LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code

Naman Jain[†] King Han[†] Alex Gu[*] [$] Wen-Ding Li[*‡]

Fanjia Yan[*†] Tianjun Zhang[*†] Sida I. Wang

Armando Solar-Lezama[$] Koushik Sen[†] Ion Stoica[†]

[†] UC Berkeley [$] MIT [‡] Cornell

Website: https://livecodebench.github.io/

{naman_jain,kingh0730,fanjiayan,tianjunz,ksen,istoica}@berkeley.edu
gua@mit.edu asolar@csail.mit.edu wl678@cornell.edu

Abstract

Large Language Models (LLMs) applied to code-related applications have emerged as a prominent field, attracting significant interest from both academia and industry. However, as new and improved LLMs are developed, existing evaluation benchmarks (e.g., HumanEval, MBPP) are no longer sufficient for assessing their capabilities. In this work, we propose LiveCodeBench, a comprehensive and contamination-free evaluation of LLMs for code, which collects *new* problems over time from contests across three competition platforms, namely LeetCode, AtCoder, and CodeForces. Notably, our benchmark also focuses on a broader range of code-related capabilities, such as self-repair, code execution, and test output prediction, beyond just code generation. Currently, LiveCodeBench hosts over five hundred coding problems that were published between May 2023 and May 2024. We have evaluated 18 base LLMs and 34 instruction-tuned LLMs on LiveCodeBench. We present empirical findings on contamination, holistic performance comparisons, potential overfitting in existing benchmarks as well as individual model comparisons. We will release all prompts and model completions for further community analysis, along with a general toolkit for adding new scenarios and models.

1 Introduction

Code has emerged as an important application area for LLMs, with a proliferation of code-specific models (Chen et al., 2021; Austin et al., 2021; Li et al., 2022; Zhong et al., 2022; Allal et al., 2023; Li et al., 2023b; Roziere et al., 2023; Guo et al., 2024; Luo et al., 2023; Royzen et al., 2023; Wei et al., 2023b; Ridnik et al., 2024; Lozhkov et al., 2024) and their applications across various domains and tasks such as program repair (Zheng et al., 2024; Olausson et al., 2023), optimization (Madaan et al., 2023a), test generation (Steenhoek et al., 2023), documentation generation (Luo et al., 2024), tool usage (Patil et al., 2023; Qin et al., 2024), SQL (Sun et al., 2023), and more. In contrast with these rapid advancements, evaluations have remained relatively stagnant, and current benchmarks like HumanEval, MBPP, and APPS may paint a skewed or misleading picture. Firstly, while coding is a multi-faceted skill, these benchmarks only focus on natural language-to-code tasks, thus overlooking broader code-related capabilities. Moreover, these benchmarks may be subject to potential contamination or overfitting, as benchmark samples are present in the training datasets.

Motivated by these shortcomings, we introduce LiveCodeBench, a holistic and contamination-free benchmark for evaluating code capabilities. LiveCodeBench is built on the following principles:

1. **Live updates to prevent contamination.** LLMs are trained on massive inscrutable corpora, and current benchmarks suffer from the risk of data contamination as they could be included in those training datasets. While previous works have attempted decontamination using both exact and fuzzy matches (Li et al., 2023b,d), it can be a non-trivial task (Team, 2024) and can be evaded using simple strategies like rephrasing (Yang et al., 2023). Here, to

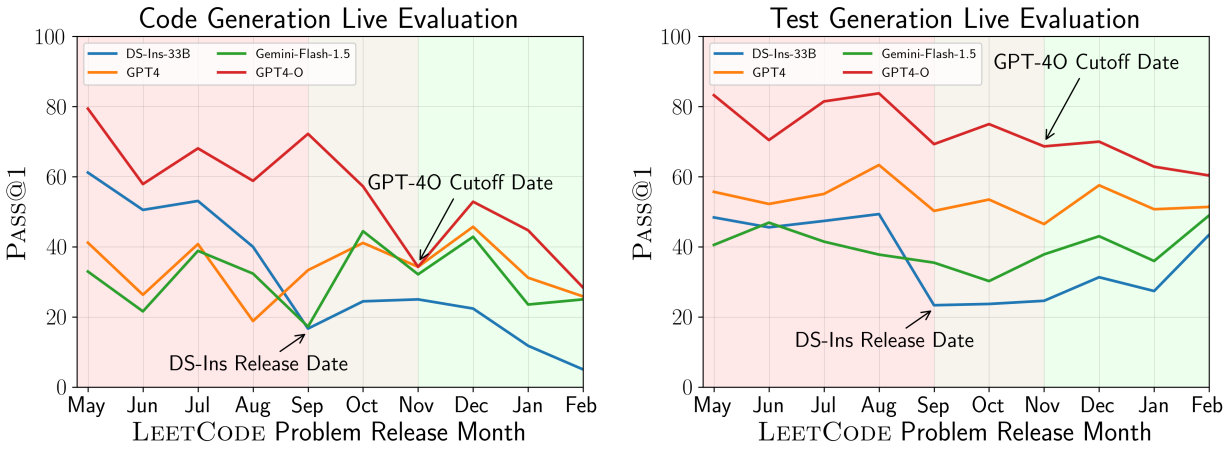

Figure 1: LiveCodeBench comprises problems marked with release dates, allowing evaluations over different time windows. For newer models, we can detect and avoid contamination by only evaluating on time-windows after the model's cutoff date. The figures demonstrate the performance of models on code generation and test output prediction LiveCodeBench scenarios with LeetCode problems released across the months between May 2023 and February 2024. Notice that DeepSeek-Instruct and GPT-4-O perform considerably worse on problems released since September and November 2023 (their release and cutoff dates respectively!) – indicating potential contamination for the earlier problems. Thus, while performing evaluations, we use the post-September/post-November time window (green) for fairly comparing these models.

prevent the risk of problem contamination, we use live updates, that is evaluate models on *new* problems. Particularly, we collect problems from weekly contests on competition platforms and tag them with a *release date*. Next, for newer models, we only consider problems released after the model's cutoff date to ensure that the model has not encountered the exact problem in the training dataset. In Figure 1, we find that the performance of the DeepSeek model starkly drops when evaluated on the LeetCode problems released after August 2023. Similarly, GPT-4-O observes a drop in performance on LeetCode problems released since November 2023, its specified cutoff date. This indicates that these models are likely trained on the older LeetCode problems and time-segmented evaluations allow fair comparisions.

2. **Holistic Evaluation.** Current code evaluations primarily focus on natural language to code generation. However, programming is a multi-faceted task that requires a variety of capabilities beyond those measured by code generation. In LiveCodeBench, we evaluate code LLMs on three additional scenarios, listed below.

    - **Self-Repair.** Fix an incorrect program from execution information, evaluating the ability to debug code from feedback. The model is given the natural language problem description, the incorrect program, the test case it fails on, and the execution feedback from that failure. The output should be a correct repaired program.
    - **Code Execution.** "Execute" a program on an input, evaluating code comprehension ability. The model is given a program and an input, and the output should be the result.
    - **Test Output Prediction.** Solve the natural language task on a specified input, evaluating the ability to generate testing outputs. The model is given the natural language problem description and an input, and the output should be the output for the problem.

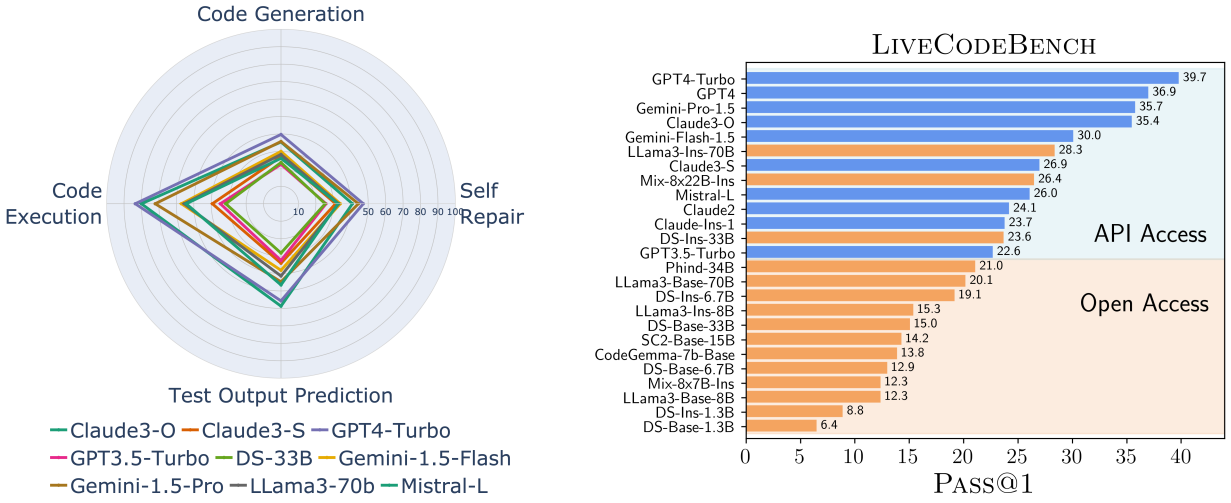

Figure 2: **Left.** We propose evaluating LLMs across scenarios capturing various coding-related capabilities. Specifically, we host four different scenarios, namely code generation, self-repair, code execution, and test output prediction. The figure depicts various model performances across the four scenarios available in LiveCodeBench in a radial plot – highlighting how relative differences across models change across the scenarios. **Right.** Comparison of open access and (closed) API access models on LiveCodeBench-Easy code generation scenario. We find that closed-access models consistently outperform the open models with only strong instruction-tuned variants of $>$ 30B models (specifically L3-Ins-70B, Mixtral and DS-Ins-33B models) crossing the performance gap.

Figure 2 (left) depicts performance on the different scenarios considered in LiveCodeBench.

3. **High-quality problems and tests.** High-quality problems and tests are crucial for reliable evaluation of LLMs. However, prior works have revealed deficiencies in existing benchmarks. (Liu et al., 2023a) identified insufficient tests and ambiguous problem descriptions in HumanEval. They released HumanEval+, a variant of the benchmark with more tests and sometimes saw up to an 8% drop in performance. Similarly, (Austin et al., 2021) had to create a sanitized MBPP subset to disambiguate problem descriptions. In LiveCodeBench, we source the problems from reputable competition websites whose quality is already validated by the platform users. In addition, for every problem, we provide a good number of tests (about 17 on average) for meaningful and robust evaluations while still finishing quickly.
4. **Balanced problem difficulty.** Competition programming is challenging for even the best-performing LLMs, and most of the current SoTA models achieve close to zero performance on a majority of problems. As a result, they can be unsuitable for meaningful comparing today's LLMs because the variance in performances is low. Furthermore, the averaging of evaluation scores across problems with different difficulty levels artificially minimizes the differences between models. Therefore, we use problem difficulty ratings (sourced from the competition websites) for filtering the harder problems and classifying problem difficulties to ensure balanced problem difficulty distribution and allow granular model comparisons.

With these principles in mind, we build LiveCodeBench, a continuously updated benchmark that avoids data contamination. Particularly, we have collected 511 problems from contests across three competition platforms – LeetCode, AtCoder, and CodeForces occurring from May 2023 to the present (May 2024) and use them to construct the different LiveCodeBench scenarios.

Empirical Findings. We have evaluated 18 base models and 34 instruction-tuned models across different LiveCodeBench scenarios. Below, we present the empirical findings from our evaluations, which have not been revealed in prior benchmarks.

1. **Contamination.** We observe a stark drop in the performance of DeepSeek, GPT-4-O, and Codestral on LeetCode problems released after Aug 2023, Oct 2023, and Jan 2024 (Figure 1). These results highlight likely contamination in older problems and time-segmented evaluations prove effective for performing fair comparisons.
2. **Holistic Evaluation.** Our evaluations reveal that model performances are correlated across tasks, but the relative differences do vary. For example, in Figure 2, the gap between open and closed models further increases on tasks like self-repair or test output prediction. Similarly, Claude-3-Opus and Mistral-L perform considerably better on code execution and test output prediction compared to code generation with Claude-3-Opus surpassing GPT-4 on the test output prediction. This highlights the importance of a holistic evaluation.
3. **HumanEval Overfitting.** Upon comparing LiveCodeBench with HumanEval, we find that models cluster into two groups, ones that perform well on both benchmarks and others that perform well on HumanEval but not on LiveCodeBench (see Figure 5). The latter group primarily comprises fine-tuned open-access models while the former group comprises base models and closed models. This indicates that these models might be overfitting to HumanEval.
4. **Model Comparisons** (Figure 4)
   (a) Among the open access base models, we find that L3-Base and DeepSeek-Base models are the strongest, followed by StarCoder2-Base and CodeLLaMa-Base.

