232	0.514
	3B	0.214	—	0.178	0.416
Granite-Code [39]	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

I.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 [63].

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 J Examples 6, 7, and 8.

Figure 14: Top 5 base LLMs

J   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.

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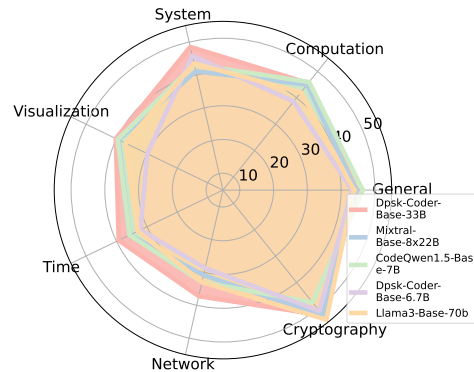from collections import Counter
import random
from itertools import cycle

# 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)`.

```python
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`.

```python
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.

```python
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 = {}

    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.

```
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.

```
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))]
        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
```

K `bigcodebench`: A Viable Evaluation Tool

In this section, we document the usage of `bigcodebench`, a viable evaluation tool for BigCodeBench. We note that the prototype of `bigcodebench` is based on EvalPlus [35].

K.1 Model Inference

```
bigcodebench.generate \
    --model [model_name] \
    --subset [complete|instruct] \
    --greedy \
    --bs [bs] \
    --temperature [temp] \
    --n_samples [n_samples] \
    --resume \
    --backend [vllm|hf|openai|mistral|anthropic|google] \
    --tp [gpu_number] \
    [--trust_remote_code] \
    [--base_url [base_url]]
```

K.2 Output Sanitizing

```
bigcodebench.sanitize --samples samples.jsonl [--calibrate]
```

K.3 Execution-based Evaluation

```
bigcodebench.evaluate --subset [complete|instruct] --samples samples-sanitized-calibrated.jsonl
```

K.4 Result Inspection

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

L   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.