(b) Closed API models such as GPTs, Claude, and Gemini generally outperform open models (Figure 2). The open models that close the gap are L3-Ins-70B, Mixtral, and DS-Ins-33B are instruction-tuned variants of large base models (> 30B parameters).
(c) Existing benchmarks are insufficient at highlighting the gap between GPT-4 and other models. Particularly, smaller models achieve similar or often even better performance compared to GPT-4. In LiveCodeBench, GPT-4 (and GPT-4-Turbo) outperforms all other models (except Claude-3-Opus) by a large margin in all scenarios.

Concurrent Work. (Huang et al., 2023) also evaluate LLMs in a time-segmented manner. However, they only focus on CodeForces problems while we combine problems across platforms and additionally propose a holistic evaluation across multiple code-related scenarios. (Li et al., 2023c) propose a large dataset of competitive programming problems with additional generated tests but do not study contamination or tasks beyond generation. Liu et al. (2024) evaluate the code comprehension capabilities of LLMs using execution. (Singhal et al., 2024) also propose evaluating LLMs on tasks beyond code generation, but they consider tasks that take into account the *non-functional-correctness aspects* of programming. (Guo et al., 2024) also evaluate DeepSeek on LeetCode problems and mention the possibility of problem contamination.

2 Holistic Evaluation

Code capabilities of LLMs are evaluated and compared using natural language to code generation tasks. However, this only captures one dimension of code-related capabilities. Indeed, real-world software engineering requires expertise in tasks beyond just *generation*, such as synthesizing informative test cases, debugging incorrect code, understanding existing code, and writing documentation. These tasks are not just additional bookkeeping; they are crucial parts of the software development process and contribute to improving the quality, maintainability, and reliability of the code (Boehm, 2006). This also applies to LLMs and adopting similar workflows can enable the models to perform better code generation. For example, AlphaCodium (Ridnik et al., 2024) is an intricate LLM pipeline for solving competition coding problems. By combining natural language reasoning, test case generation, code generation, and self-repair, they achieve significant improvements over a naive direct code generation baseline, showcasing the importance of these broader capabilities. Motivated by this, we propose a more holistic evaluation of LLMs in this work using a suite of evaluation setups that capture a broader range of code-related capabilities.

Specifically, we evaluate code LLMs in four scenarios, namely code generation, self-repair, code execution, and test output prediction. Our selection criterion was to pick settings that are useful components in code LLM workflows and in addition, have clear and automated evaluation metrics.

Following we describe each of these scenarios in detail.

Code Generation. The code generation scenario follows the standard setup for generating code from natural language. The model is given a problem statement, which includes a natural language description and example tests (input-output pairs), and is tasked with generating a correct solution. The evaluation is performed based on functional correctness, using a set of *unseen* test cases. We use the Pass@1 metric measured as the fraction of the problems for which the model was able to generate a program passing all tests. Figure 3 (left) provides an example of this scenario.

Self Repair. The self-repair scenario is based on previous works that tested the self-repair capabilities of LLMs (Olausson et al., 2023; Shinn et al., 2023; Chen et al., 2023). Here, the model is given

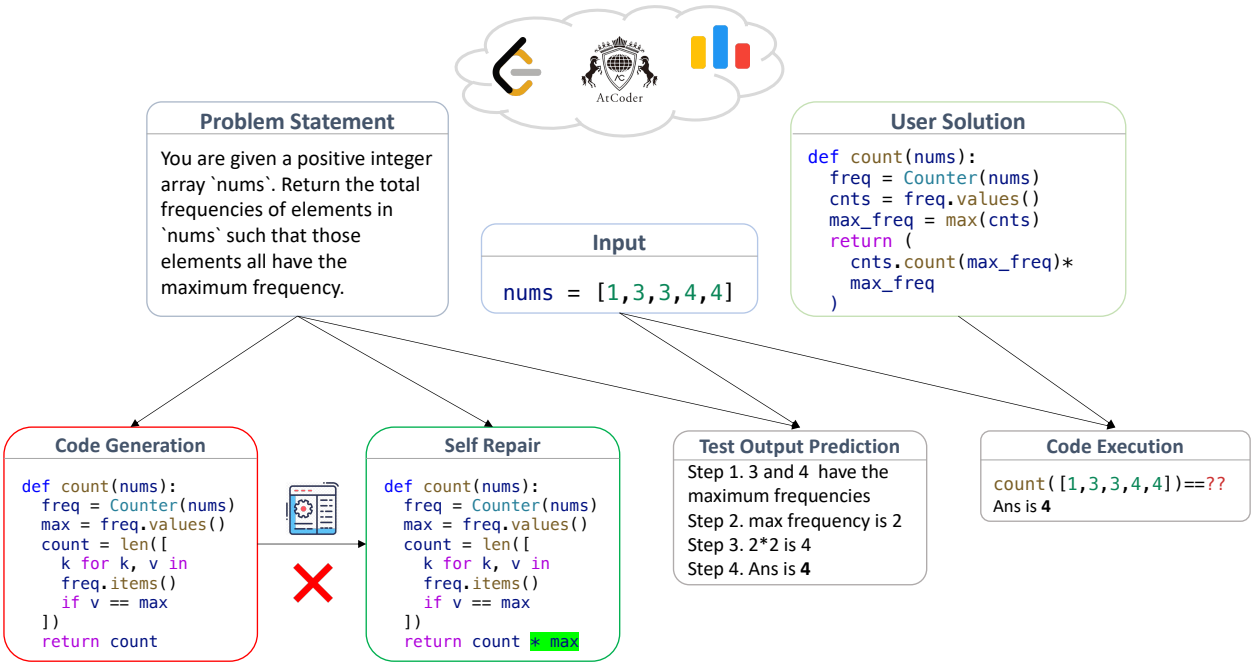

Figure 3: Overview of the different scenarios present in LiveCodeBench. Coding is multi-faceted and we propose evaluating LLMs on a suite of evaluation setups that capture various coding-related capabilities. Specifically, beyond the standard code generation setting, we consider three additional scenarios, namely self-repair, code execution, and a newly introduced test output prediction task.

a problem statement from which it generates a candidate program (similar to the single-step code generation scenario above). However, in case of a mistake, the model is additionally provided with error feedback (either the exception message or a failing test case in case of incorrect code generation) and is tasked with generating the fixed solution. Similar to the code generation scenario, the evaluation is performed via functional correctness on the final program, i.e. either the single-step correct generation or the attempted repair. We use the Pass@1 metric to measure the combined performance after the repair step. Figure 3 (mid-left) provides an example of this scenario.

Code Execution. The code execution scenario is based on the output prediction setup used in CRUXEval (Gu et al., 2024). The model is provided a program snippet consisting of a function (`f`) along with a test input to the program and is tasked with predicting the output of the program on the input test case. The evaluation is performed via an execution-based correctness metric where the model generation is considered correct if `assert f(input) == generated_output` passes. Figure 3 (right) provides an example of the code execution scenario.

Test Case Output Prediction. Finally, we introduce a new task that is designed to study natural language reasoning and test generation. In this task, the model is given the problem statement along with a test case input, and it is tasked with generating the expected output for that input. This task follows a setup similar to the one used in CodeT (Chen et al., 2022), where tests are generated solely from problem statements, without the need for the function's implementation. A key difference is that we provide a fixed set of test inputs for each problem in our dataset, and the models are then prompted to only predict the expected output for those specific inputs. This approach allows for a straightforward evaluation of the test generation capabilities by avoiding test input prediction, a hard-to-evaluate task. Figure 3 (mid-right) provides an example of this scenario.

Finally, we would like to point out that LiveCodeBench also offers an extensible framework to add new scenarios in the future. So other relevant settings like input generation, program summarization, optimization, etc. can be integrated with our setup.

3 Benchmark Curation

We curate our problems from three coding competition websites: LeetCode, AtCoder, and CodeForces. These websites periodically host contests containing problems that assess the coding and problem-solving skills of participants. The problems consist of a natural language problem statement along with example input-output examples, and the goal is to write a program that passes a set of hidden tests. Further, thousands of participants participate, solving these problems thus ensuring that the problems are vetted for clarity and correctness.

3.1 Data Collection

We have written HTML scrapers for each of the above websites to collect problems and the corresponding metadata. To ensure quality and consistency, we parse mathematical formulas and exclude problems with images. We also exclude problems that are not suitable for grading by input-output examples, such as those that accept multiple correct answers or require the construction of data structures. Besides parsing the problem descriptions, we also collect associated ground truth solutions and test cases whenever directly available. Thus for each problem, we collect tuples of natural language problem statement P, test cases T, and ground truth solution S. Finally, we associate the contest date D to mark the release date of each problem and use the collected attributes to construct problems for our four scenarios (detailed in Section 3.3 ahead).

Scrolling through time. As noted, we associate the contest date D for each problem. The release date allows us the measure the performance of LLMs over different time windows by filtering problems based on whether the problem release date falls within a time window (referred to as "scrolling" through time). This is crucial for evaluating and comparing models trained at different times. Specifically, for a new model and the corresponding cutoff date (normalized to the release date if the training cutoff date is not published), we can measure the performance of the model on benchmark problems released after the cutoff date. We have developed a UI that allows comparing models on problems released during different time windows (shown in Figure 9).

Test collection. Tests are crucial for assessing the correctness of the generated outputs and are used in all four scenarios. We collect tests available on platform websites whenever possible and use them for the benchmark. Otherwise, following Liu et al. (2023b), we use a LLM (here GPT-4-Turbo) to generate tests for the problems. A key difference between our test generation approach is that instead of generating inputs directly using the LLM, we construct generators that sample inputs based on the problem specifications using in context learning. Details and examples of such input generators can be found in Section A.2. Finally, we collect a small fraction of failing tests from the platform for the more recent problems allowing more directed adversarial test collection.

Problem difficulty. Competition programming has remained a challenge for LLMs, with GPT-4 achieving an average CodeForces rating (ELO) of 392, placing it in the bottom 5 percentile (OpenAI, 2023). This makes it difficult to compare LLMs, as the variation in performance across models is low. In LiveCodeBench, we collect problems of diverse difficulties as labeled in competition platforms, excluding problems that are rated above a certain threshold that are likely too

Platform	Total Count	#Easy	#Medium	#Hard	Average Tests
LCB (May-end)	511	182	206	123	17.0
LCB (Sep-end)	349	125	136	88	18.0
AtCoder	267	99	91	77	15.6
LeetCode	235	79	113	43	19.0
CodeForces	9	4	2	3	11.1
LCB-Easy	182	182	0	0	16.1
LCB-Medium	206	0	206	0	17.4
LCB-Hard	123	0	0	123	18.0

Table 1: The statistics of problems collected in LiveCodeBench (LCB). We present the number of problems, their difficulty distributions and the average number of tests per problem. We present the results on the following subsets of LiveCodeBench (used throughout this manuscript) - (a) problems in the May'23-May'24 and Sep'23-May'24 time windows, (b) problems sourced from the three platforms, and (c) problems in the LCB-Easy, LCB-Medium, and LCB-Hard subsets.

difficult for even the best models[1]. Further, we use these ratings to classify problems as Easy, Medium, and Hard for more granular model comparisons.

3.2 Platform Specific Curation

We describe the curation process for each platform.

LeetCode. We collect problems from all weekly and biweekly contests on LeetCode that have taken place after April'23. For each problem, we collect the problems, public tests, and user solutions. The platform also provides a difficulty label for each problem which we use to tag the problems as Easy, Medium, and Hard. Since LeetCode provides a starter code for each problem, we also collect it and provide it to the LLM in the STDIN format. Since the hidden tests are not directly available, we use our generator-based test input generation approach (Section A.2) and also collect the auto grader failing tests for some of the recent problems.

AtCoder. We collect problems from the `abc` (beginner round) contests on AtCoder that have taken place after April'23. We deliberately avoid the more challenging `arc` and `agc` contests which are designed for more advanced Olympiad participants. The problems are assigned numeric difficulty ratings, and we exclude `abc` problems with a rating of more than 500. We also use these numeric ratings to tag the problems as Easy, Medium, and Hard. Specifically, we use the rating brackets $[0-200)$, $[200-400)$, and $[400-500]$ to perform the classification. AtCoder provides public and hidden tests for each problem which we directly use in the benchmark.

CodeForces. We have collected problems from the Division 3 and Division 4 contests on CodeForces. Notably, we find that even with this filter, the problems are harder than the other two platforms. CodeForces also provides difficulty ratings for the problems which we use to tag the problems as Easy, Medium, and Hard using the rating brackets $\{800\}$, $(800-1000]$, and $(1000-1300]$ respectively. Due to the higher difficulty, we only consider a small fraction of problems from CodeForces and semi-automatically construct test case generators, as they do not provide complete tests on the platform (long tests are truncated).

[1]From our early explorations, we find CodeForces problems being considerably more difficult than AtCoder and LeetCode problems and thus focus primarily on the latter platforms.

Table 1 provides various statistics about the problems that we have collected for LiveCodeBench.

3.3 Scenario-specific benchmark construction

Code Generation and Self-Repair. We use the natural language problem statement as the problem statement for these scenarios. For LeetCode, as noted above, an additional starter code is provided for the functional input format. For AtCoder and CodeForces problems, we use the standard input format (similar to Hendrycks et al. (2021)). The collected or generated tests are then used to evaluate the correctness of the generated programs. Our final dataset consists of 511 problem instances across the three platforms.

Code Execution. We draw inspiration from the benchmark creation procedure used in Gu et al. (2024). First, we collect a large pool of ~ 2000 *correct, human-submitted solutions* from the LeetCode subset. However, many of these programs have multiple nested loops, complex numerical computations, and a large number of execution steps. Therefore, we apply compile-time and run-time filters to ensure samples are reasonable, and we double-check this with a manual inspection. More details on the filtering criteria and statistics of the dataset can be found in Appendix A.3. Our final dataset consists of 479 samples from 85 problems.

Test Case Output Prediction. We use the natural language problem statement from the LeetCode platform and the example test inputs to construct our test case output prediction dataset. Since the example test inputs in the problems are reasonable test cases for humans to reason about and understand the problems, they also serve as ideal test inputs for LLMs to process. Our final dataset consists of 442 problem instances from a total of 181 LeetCode problems.

4 Experiment Setup

We describe the experimental setup in this section. First, we provide the common setup across the scenarios, followed by the scenario-specific setups in Section 4.1.

Models. We evaluate 52 models across various sizes, ranging from 1.3B to 70B, including base models, instruction models, and both open and closed models. Our experiments include models from different classes, such as GPTs (GPT-3.5-turbo, GPT-4, GPT-4-Turbo,GPT-4-O), Claudes (Claude-Ins-1, Claude-2, Claude-3s), Geminis(Gemini-Pro, Gemini-Flash), Mistral among closed-access and LLaMa-3s(L3-Base-{7, 70}B, L3-Ins-{7, 70}B), DeepSeeks (DS-Base-{1.3, 6.7, 33}B, DS-Ins-{1.3, 6.7, 33}B), CodeLLaMas (CL-Ins-{7, 13, 34}B, CL-Base-{7, 13, 34}B), StarCoder2 (SC2-Base-{3,7,15}B), CodeQwen among open. Additionally, we also include fine-tuned models Phind-34B from CL-Base-34B, and MagiCoders (MC-{6.7, 7}B) from CL-Base-7B and DS-Base-6.7B. See Appendix C.1 for a complete list of models and estimated cutoff dates.

Evaluation Metrics. We use the Pass@1 (Kulal et al., 2019; Chen et al., 2021) metric for our evaluations. Specifically, we generate 10 candidate answers for each problem either using API or using vLLM (Kwon et al., 2023). We use nucleus sampling with temperature 0.2 and top_p 0.95 and calculate the fraction of programs or answers that are correct. For the code generation and self-repair scenarios, we use tests to verify the correctness of the programs. For these scenarios, programs must pass all tests to be considered correct. For the code execution scenario, we use an execution-based correctness metric between the generated output and the ground truth output. For the test output prediction scenario, we parse the generated response to extract the answer and

use equivalence checks for grading as specified in Section 2.

4.1 Scenario-specific setup

The setup for each scenario is presented below. Note that the base models are only used in the code generation scenario since they do not easily follow the format for the other scenarios.

Code Generation. For the instruction-tuned models, we use a zero-shot prompt and follow the approach of Hendrycks et al. (2021) by adding appropriate instructions to generate solutions in either functional or stdin format. For the base models, we use a constant one-shot example, with a separate example provided for problems that accept stdin input and for problems that accept functional output. Section C.2 shows the high-level zero-shot prompt used.

Self Repair. Similar to prior work Olausson et al. (2023), we use the programs generated during the code generation scenario along with the corresponding error feedback to build the zero-shot prompt for the self-repair scenario. The type of error feedback includes syntax errors, runtime errors, wrong answers, and time-limit errors, as applicable. Section C.3 provides the pseudo-code for computing the error feedback and the corresponding prompt.

Code Execution. We use few-shot prompts for the code execution scenario, both with and without chain-of-thought prompting (COT). Particularly, we use a 2-shot prompt without COT and a 1-shot prompt with COT with manually detailed steps. The prompts are detailed in Section C.4.

Test Output Prediction. We use a zero-shot prompt that queries the model to complete assertions, given the problem, function signature, and test input. We provide the prompt in Section C.5.

5 Results

We first describe how LiveCodeBench helps detect and avoid benchmark contamination in Section 5.1. Next, we present the findings from our evaluations on LiveCodeBench in Section 5.2.

5.1 Avoiding Contamination

A distinguishing aspect of our benchmark is the ability to evaluate models on problems released over different time windows. This allows us to measure the model performance on problems released after the cutoff date, thereby giving a performance estimate on *unseen* problems.

Contamination in DeepSeek and GPT-4-O. LiveCodeBench comprises problems released since May 2023. However, DeepSeek models were released in Sep 2023 and might have already been trained on some of the problems in our benchmark. Similarly, OpenAI notes GPT-4-O cutoff date in November. We can measure the performance of the models on the benchmark using problems released after the cutoff date, thereby estimating the performance of the model on previously unseen problems. Figure 1 shows the performance of these models on LiveCodeBench code generation and test output prediction scenario on LeetCode problems released in different months from May 2023 and Feb 2024. We notice a stark drop in the performance of DS-Ins-33B model after Aug. 2023 (right before its release date), which suggests that the earlier problems might indeed be contaminated. This trend is consistent across other LiveCodeBench scenarios like repair and code execution, as depicted in Figure 10. Concurrently, Guo et al. (2024) (Section 4.1, last paragraph)

also acknowledge the possibility of LeetCode contamination, noting that "*models achieved higher scores in the LeetCode Contest held in July and August.* Similarly, performance of the GPT-4-O model drops on problems released since November (its official cutoff date).

Interestingly, we find that this drop in performance primarily occurs for the LeetCode problems only and that the model performance is relatively smooth across the months for problems from other platforms. Figure 11 shows a relatively stable performance for all models on AtCoder problems released over different periods, with the possible exception of May and June.

Performances of other models. We study performance variations in other models released recently. Particularly, GPT-4-Turbo, Gemini-Pro, Mistral-L, and Claude-3s models were released in November 2023, December 2023, February 2024, and March 2024 respectively. Note that GPT-4-Turbo (1106-preview variant) and Claude-3s have cutoff dates April 2023 and August 2023 respectively. Irrespective of the release or cutoff dates, we do not find any drastic performance variations across the months, as shown in Figure 12, particularly compared to the DeepSeek models. Interestingly, we find that even the DS-Base-33B model also suffers from contamination dropping from Pass@1 $\sim$ 60 in May problems to Pass@1 $\sim$ 0 in September LeetCode problems. This also suggests the likely inclusion of competition problems in the pretraining of the DeepSeek models, thereby affecting all instruction models trained from it. Finally, Codestral achieves Pass@1 36.5 on problems released between May'23 and Jan'24 and Pass@1 28.3 on problems since Feb'24.

5.2 Performance and Model Comparisons

We evaluate 34 instruction-tuned models (and 18 base models used in the code generation scenario) on LiveCodeBench. These models range from closed access to open access with their various fine-tuned variants. To overcome contamination issues in DeepSeek models, we only consider problems released since Sep 2023 for all evaluations below. Figure 4 shows the performance of a subset of models across the four scenarios. We highlight our key findings below.

Holistic Evaluations. We have evaluated the models across the four scenarios currently available in LiveCodeBench. Figure 2 displays the performance of models on all scenarios along the axes of the polar chart. First, we observe that the relative order of models remains mostly consistent across the scenarios. This is also supported by high correlations between Pass@1 metric across the scenarios – over 0.88 across all pairs as shown in Figure 13. Interestingly, the correlations are larger for related tasks, 0.98 for generation and self-repair, and 0.96 for test output prediction and code execution. This correlation drops to 0.89 for generation and execution scenarios.

However, despite the strong correlation, the relative differences in performance do vary across the scenarios. For example, GPT-4-Turbo further gains performance gap over GPT-4 in the self-repair scenario after already leading in the code generation scenario. Similarly, Claude-3-Opus and Mistral-L perform well in tasks involving COT, particularly in the code execution and test output prediction scenarios. For instance, Claude-3-Opus even outperforms GPT-4-Turbo in the test output prediction scenario. Similarly, Mistral-L outperforms Claude-3-Sonnet in both scenarios after trailing behind in code generation and repair scenarios. These differences highlight the need for holistic evaluations beyond measuring code generation capabilities.

Comparison to HumanEval. Next, we compare how code generation performance metrics translate between LiveCodeBench and HumanEval, the primary benchmark used for evaluating coding capabilities. Note that we use HumanEval+ version of HumanEval problems as it is more reliable with more exhaustive test cases. Figure 5 shows a scatter plot of Pass@1 on HumanEval+ versus

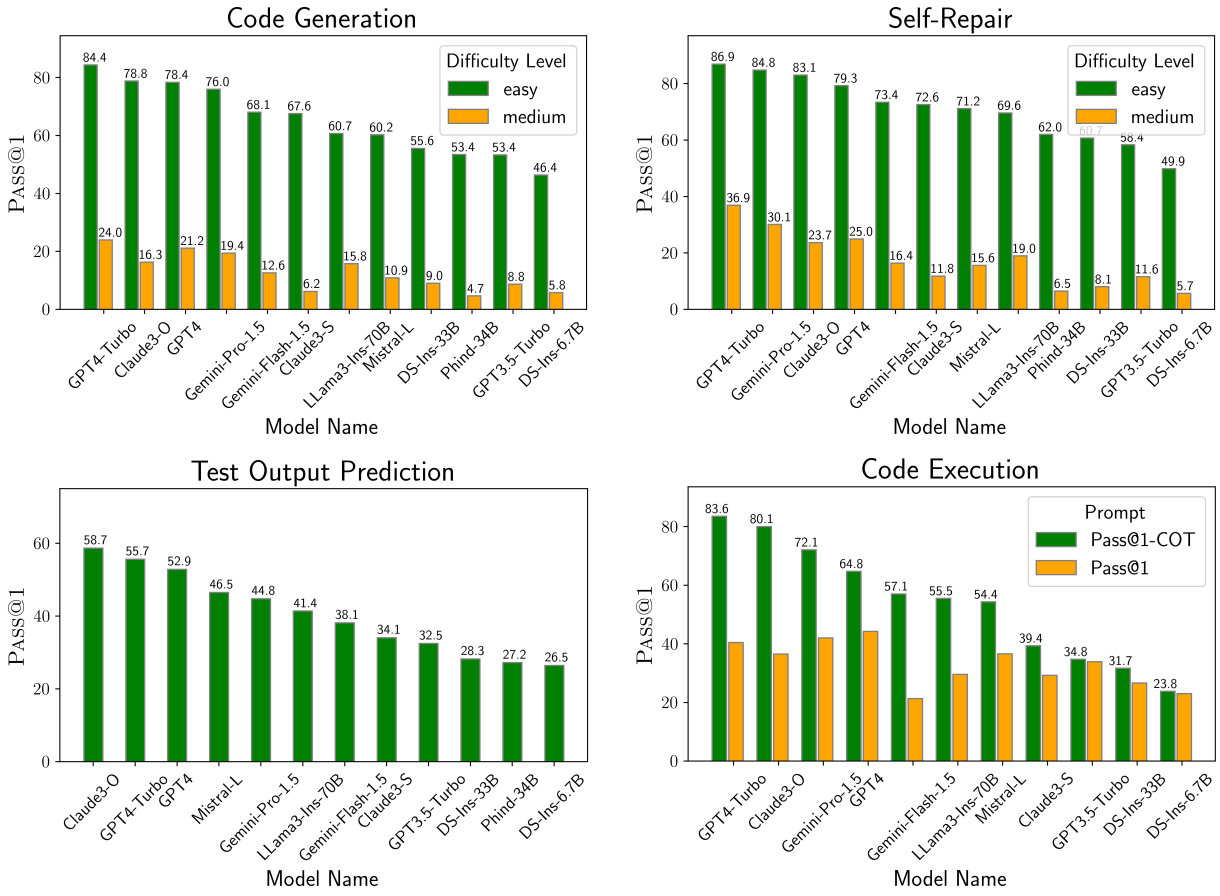

Figure 4: Model performances across the four scenarios available in LiveCodeBench (filtering on the time-window post September). The top-left and top-right plots depict Pass@1 of models on easy and medium splits across the code generation and self-repair scenarios respectively (results on hard subset deferred to the Appendix). The bottom-left and bottom-right plots depict Pass@1 of models across the test output prediction and code execution scenarios respectively.

LCB-Easy code generation scenario. We find only a moderate correlation of 0.72, with much larger performance variations on LCB-Easy.

Additionally, we observe that the models cluster into two groups, shaded in red and green. The models in the green-shaded region lie close to the $x = y$ line, indicating that they perform similarly on both benchmarks. On the other hand, the models shaded in red lie in the top-left region of the graph, indicating that they perform well only on HumanEval+ but not as well on LiveCodeBench. Interestingly, the green-shaded cluster contains base models or closed-access models, while the red-shaded cluster primarily comprises fine-tuned variants of open-access models. The well-separated clusters suggest that many models that perform well on HumanEval might be overfitting on the benchmark, and their performances do not translate well to problems from other domains or difficulty levels like those present in LiveCodeBench.

Indeed, HumanEval is an easier benchmark with small and isolated programming problems and thus easier to overfit on. In contrast, LiveCodeBench problems are sourced from reputable coding platforms offering more challenging problems with higher diversity and difficulty levels. This potential overfitting is particularly exemplified by DS-Ins-1.3B which achieves 59.8% Pass@1 on HumanEval+ but only 26.3% on LCB-Easy. Thus, while it boasts better performance compared

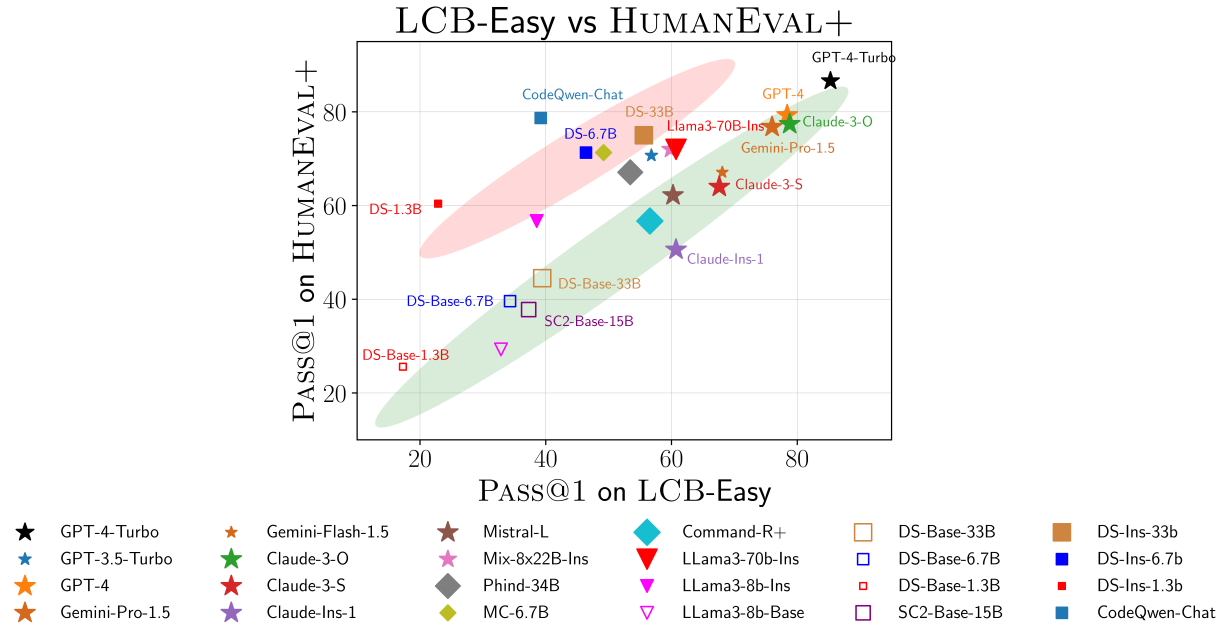

Figure 5: Scatter plot comparing Pass@1 of models on HumanEval+ versus Pass@1 on the easy subset of LiveCodeBench code generation scenario. Star markers denote the closed-access models while other markers denote different open model families. We find that the models are separated into two groups – the green-shaded region where performances on the two datasets are *aligned* and the red-shaded region where models perform well on HumanEval+ but perform poorly on LiveCodeBench. This indicates potential overfitting on HumanEval+ and primarily occurs in the fine-tuned variants of open-access models. For example, DS-Ins-1.3B which achieves Pass@1 of 60 and 26 on HumanEval+ and LCB-Easy subset. Thus, while it ranks above CMD-R+ on HumanEval+, it performs significantly worse on the LCB. Similarly, DS-Ins-6.7B and CodeQwen outperform Claude-3-Sonnet on HumanEval+ but are > 20 points behind on LCB-Easy.

with Gemini-Pro and Claude-Ins-1 on HumanEval+, it performs considerably worse on LCB-Easy. Similarly, CodeQwen, DS-Ins-6.7B, and MC-6.7B perform better than Mistral-L, Claude-2, and Claude-3-Sonnet on HumanEval+ but are considerably worse on LCB-Easy.

Highlighting the gap between SoTA and open models. One distinct observation from our evaluations is the large gap between SoTA models and open models across all scenarios. Particularly, GPT-4-Turbo, GPT-4, Gemini-Pro-1.5 and Claude-3-Opus lead across the benchmarks with wide performance margins over other models. This distinguishes LiveCodeBench from prior benchmarks (like HumanEval) where various open models have achieved similar or better performance. For example, DS-Ins-33B is merely 4.3 point behind GPT-4-Turbo on HumanEval+ but 16.2 points (69%) on LCB code generation scenario. This gap either holds or sometimes even amplifies across other scenarios. For instance, consider test output prediction and code execution (with COT) where GPT-4-Turbo leads the DS-Ins-33B model by 96% and 134% respectively!

We qualitatively analyze code samples generated by the leading model, GPT-4-Turbo, and find that it generates more readable code. Specifically, the code consists of more inline natural language comments that *reason* or *plan* before producing the code. We verify this quantitatively and find GPT-4-Turbo generated uses 19.5× more comment tokens compared to GPT-4.

Comparing Base Models. We use four families of base models – L3-Base, DeepSeek, CodeLLaMa, and StarCoder2 and compare them on the code generation scenario. A one-shot prompt is used for all models to avoid any formatting and answer extraction issues. We find L3-Base and DS models are significantly better than both CodeLLaMa and StarCoder2 base models with a DS-Base-6.7B model even outperforming both CL-Base-34B and SC2-Base-15B models. Next, we observe that SC2-Base-15B also outperforms the CL-Base-34B model (similiar to findings in Lozhkov et al. (2024)). Note that some LiveCodeBench specific differences can potentially be attributed to data curation approaches. For instance, StarCoder2 models (and potentially DeepSeeks as discussed in Section 5.1) use competition problems in the pre-training corpus.

Role of Post Training. We find that post-training improves performance on both HumanEval+ and LiveCodeBench for the code generation scenario. Particularly, L3-Ins-70B, DS-Ins-33B and Phind-34B achieve 28.3, 23.6, and 21 Pass@1 on LCB improving over their base models by 8.2, 7.3 and 9.5 points respectively. Similar gains are observed in previous benchmarks (like HumanEval+) as well. This highlights the importance of good post-training datasets for building strong LLMs.

At the same time, we note that the base models have *aligned* performances on LCB code generation and HumanEval+ benchmarks and lie within or close to the green shaded region in Figure 5. However, the fine-tuned open models exhibit a larger performance gap, with much better performances on HumanEval+. On the other hand, the closed-access models are still aligned across both benchmarks. This suggests that the fine-tuning data for open models might not be as diverse as that for closed models, leading to a lack of generalization to different kinds of problems.

Comparing open-access instruction-tuned models. Here, we compare various fine-tuned variants of the L3-Base, DeepSeek and CodeLLaMa base models across different model sizes. We find that fine-tuned L3-Base and DeepSeek models lead in performance, followed by Phind-34B and CodeLLaMa models across most scenarios. Broadly, we find that model performances correlate with model sizes. For example, Phind-34B model outperforms the 6.7B models across all scenarios.

Comparing Closed Models. We evaluate a range of closed (API access) models ranging from different model families like GPTs, Claudes, Gemini, and Mistral. We find the GPT-4-Turbo and Claude-3-Opus rank at the top across all scenarios followed by Mistral-L and Claude-3-Sonnet models. Finally, Gemini-Pro and GPT-3.5-turbo lie on the lower end of the models. The relative differences between the models vary across the scenarios. For example, GPT-4-Turbo demonstrates remarkable improvement from self-repair (24.5% to 36.9% on the LCB-Medium problems) while Gemini-Pro only improves from 8.5% to 9.4%. Similarly, as identified above, Claude-3-Opus and Mistral-L perform considerably better on test output prediction and code execution scenarios.

Open-Access vs Closed-Access Models. In general, closed (API) access model families generally outperform the open access models. The gap is only closed by three models, namely L3-Ins-70B, Mixtral, and DS-Ins-33B which reach the performance levels of the closed models. For instance, in the code generation scenario (Figure 2 right), these models reach close to or even outperform closed access models like Gemini-Pro, GPT-3.5-turbo, and Claude-3-Sonnet. The performances vary across scenarios with the closed-access models performing better in test output prediction and code execution scenarios. Overall, our findings confirm that a combination of strong base models and high-quality post-training datasets is a viable recipe for good code LLMs.

6 Related Work

6.1 Code Generation

Language Models for Code Generation. Starting with Codex (Chen et al., 2021), there are over a dozen code LLMs. These include CodeT5 (Wang et al., 2021, 2023), CodeGen (Nijkamp et al., 2022), SantaCoder (Allal et al., 2023), StarCoder (Li et al., 2023b), AlphaCode (Li et al., 2022), InCoder (Fried et al., 2022), and CodeGeeX (Zheng et al., 2023). As of May 2024, L3-Base and DeepSeek (Bi et al., 2024), StarCoder Lozhkov et al. (2024); Li et al. (2023b) and CodeLLaMa (Roziere et al., 2023) are the most popular open models. Many downstream models resulted from fine-tuning them on synthetically generated data, such as WizardCoder (Luo et al., 2023), MagiCoders (Wei et al., 2023b), and Phind-34B.

Code Generation Benchmarks. Many benchmarks have been proposed to compare and evaluate these models. These primarily focus on natural language to Python code generation: HumanEval (Chen et al., 2021), HumanEval+ (Liu et al., 2023b), APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), MBPP (Austin et al., 2021), L2CEval (Ni et al., 2023). Their variants have been proposed to cover more languages, (Wang et al., 2022a; Zheng et al., 2023; Cassano et al., 2022; Athiwaratkun et al., 2022). Many benchmarks have focused on code generation in APIs. Benchmarks like DS-1000 (Lai et al., 2023), ARCADE (Yin et al., 2022), NumpyEval (Zhang et al., 2023b), and PandasEval (Jain et al., 2022) focus on data science APIs. Other benchmarks measure using broader APIs or general software engineering tasks, such as JuICe (Agashe et al., 2019), APIBench (Patil et al., 2023), RepoBench (Liu et al., 2023c), ODEX (Wang et al., 2022b), SWE-Bench (Jimenez et al., 2023), GoogleCodeRepo (Shrivastava et al., 2023), RepoEval (Zhang et al., 2023a), and Cocomic-Data (Ding et al., 2022).

A few benchmarks specifically measure competitive programming, such as APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), CodeScope (Yan et al., 2023), xCodeEval (Khan et al., 2023), and LeetCode-Hard (Shinn et al., 2023), and TACO (Li et al., 2023c). Methods such as AlphaCode (Li et al., 2022), AlphaCode 2(Gemini Team et al., 2023), ALGO (Zhang et al., 2023d), Parsel (Zelikman et al., 2022), code cleaning (Jain et al., 2023), code explainations (Li et al., 2023a), analogical reasoning (Yasunaga et al., 2023), and AlphaCodium (Ridnik et al., 2024) have been pushing the boundaries of what is possible with LLMs in this domain. The biggest differentiating factor between LiveCodeBench and these benchmarks is that our benchmark is **continuously updated**, **problem curation with balanced difficulty**, **higher tests and problem quality**, and contains **more scenarios** such as code repair, code execution, and test output prediction capturing more facets for building agentic coding systems.

6.2 Holistic Tasks

LiveCodeBench considers self-repair, test output prediction, and code execution as additional scenarios. Below we note pertinent related work for these domains.

Code Repair. (Chen et al., 2023; Olausson et al., 2023; Madaan et al., 2023b; Peng et al., 2023; Zhang et al., 2023c) have investigated self-repair for existing code LLM benchmarks. Particularly, these methods use error feedback for models to improve inspiring our code repair scenario.

Code Execution. Code execution was first studied in Austin et al. (2021); Nye et al. (2021) LiveCodeBench's execution scenario is particularly inspired by CRUXEval (Gu et al., 2024), a

recent benchmark measuring the reasoning and execution abilities of code LLMs. We differ from CRUXEval in that our benchmark is live, and our functions are more complex and human-produced (unlike Code Llama generations in CRUXEval).

Test Generation. Test generation using LLMs has been explored in (Yuan et al., 2023; Schäfer et al., 2024; Tufano et al., 2022; Watson et al., 2020). Furthermore, Chen et al. (2022) demonstrated that LLMs can assist in generating test case inputs/outputs for competitive programming problems, thereby improving the accuracy of the generated code, thus inspiring our test generation scenario. However, LiveCodeBench's test generation scenario is unique in that it decouples the test inputs and outputs allowing more proper evaluations.

Finally, some works have additionally studied other tasks and scenarios like type prediction (Mir et al., 2022; Wei et al., 2023a; Malik et al., 2019), code summarization (LeClair et al., 2019; Iyer et al., 2016; Barone and Sennrich, 2017; Hasan et al., 2021; Alon et al., 2018), code security (Liguori et al., 2022; Pearce et al., 2022; Tony et al., 2023), etc.

6.3 Contamination

Data contamination and test-case leakage have received considerable attention Oren et al. (2024); Golchin and Surdeanu (2023); Weller et al. (2023); Roberts et al. (2024) as LLMs might be getting trained on benchmarks. Sainz et al. (2023) demonstrated contamination by simply prompting the model to highlight its contamination. Some detection methods have also been built to avoid these cases (Shi et al., 2023; Zhou et al., 2023). For code, Riddell et al. (2024) use edit distance and AST-based semantic-similarity to detect contamination.

7 Limitations

Benchmark Size. LiveCodeBench code generation scenario currently hosts over 400 instances from problems released between May and February. To account for contamination in DeepSeek, we only perform evaluations on problems released after the model cutoff date. This leads to only 349 problems used in our final evaluations which might add noise due to problem set samples. We currently estimate $1 - 1.5\%$ performance variations in LiveCodeBench code generation due to this issue (measured by bootstrapping 349 sized problem sets from the 511 sized dataset). Other scenarios, i.e. self-repair, code execution, and test output prediction comprise 349, 188, and 254 problems would have similar performance variations. We thus recommend exercising proper judgement when comparing models with small performance differences. Note that HumanEval has 164 problems and would also struggle with similar issues.

This issue is also exacerbated for newer models, with more recent cutoff dates, as they might only have access to a smaller evaluation set. We propose two solutions addressing this issue as we evolve LiveCodeBench. First, we will use other competition platforms for problem collection, allowing larger number of recent problems to be added to the benchmark. In addition, we also hope supplement this with an unreleased private test set constructed specifically for model evaluation. These problems will use a similar flavor to current problems and will be used when models are submitted for evaluation to the LiveCodeBench platform. This would reduce the reliance on public accessible problems and provide a more robust evaluation of the models while providing community public access to similar problems, similar to strategies employed by popular platforms like Kaggle.

Focus on Python. LiveCodeBench currently only focuses on Python which might not provide enough signal about model capabilities in other languages. However, since we collected problem statements and serialized tests, adding new programming languages would be straightforward once appropriate evaluation engines are used.

Robustness to Prompts. Recent works have identified huge performance variances that can be caused due to insufficient prompt. Here, we either do not tune prompts across models or make minor adjustments based on the system prompts and delimiter tokens. This can lead to performance variance in our results. Our findings and model comparison orders generalize across LiveCodeBench scenarios and mostly match the performance trends observed on HumanEval making this a less prominient issue.

This issue can be particularly observed open models on the code execution scenario with COT prompting. Interestingly, often the open models perform even worse in comparsion to the direct code execution baseline. Note that we used same prompts for the closed models all of which show noticable improvement from COT. While the used prompts might be sub-optimal, this highlights how open-models perform worse against the closed models at performing chain-of-thought.

Problem Domain. Programming is a vast domain and occurs in various forms such as programming puzzles, competition programming, and real-world software development. Different domains might have individual requirements, constraints, challenges, and difficulty levels. LiveCodeBench currently focuses on competition problems sourced from three platforms. This might not be representative of the "most general" notion of LLM programming capabilities. Particularly, real-world usage of LLMs is drawn upon open-ended and unconstrained problems rasied by users. We therefore recommend using LiveCodeBench as a starting point for evaluating LLMs and further using domain-specific evaluations to measure and compare LLMs in specific settings as required.

8 Conclusion

In this work, we propose LiveCodeBench, a new benchmark for evaluating LLMs for code. Our benchmark mitigates contamination issues in existing benchmarks by introducing live evaluations and emphasizing scenarios beyond code generation to account for the broader coding abilities of LLMs. LiveCodeBench is an extensible framework, that will keep on updating with new problems, scenarios, and models. Our evaluations reveal novel findings such as contamination detection and potential overfitting on HumanEval. We hope LiveCodeBench with serve to advance understanding of current code LLMs and also guide future research in this area through our findings.

Acknowledgements

This work was supported in part by NSF grants CCF:1900968, CCF:1908870 and by SKY Lab industrial sponsors and affiliates Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Nexla, Samsung SDS, Uber, and VMware. A. Gu is supported by the NSF Graduate Research Fellowship under Grant No. 2141064. A. Solar-Lezama is supported by the NSF and Intel Corporation through NSF Grant CCF:2217064. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position of the sponsors.

Finally, we thank Manish Shetty, Wei-Lin Chiang, Jierui Li, Horace He, Federico Cassano, Pengcheng

Yin, and Aman Madaan for helpful feedback at various stages of the work.

References

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. Juice: A large scale distantly supervised dataset for open domain context-based code generation. *arXiv preprint arXiv:1910.02216*. (Cited on pg. 15)

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*. (Cited on pg. 2, 15)

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*. (Cited on pg. 16)

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*. (Cited on pg. 15)

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*. (Cited on pg. 2, 4, 15)

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*. (Cited on pg. 16)

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*. (Cited on pg. 15)

Barry Boehm. 2006. A view of 20th and 21st century software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 12–29, New York, NY, USA. Association for Computing Machinery. (Cited on pg. 5)

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*. (Cited on pg. 15)

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*. (Cited on pg. 6, 16)

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*. (Cited on pg. 2, 9, 15)

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*. (Cited on pg. 5, 15)

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007*. (Cited on pg. 15)

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *preprint arXiv:2204.05999*. (Cited on pg. 15)

A Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*. (Cited on pg. 15)

Shahriar Golchin and Mihai Surdeanu. 2023. Time travel in llms: Tracing data contamination in large language models. *arXiv preprint arXiv:2308.08493*. (Cited on pg. 16)

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*. (Cited on pg. 6, 9, 15, 34)

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*. (Cited on pg. 2, 5, 10)

Masum Hasan, Tanveer Muttaqueen, Abdullah Al Ishtiaq, Kazi Sajeed Mehrab, Md Mahim Anjum Haque, Tahmid Hasan, Wasi Uddin Ahmad, Anindya Iqbal, and Rifat Shahriyar. 2021. Codesc: A large code-description parallel dataset. *arXiv preprint arXiv:2105.14220*. (Cited on pg. 16)

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*. (Cited on pg. 9, 10, 15, 26)

Yiming Huang, Zhenghao Lin, Xiao Liu, Yeyun Gong, Shuai Lu, Fangyu Lei, Yaobo Liang, Yelong Shen, Chen Lin, Nan Duan, et al. 2023. Competition-level problems are effective llm evaluators. *arXiv preprint arXiv:2312.02143*. (Cited on pg. 5)

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *54th Annual Meeting of the Association for Computational Linguistics 2016*, pages 2073–2083. Association for Computational Linguistics. (Cited on pg. 16)

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231. (Cited on pg. 15)

Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E Gonzalez, Koushik Sen, and Ion Stoica. 2023. Llm-assisted code cleaning for training accurate code generators. *arXiv preprint arXiv:2311.14904*. (Cited on pg. 15)

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*. (Cited on pg. 15)

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*. (Cited on pg. 15)

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32. (Cited on pg. 9)

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. (Cited on pg. 9)

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR. (Cited on pg. 15)

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE. (Cited on pg. 16)

Jierui Li, Szymon Tworkowski, Yingying Wu, and Raymond Mooney. 2023a. Explaining competitive-level programming solutions using llms. *arXiv preprint arXiv:2307.05337*. (Cited on pg. 15)

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023b. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*. (Cited on pg. 2, 15)

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023c. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*. (Cited on pg. 5, 15)

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023d. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*. (Cited on pg. 2)

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097. (Cited on pg. 2, 15)

Pietro Liguori, Erfan Al-Hossami, Domenico Cotroneo, Roberto Natella, Bojan Cukic, and Samira Shaikh. 2022. Can we generate shellcodes via natural language? an empirical study. *Automated Software Engineering*, 29(1):30. (Cited on pg. 16)

Changshu Liu, Shizhuo Dylan Zhang, and Reyhaneh Jabbarvand. 2024. Codemind: A framework to challenge large language models for code reasoning. *arXiv preprint arXiv:2402.09664*. (Cited on pg. 5)

Hanmeng Liu, Ruoxi Ning, Zhiyang Teng, Jian Liu, Qiji Zhou, and Yue Zhang. 2023a. Evaluating the logical reasoning ability of chatgpt and gpt-4. *arXiv preprint arXiv:2304.03439*. (Cited on pg. 4)

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*. (Cited on pg. 7, 15)

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023c. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*. (Cited on pg. 15)

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Eduardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. (Cited on pg. 2, 14, 15)

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, et al. 2024. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*. (Cited on pg. 2)

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*. (Cited on pg. 2, 15)

Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2023a. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*. (Cited on pg. 2)

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023b. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*. (Cited on pg. 15)

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. Nl2type: inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 304–315. IEEE. (Cited on pg. 16)

Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2241–2252. (Cited on pg. 16)

Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, et al. 2023. L2ceval: Evaluating language-to-code generation capabilities of large language models. *arXiv preprint arXiv:2309.17446*. (Cited on pg. 15)

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*. (Cited on pg. 15)

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*. (Cited on pg. 15)

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying gpt self-repair for code generation. *arXiv preprint arXiv:2306.09896*. (Cited on pg. 2, 5, 10, 15)

R OpenAI. 2023. Gpt-4 technical report. arxiv 2303.08774. *View in Article*. (Cited on pg. 7)

Yonatan Oren, Nicole Meister, Niladri Chatterji, Faisal Ladhak, and Tatsunori B Hashimoto. 2024. Proving test set contamination for black-box language models. In *The Twelfth International Conference on Learning Representations*. (Cited on pg. 16)

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*. (Cited on pg. 2, 15)

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE. (Cited on pg. 16)

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, and Jianfeng Gao. 2023. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*. (Cited on pg. 15)

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*. (Cited on pg. 2)

Martin Riddell, Ansong Ni, and Arman Cohan. 2024. Quantifying contamination in evaluating code generation capabilities of language models. (Cited on pg. 16)

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*. (Cited on pg. 2, 5, 15)

Manley Roberts, Himanshu Thakur, Christine Herlihy, Colin White, and Samuel Dooley. 2024. To the cutoff... and beyond? a longitudinal perspective on LLM data contamination. In *The Twelfth International Conference on Learning Representations*. (Cited on pg. 16)

Michael Royzen, Justin Wei, and Russell Coleman. 2023. Phind. (Cited on pg. 2)

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*. (Cited on pg. 2, 15)

Oscar Sainz, Jon Ander Campos, Iker García-Ferrero, Julen Etxaniz, and Eneko Agirre. 2023. Did chatgpt cheat on your test? (Cited on pg. 16)

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105. (Cited on pg. 16)

Weijia Shi, Anirudh Ajith, Mengzhou Xia, Yangsibo Huang, Daogao Liu, Terra Blevins, Danqi Chen, and Luke Zettlemoyer. 2023. Detecting pretraining data from large language models. *arXiv preprint arXiv:2310.16789*. (Cited on pg. 16)

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*. (Cited on pg. 5, 15)

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pages 31693–31715. PMLR. (Cited on pg. 15)

Manav Singhal, Tushar Aggarwal, Abhijeet Awasthi, Nagarajan Natarajan, and Aditya Kanade. 2024. Nofuneval: Funny how code lms falter on requirements beyond functional correctness. *arXiv preprint arXiv:2401.15963*. (Cited on pg. 5)

Benjamin Steenhoek, Michele Tufano, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Reinforcement learning from automatic feedback for high-quality unit test generation. *arXiv preprint arXiv:2310.02368*. (Cited on pg. 2)

Ruoxi Sun, Sercan O Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2023. Sql-palm: Improved large language modeladaptation for text-to-sql. *arXiv preprint arXiv:2306.00739*. (Cited on pg. 2)

Gemini Team. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. (Cited on pg. 2)

Catherine Tony, Markus Mutas, Nicolás E Díaz Ferreyra, and Riccardo Scandariato. 2023. Llmseceval: A dataset of natural language prompts for security evaluations. *arXiv preprint arXiv:2303.09384*. (Cited on pg. 16)

Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. 2022. Methods2test: A dataset of focal methods mapped to test cases. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 299–303. (Cited on pg. 16)

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2022a. Recode: Robustness evaluation of code generation models. *arXiv preprint arXiv:2212.10264*. (Cited on pg. 15)

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*. (Cited on pg. 15)

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708. (Cited on pg. 15)

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022b. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481*. (Cited on pg. 15)

Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1398–1409. (Cited on pg. 16)

Jiayi Wei, Greg Durrett, and Isil Dillig. 2023a. Typet5: Seq2seq type inference using static analysis. *arXiv preprint arXiv:2303.09564*. (Cited on pg. 16)

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023b. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*. (Cited on pg. 2, 15)

Orion Weller, Marc Marone, Nathaniel Weir, Dawn Lawrie, Daniel Khashabi, and Benjamin Van Durme. 2023. " according to..." prompting language models improves quoting from pre-training data. *arXiv preprint arXiv:2305.13252*. (Cited on pg. 16)

Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Shuiguang Deng, et al. 2023. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. *arXiv preprint arXiv:2311.08588*. (Cited on pg. 15)

Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E. Gonzalez, and Ion Stoica. 2023. Rethinking benchmark and contamination for language models with rephrased samples. (Cited on pg. 2)

Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H Chi, and Denny Zhou. 2023. Large language models as analogical reasoners. *arXiv preprint arXiv:2310.01714*. (Cited on pg. 15)

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, et al. 2022. Natural language to code generation in interactive data science notebooks. *arXiv preprint arXiv:2212.09248*. (Cited on pg. 15)

Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*. (Cited on pg. 16)

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D Goodman, and Nick Haber. 2022. Parsel: A unified natural language framework for algorithmic reasoning. *arXiv preprint arXiv:2212.10561*. (Cited on pg. 15)

Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*. (Cited on pg. 15)

Kechi Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023b. Toolcoder: Teach code generation models to use apis with search tools. *arXiv preprint arXiv:2305.04032*. (Cited on pg. 15)

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023c. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787, Toronto, Canada. Association for Computational Linguistics. (Cited on pg. 15)

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023d. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *arXiv preprint arXiv:2305.14591*. (Cited on pg. 15)

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*. (Cited on pg. 15)

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *https://arxiv.org/abs/2402.14658*. (Cited on pg. 2)

Maosheng Zhong, Gen Liu, Hongwei Li, Jiangling Kuang, Jinshan Zeng, and Mingwen Wang. 2022. Codegen-test: An automatic code generation model integrating program test information. *arXiv preprint arXiv:2202.07612*. (Cited on pg. 2)

Kun Zhou, Yutao Zhu, Zhipeng Chen, Wentong Chen, Wayne Xin Zhao, Xu Chen, Yankai Lin, Ji-Rong Wen, and Jiawei Han. 2023. Don't make your llm an evaluation benchmark cheater. *arXiv preprint arXiv:2311.01964*. (Cited on pg. 16)

A Dataset

A.1 License

Similar to Hendrycks et al. (2021), we scrape only the problem statements, ground-truth solutions, and test cases from competition websites – LeetCode, AtCoder, and CodeForces. Further, we only scrape publicly visible portions of websites, avoiding any data collection that might be paywalled or require login or interaction with the website. Following, Hendrycks et al. (2021) we abide by Fair Use §107: "the fair use of a copyrighted work, including such use by ... scholarship, or research, is not an infringement of copyright", where fair use is determined by "the purpose and character of the use, including whether such use is of a commercial nature or is for nonprofit educational purposes", "the amount and substantiality of the portion used in relation to the copyrighted work as a whole", and "the effect of the use upon the potential market for or value of the copyrighted work." Finally, we use the collected problems for academic purposes only and in addition, do not train on the collected problems.

A.2 Generator Based Test Generation

We use GPT-4-Turbo to construct input generators. The following prompts (Figures 6 and 7) provide one-shot prompt templates used for synthesizing random and adversarial input generators. These generators define a function returns the arguments sampled in some distribution. These generators are then executed to construct inputs which validated on the collected correct programs. We use separate generators for random and adversarial setting since often times programming problems have corner cases which might not be captured by randomly sampling over the inputs. We build 2 random input generators, 4 adversarial input generators and check if the sampled inputs work for the correct programs. Finally, the number of collected inputs is thresholded to 100 for efficient grading (using random selection). We find that our generators can already function well but future work can study the design space of constructing such generators.

Note that for CodeForces, we construct the generators in semi-autonomous manner since only 9 problems were used.

A.3 Code Execution

The code execution split of LiveCodeBench consists of 479 samples from 85 distinct problems. To encourage diversity in our benchmark while keeping our benchmark small and usable, we place a limit of six samples for each given problem. These sample programs and corresponding test cases are chosen uniformly at random from all those passing the filter.

Filtering Criteria: The specific filtering criteria are as follows:

- Compile time: length of code is between 100 and 500 characters, no syntax errors, all necessary imports are included
- Runtime: no floating point operations, true division, exp, other integer operations must have at least one argument ≤ 3, string and list operations must have at least one argument with length ≤ 3, finish running in 2 seconds, "reasonable" number of steps (roughly, under 1000 Python bytecode operations).

```
You are an expert python competitive programmar and your goal is to construct input-
    ↪ generators for testing programming contest problems. You will write relevant
    ↪ generators and finally construct ‘construct_inputs‘ function that returns a list
    ↪  of diverse inputs sampled from the generator. Remember to strictly follow the
    ↪ instructions and constraints present in the problem statement

<USER>
{EXAMPLE_PROBLEM}

Construct a random input generator.
</USER>

<ASSISTANT>
import numpy as np
def random_input_generator(weight_min, weight_max, size_min, size_max):
    weights_size = np.random.randint(size_min, size_max+1)
    weights = np.random.randint(weight_min, weight_max, size=weights_size).tolist()
    k = np.random.randint(1, len(weights)+1)
    return weights, k

def construct_inputs():
    inputs_list = []
    ## small inputs
    for i in range(15):
        inputs_list.append(random_input_generator(1, 10**3, 1, 10))
    ## medium inputs
    for i in range(15):
        inputs_list.append(random_input_generator(1, 10**6, 1, 10**3))
    ## large inputs
    for i in range(15):
        inputs_list.append(random_input_generator(1, 10**9, 1, 10**5))
    return inputs_list
</ASSISTANT>

<USER>
{PROBLEM}

Construct a random input generator. Use the format used in the above example by
    ↪ returning a single function that builds diverse inputs named ‘construct_inputs‘
</USER>
```

Prompt for random input generation

Figure 6: Random Input Generator Prompt

We give two examples of two programs that are filtered out in the Listings below. Our final benchmark consists of 479 samples from 85 problems, but will increase in size due to its live nature.

Dataset Statistics: As mentioned, we filter for codes between 100 and 500 characters, as well as below 1000 steps. The statistics for programs in our dataset are shown in Fig. 8.

```
You are an expert python competitive programmar and your goal is to construct input-
    ↪ generators for testing programming contest problems. You will write relevant
    ↪ generators and finally construct ‘construct_inputs‘ function that returns a list
    ↪  of diverse inputs sampled from the generator. Remember to strictly follow the
    ↪ instructions and constraints present in the problem statement

<USER>
{EXAMPLE_PROBLEM}

Construct an adversarial input generator.
</USER>

<ASSISTANT>
import numpy as np
## case 1 - alternating large and small weights
def generate_adversarial_inputs_1(weight_size, max_weight, k):
    weights = [1 if i%2==0 else max_weight for i in range(weight_size)]
    return weights, k

## case 2 - equal_weights
def adversarial_input_generator_2(weight_size, max_weight, k):
    weights = [max_weight for _ in range(weight_size)]
    return weights, k

# Case 3 - Large weights at the ends
def adversarial_input_generator_3(weight_size, max_weight, k):
    pass ## truncatede
def construct_inputs():
    inputs_list = []

    weight_sizes = [10, 1000, 100000]
    max_weights = [10**3, 10**6, 10**9]

    for weight_size in weight_sizes:
        for max_weight in max_weights:
            ks = [1, 2, 5, weight_size//2, weight_size-1, weight_size]
            for k in ks:
                inputs_list.append(generate_adversarial_inputs_1(weight_size,
    ↪ max_weight, k))
                # truncated
    return inputs_list
</ASSISTANT>

<USER>
{PROBLEM}

Construct an adversarial input generator. Use the format used in the above example by
    ↪ returning a single function that builds diverse inputs named ‘construct_inputs‘
</USER>
```

Prompt for adversarial input generation

Figure 7: Adversarial Input Generator Prompt

```
def check(x, t):
    if x == '':
        return t == 0
    if t < 0:
        return False
    for i in range(len(x)):
        if check(x[:i], t - int(x[i:])):
            return True
    return False

@cache
def punishmentNumber(n: int) -> int:
    if n == 0:
        return 0
    ans = punishmentNumber(n-1)
    if check(str(n * n), n):
        ans += n * n
    return ans
assert punishmentNumber(n = 37) == 1478
```

Program filtered because of multiplication

```
dp = [True for _ in range(int(1e6 + 5))]
MAXN = int(1e6 + 5)
p = []
dp[0] = False
dp[1] = False
for i in range(2, MAXN):
    if not dp[i]: continue
    p.append(i)
    for j in range(2 * i, MAXN, i):
        dp[j] = False
def findPrimePairs(n: int) -> List[List[int]]:
    res = []
    for i in range(1, n):
        if n % 2 == 1 and i > n//2: break
        if n % 2 == 0 and i > n//2: break
        if dp[i] and dp[n - i]:
            res.append([i, n - i])
    return res
assert findPrimePairs(n = 2) == []
```

Program filtered because of control flow

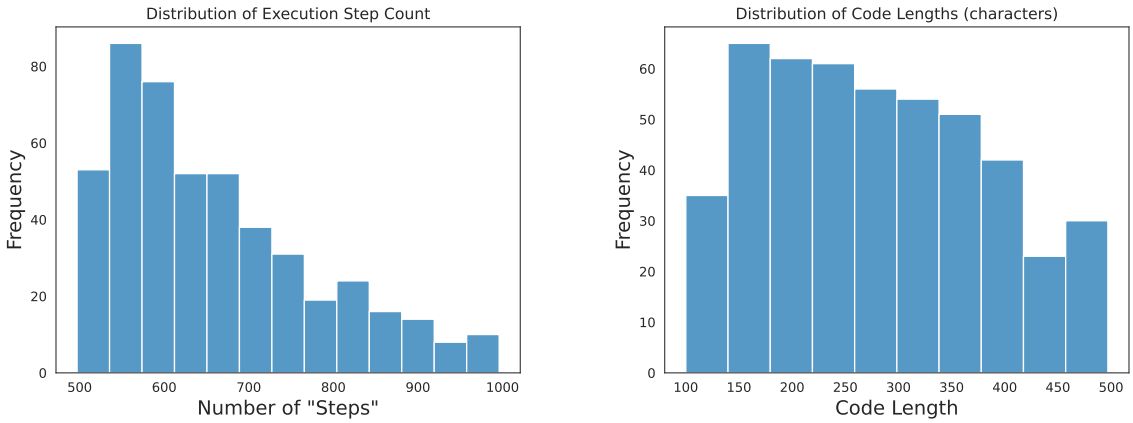

Figure 8: Distribution of code lengths and number of execution steps

B UI

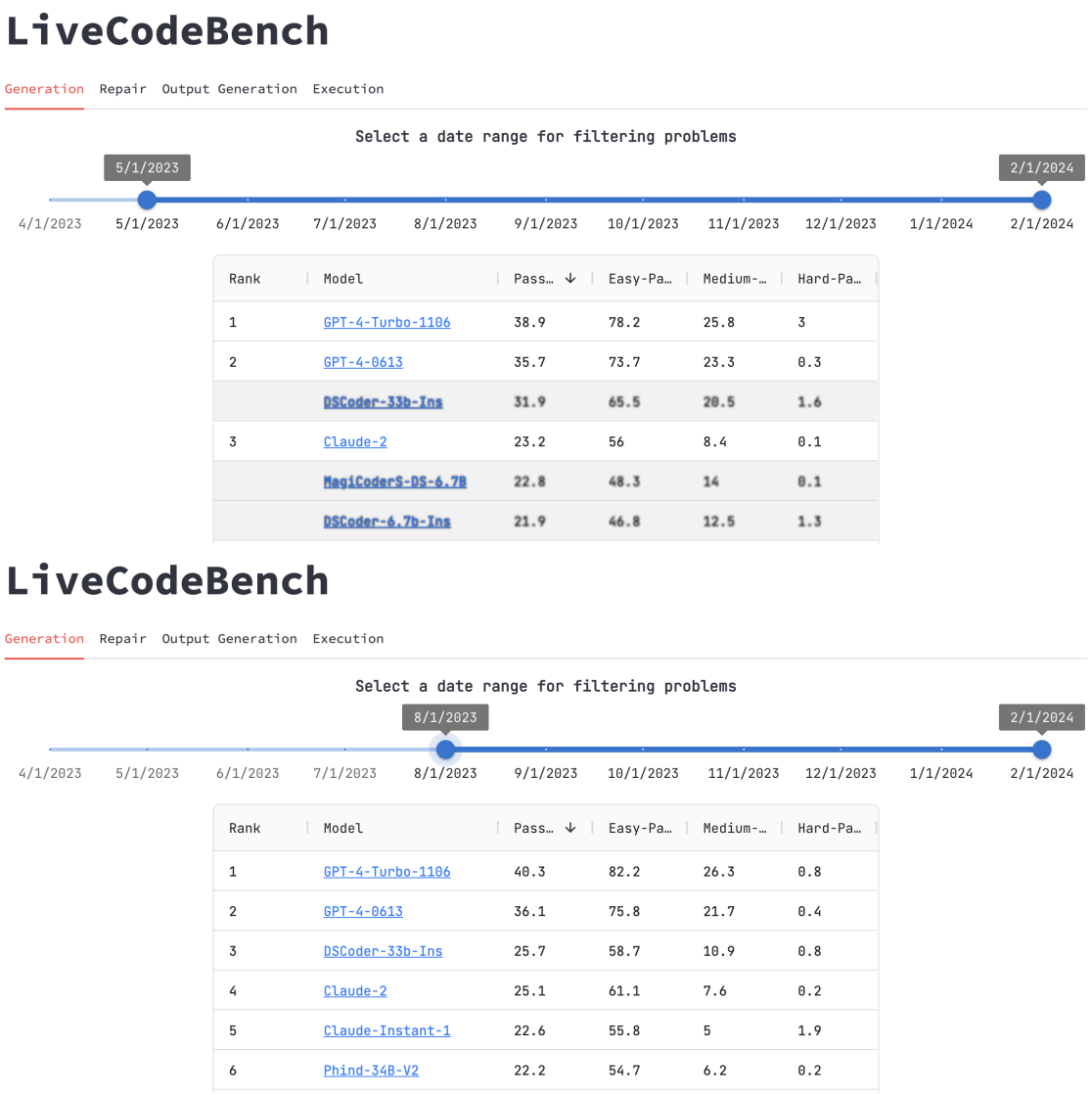

Figure 9: UI of LiveCodeBench showing two views – May-Jan and Sep-Jan. The contaminated models are blurred and the performance difference is visible across the two views. The scroller on the top allows selecting different periods of time highlighting the live nature of the benchmark.

C Experimental Setup

C.1 Models

We describe the details of models considered in our study in Table 2.

Model ID	Short Name	Approximate Cutoff Date	Link
deepseek-ai/deepseek-coder-33b-instruct	DSCoder-33b-Ins	08/30/2023	deepseek-coder-33b-instruct
deepseek-ai/deepseek-coder-6.7b-instruct	DSCoder-6.7b-Ins	08/30/2023	deepseek-coder-6.7b-instruct
deepseek-ai/deepseek-coder-1.3b-instruct	DSCoder-1.3b-Ins	08/30/2023	deepseek-coder-1.3b-instruct
codellama/CodeLlama-70b-Instruct-hf	CodeLlama-70b-Ins	01/01/2023	CodeLlama-70b-Instruct-hf
openbmb/Eurus-70b-sft	Eurus-70B-SFT (n=1)	01/01/2023	Eurus-70b-sft
openbmb/Eurux-8x22b-nca	Eurux-8x22b-NCA (n=1)	04/30/2023	Eurux-8x22b-nca
codellama/CodeLlama-34b-Instruct-hf	CodeLlama-34b-Ins	01/01/2023	CodeLlama-34b-Instruct-hf
codellama/CodeLlama-13b-Instruct-hf	CodeLlama-13b-Ins	01/01/2023	CodeLlama-13b-Instruct-hf
codellama/CodeLlama-7b-Instruct-hf	CodeLlama-7b-Ins	01/01/2023	CodeLlama-7b-Instruct-hf
meta-llama/Meta-Llama-3-8B-Instruct	LLama3-8b-Ins	01/01/2023	Meta-Llama-3-8B-Instruct
meta-llama/Meta-Llama-3-70B-Instruct	LLama3-70b-Ins	01/01/2023	Meta-Llama-3-70B-Instruct
Phind/Phind-CodeLlama-34B-v2	Phind-34B-V2	01/01/2023	Phind-CodeLlama-34B-v2
Smaug-2-72B	Smaug-2-72B	01/01/2023	Smaug-2-72B
Qwen-1.5-72B-Chat	Qwen-1.5-72B-Chat	01/01/2023	Qwen-1.5-72B-Chat
Qwen/CodeQwen1.5-7B	CodeQwen15-7B	08/30/2023	CodeQwen1.5-7B
Qwen/CodeQwen1.5-7B-Chat	CodeQwen15-7B-Chat	08/30/2023	CodeQwen1.5-7B-Chat
gpt-3.5-turbo-0301	GPT-3.5-Turbo-0301	10/01/2021	gpt-3.5-turbo-0301
gpt-3.5-turbo-0125	GPT-3.5-Turbo-0125	10/01/2021	gpt-3.5-turbo-0125
gpt-4-0613	GPT-4-0613	10/01/2021	gpt-4-0613
gpt-4-1106-preview	GPT-4-Turbo-1106	04/30/2023	gpt-4-1106-preview
gpt-4-turbo-2024-04-09	GPT-4-Turbo-2024-04-09	04/30/2023	gpt-4-turbo-2024-04-09
gpt-4o-2024-05-13	GPT-4O-2024-05-13	10/30/2023	gpt-4o-2024-05-13
claude-2	Claude-2	12/31/2022	claude-2
claude-instant-1	Claude-Instant-1	12/31/2022	claude-instant-1
claude-3-opus-20240229	Claude-3-Opus	04/30/2023	claude-3-opus-20240229
claude-3-sonnet-20240229	Claude-3-Sonnet	04/30/2023	claude-3-sonnet-20240229

claude-3-haiku-20240307	Claude-3-Haiku	04/30/2023	claude-3-haiku-20240307
codestral-latest	Codestral-Latest	01/31/2024	codestral-latest
gemini-pro	Gemini-Pro	04/30/2023	gemini-pro
gemini-1.5-pro-latest	Gemini-Pro-1.5-May	04/30/2023	gemini-1.5-pro-latest
gemini-1.5-flash-latest	Gemini-Flash-1.5-May	04/30/2023	gemini-1.5-flash-latest
ise-uiuc/Magicoder-S-DS-6.7B	MagiCoderS-DS-6.7B	08/30/2023	Magicoder-S-DS-6.7B
ise-uiuc/Magicoder-S-CL-7B	MagiCoderS-CL-7B	01/01/2023	Magicoder-S-CL-7B
bigcode/starcoder2-3b	StarCoder2-3b	01/01/2023	starcoder2-3b
bigcode/starcoder2-7b	StarCoder2-7b	01/01/2023	starcoder2-7b
bigcode/starcoder2-15b	StarCoder2-15b	01/01/2023	starcoder2-15b
codellama/CodeLlama-70b-hf	CodeLlama-70b-Base	01/01/2023	CodeLlama-70b-hf
codellama/CodeLlama-34b-hf	CodeLlama-34b-Base	01/01/2023	CodeLlama-34b-hf
codellama/CodeLlama-13b-hf	CodeLlama-13b-Base	01/01/2023	CodeLlama-13b-hf
codellama/CodeLlama-7b-hf	CodeLlama-7b-Base	01/01/2023	CodeLlama-7b-hf
deepseek-ai/deepseek-coder-33b-base	DSCoder-33b-Base	08/30/2023	deepseek-coder-33b-base
deepseek-ai/deepseek-coder-6.7b-base	DSCoder-6.7b-Base	08/30/2023	deepseek-coder-6.7b-base
deepseek-ai/deepseek-coder-1.3b-base	DSCoder-1.3b-Base	08/30/2023	deepseek-coder-1.3b-base
google/codegemma-7b	CodeGemma-7b-Base	01/01/2023	codegemma-7b
google/codegemma-2b	CodeGemma-2b-Base	01/01/2023	codegemma-2b
google/gemma-7b	Gemma-7b-Base	01/01/2023	gemma-7b
google/gemma-2b	Gemma-2b-Base	01/01/2023	gemma-2b
meta-llama/Meta-Llama-3-70B	LLama3-70b-Base	01/01/2023	Meta-Llama-3-70B
meta-llama/Meta-Llama-3-8B	LLama3-8b-Base	01/01/2023	Meta-Llama-3-8B
mistral-large-latest	Mistral-Large	01/01/2023	mistral-large-latest
open-mixtral-8x22b	Mixtral-8x22B-Ins	01/01/2023	open-mixtral-8x22b
open-mixtral-8x7b	Mixtral-8x7B-Ins	01/01/2023	open-mixtral-8x7b
m-a-p/OpenCodeInterpreter-DS-33B	OC-DS-33B	08/30/2023	OpenCodeInterpreter-DS-33B
m-a-p/OpenCodeInterpreter-DS-6.7B	OC-DS-6.7B	08/30/2023	OpenCodeInterpreter-DS-6.7B
m-a-p/OpenCodeInterpreter-DS-1.3B	OC-DS-1.3B	08/30/2023	OpenCodeInterpreter-DS-1.3B
command-r	Command-R	01/01/2023	command-r
command-r+	Command-R+	01/01/2023	command-r+

Table 2: Language Models Overview

C.2 Code Generation

Below we provide the prompt format (with appropriate variants adding special tokens accommodating each instruct-tuned model) used for this scenario.

```
You are an expert Python programmer. You will be given a question (problem
    ↪ specification) and will generate a correct Python program that matches the
    ↪ specification and passes all tests. You will NOT return anything except for the
    ↪ program

### Question:\n{question.question_content}


{ if question.starter_code }
 ### Format: {PromptConstants.FORMATTING_MESSAGE}

‘‘‘python
{question.starter_code}
‘‘‘
{ else }
### Format: {PromptConstants.FORMATTING_WITHOUT_STARTER_MESSAGE}

‘‘‘python
# YOUR CODE HERE
‘‘‘
{ endif }


### Answer: (use the provided format with backticks)
```

Code Generation Prompt

C.3 Self Repair

Below we provide the prompt format (with appropriate variants adding special tokens accommodating each instruct-tuned model) used for this scenario.

C.4 Code Execution

Below we provide the prompts for code execution with and without CoT. The prompts are modified versions of those from (Gu et al., 2024) to fit the format of the samples in our benchmark.

C.5 Test Output Prediction

Below we provide the prompt format (with appropriate variants adding special tokens accommodating each instruct-tuned model) used for this scenario.

```
{if check_result.result_status is "Wrong Answer"}
The above code is incorrect and does not pass the testcase.
Input: {wrong_testcase_input}
Output: {wrong_testcase_output}
Expected: {wrong_testcase_expected}


{elif check_result.result_status is "Time Limit Exceeded"}
The above code is incorrect and exceeds the time limit.
Input: {wrong_testcase_input}


{elif check_result.result_status is "Runtime Error"}
The above code is incorrect and has a runtime error.
Input: {wrong_testcase_input}
Error Message: {wrong_testcase_error_message}

{endif}
```

Self Repair Error Feedback Pseudocode

```
You are a helpful programming assistant and an expert Python programmer. You are
    ↪ helping a user write a program to solve a problem. The user has written some
    ↪ code, but it has some errors and is not passing the tests. You will help the
    ↪ user by first giving a concise (at most 2-3 sentences) textual explanation of
    ↪ what is wrong with the code. After you have pointed out what is wrong with the
    ↪ code, you will then generate a fixed version of the program. You must put the
    ↪ entired fixed program within code delimiters only for once.

### Question:\n{question.question_content}

### Answer: ```python
{code.code_to_be_corrected}
```

Format: {PromptConstants.FORMATTING_CHECK_ERROR_MESSAGE}

Answer: (use the provided format with backticks)
```

Self-Repair Prompt
```

```
You are given a Python function and an assertion containing an input to the function.
    ↪ Complete the assertion with a literal (no unsimplified expressions, no function
    ↪ calls) containing the output when executing the provided code on the given input
    ↪ , even if the function is incorrect or incomplete. Do NOT output any extra
    ↪ information. Provide the full assertion with the correct output in [ANSWER] and
    ↪ [/ANSWER] tags, following the examples.

[PYTHON]
def repeatNumber(number : int) -> int:
    return number
assert repeatNumber(number = 17) == ??
[/PYTHON]
[ANSWER]
assert repeatNumber(number = 17) == 17
[/ANSWER]

[PYTHON]
def addCharacterA(string : str) -> str:
    return string + "a"
assert addCharacterA(string = "x9j") == ??
[/PYTHON]
[ANSWER]
assert addCharacterA(string = "x9j") == "x9ja"
[/ANSWER]

[PYTHON]
{code}
assert {input} == ??
[/PYTHON]
[ANSWER]
```

Code Execution Prompt

```
You are given a Python function and an assertion containing an input to the function.
    ↪ Complete the assertion with a literal (no unsimplified expressions, no function
    ↪ calls) containing the output when executing the provided code on the given input
    ↪ , even if the function is incorrect or incomplete. Do NOT output any extra
    ↪ information. Execute the program step by step before arriving at an answer, and
    ↪ provide the full assertion with the correct output in [ANSWER] and [/ANSWER]
    ↪ tags, following the examples.

[PYTHON]
def performOperation(s):
    s = s + s
    return "b" + s + "a"
assert performOperation(s = "hi") == ??
[/PYTHON]
[THOUGHT]
Let's execute the code step by step:

1. The function performOperation is defined, which takes a single argument s.
2. The function is called with the argument "hi", so within the function, s is
    ↪ initially "hi".
3. Inside the function, s is concatenated with itself, so s becomes "hihi".
4. The function then returns a new string that starts with "b", followed by the value
    ↪ of s (which is now "hihi"), and ends with "a".
5. The return value of the function is therefore "bhihia".
[/THOUGHT]
[ANSWER]
assert performOperation(s = "hi") == "bhihia"
[/ANSWER]

[PYTHON]
{code}
assert {input} == ??
[/PYTHON]
[THOUGHT]
```

Code Execution Prompt with CoT

````
### Instruction: You are a helpful programming assistant and an expert Python
    ↪ programmer. You are helping a user to write a test case to help to check the
    ↪ correctness of the function. The user has written a input for the testcase. You
    ↪ will calculate the output of the testcase and write the whole assertion
    ↪ statement in the markdown code block with the correct output.

Problem:
{problem_statement}

Function:
```
{function_signature}
```
Please complete the following test case:

```
assert {function_name}({testcase_input}) == # TODO
```
### Response:
````

Test Output Prediction Prompt

D Results

D.1 Contamination

Figure 10 demonstrates contamination in DeepSeek in self repair and test output prediction scenarios.

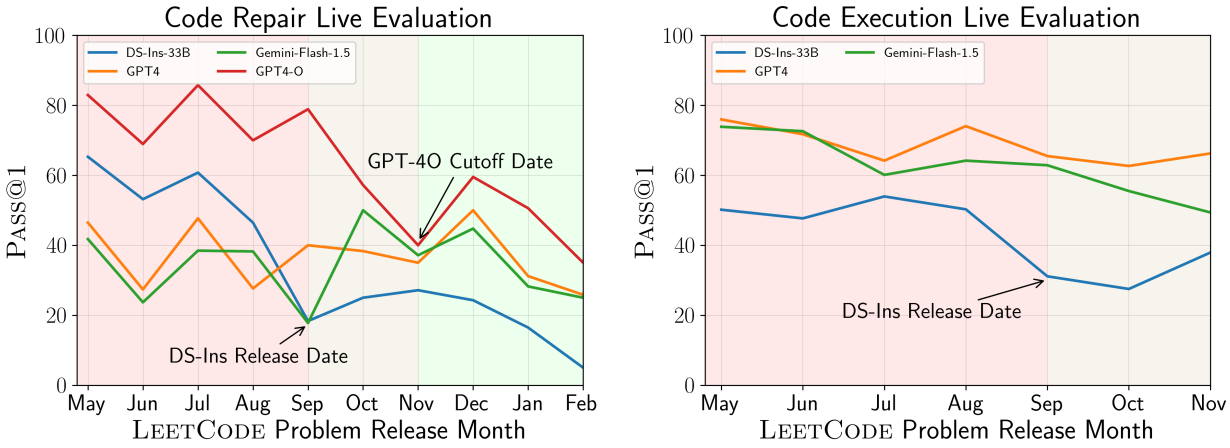

Figure 10: Contamination in DS models across self-repair and code execution (without COT) scenarios over time. Note that code execution currently runs between May and November

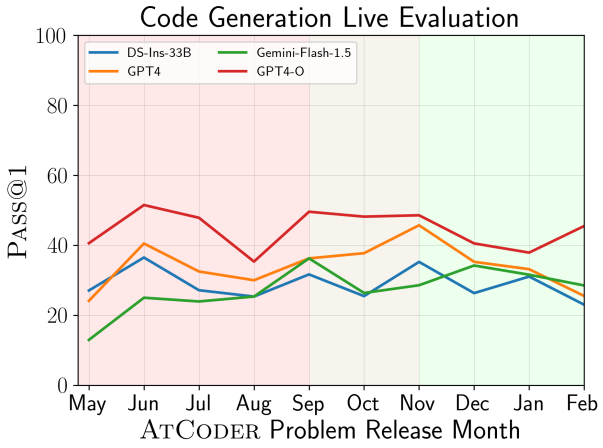

Figure 11: Performance on problems released over different months for AtCoder

D.2 All Results

Below we provide the tables comprising of results across different LiveCodeBench scenarios.

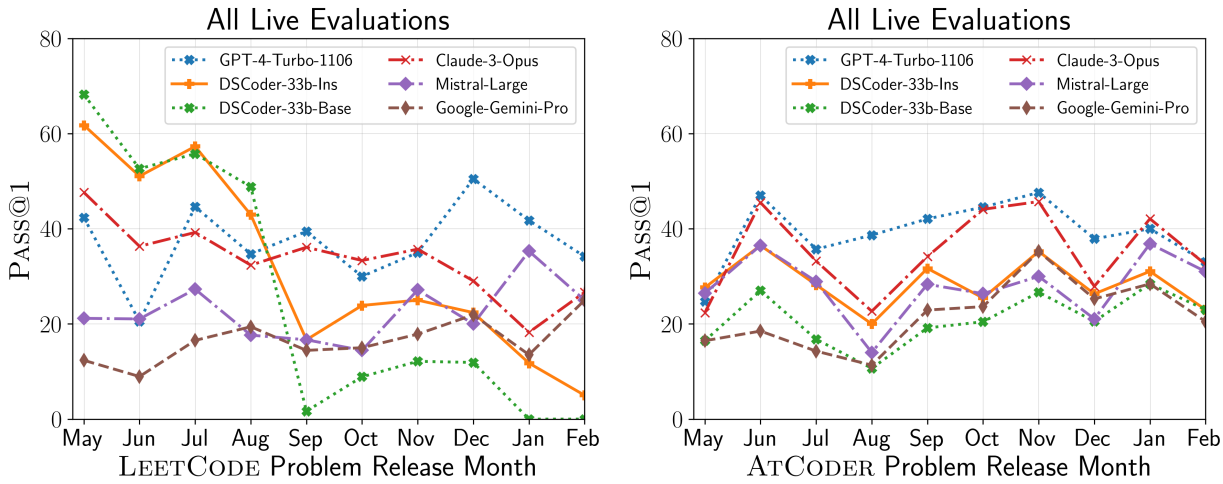

Figure 12: Live evaluation over time for various models on code generation scenario in LiveCodeBench. We consider many recently released models and do not find significant performance variations across months except for DS models.

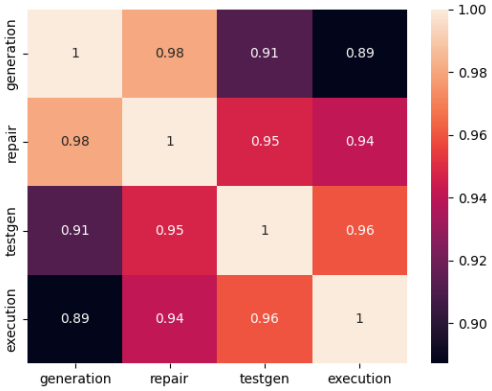

Figure 13: Correlations across different scenarios studied in LiveCodeBench

Model Name	Easy	Medium	Hard	Total
Claude-2	61.80	4.90	0.20	22.30
Claude-3-Haiku	63.00	4.30	1.10	22.80
Claude-3-Opus	78.80	16.30	3.20	32.80
Claude-3-Sonnet	67.60	6.20	1.10	25.00
Claude-Instant-1	60.70	4.30	1.10	22.10
CodeGemma-2b-Base	18.30	0.40	0.00	6.30
CodeGemma-7b-Base	35.70	2.60	0.10	12.80
CodeLlama-13b-Base	24.60	0.90	0.00	8.50
CodeLlama-13b-Ins	36.60	2.40	0.00	13.00
CodeLlama-34b-Base	32.20	1.80	0.10	11.40
CodeLlama-34b-Ins	33.70	2.40	1.10	12.40
CodeLlama-70b-Base	15.80	1.20	0.00	5.70
CodeLlama-70b-Ins	7.80	0.60	0.00	2.80
CodeLlama-7b-Base	19.00	0.40	0.00	6.50
CodeLlama-7b-Ins	28.60	2.50	0.00	10.40
CodeQwen15-7B	40.40	4.80	0.00	15.10
CodeQwen15-7B-Chat	39.20	13.10	0.50	17.60
Codestral-Latest	69.00	18.70	0.90	29.50
Command-R	39.00	3.60	0.00	14.20
Command-R+	56.60	6.80	0.00	21.10
DSCoder-1.3b-Base	17.30	0.70	0.00	6.00
DSCoder-1.3b-Ins	22.90	1.50	0.00	8.10
DSCoder-33b-Base	39.40	2.30	0.00	13.90
DSCoder-33b-Ins	55.60	9.00	0.70	21.80
DSCoder-6.7b-Base	34.30	1.40	0.10	11.90
DSCoder-6.7b-Ins	46.40	5.80	0.70	17.60
GPT-3.5-Turbo-0125	56.80	10.80	0.10	22.60
GPT-3.5-Turbo-0301	53.40	8.80	0.20	20.80
GPT-4-0613	78.40	21.20	2.30	33.90
GPT-4-Turbo-1106	84.40	24.00	0.50	36.30
GPT-4-Turbo-2024-04-09	85.30	33.00	5.10	41.10
GPT-4O-2024-05-13	88.30	33.20	4.20	41.90
Gemini-Flash-1.5-May	68.10	12.60	2.70	27.80
Gemini-Pro-1.5-May	76.00	19.40	3.50	33.00
Gemma-7b-Base	27.00	0.90	0.00	9.30
LLama3-70b-Base	52.20	3.20	0.60	18.60
LLama3-70b-Ins	60.70	15.80	1.40	26.00
LLama3-8b-Base	32.90	1.50	0.00	11.50
LLama3-8b-Ins	38.60	3.50	0.50	14.20
MagiCoderS-DS-6.7B	49.20	7.50	0.00	18.90
Mistral-Large	60.20	10.90	0.90	24.00
Mixtral-8x22B-Ins	59.80	12.70	0.00	24.20
Mixtral-8x7B-Ins	31.60	2.60	0.00	11.40
OC-DS-1.3B	11.30	0.10	0.00	3.80
OC-DS-33B	53.90	5.10	0.00	19.70
OC-DS-6.7B	46.30	4.50	0.00	16.90
Phind-34B-V2	53.40	4.70	0.10	19.40
StarCoder2-15b	37.30	2.20	0.00	13.20
StarCoder2-3b	28.20	0.70	0.00	9.60
StarCoder2-7b	29.90	1.20	0.00	10.40

Table 3: Code Generation Performances

Model Name	Easy	Medium	Hard	Total
Claude-2	66.20	10.30	0.40	25.60
Claude-3-Haiku	66.50	8.70	2.50	25.90
Claude-3-Opus	83.10	23.70	6.70	37.80
Claude-3-Sonnet	72.60	11.80	2.20	28.90
Claude-Instant-1	64.40	7.10	2.20	24.60
CodeLlama-13b-Ins	43.10	3.00	0.00	15.30
CodeLlama-34b-Ins	31.50	3.50	1.80	12.30
CodeLlama-7b-Ins	31.90	3.10	1.50	12.10
Codestral-Latest	72.50	25.90	3.30	33.90
DSCoder-1.3b-Ins	29.50	2.10	0.00	10.60
DSCoder-33b-Ins	60.70	8.10	1.50	23.40
DSCoder-6.7b-Ins	49.90	5.70	1.10	18.90
GPT-3.5-Turbo-0125	59.30	11.90	0.50	23.90
GPT-3.5-Turbo-0301	58.40	11.60	0.70	23.60
GPT-4-0613	79.30	25.00	2.40	35.60
GPT-4-Turbo-1106	86.90	36.90	4.00	42.60
GPT-4-Turbo-2024-04-09	88.70	39.70	8.40	45.60
GPT-4O-2024-05-13	92.60	46.40	8.20	49.10
Gemini-Flash-1.5-May	73.40	16.40	4.40	31.40
Gemini-Pro	53.80	9.40	0.20	21.10
Gemini-Pro-1.5-April (n=1)	71.80	19.40	5.50	32.20
Gemini-Pro-1.5-May	84.80	30.10	7.30	40.70
LLama3-70b-Ins	69.60	19.00	1.80	30.10
LLama3-8b-Ins	47.10	6.10	0.00	17.70
MagiCoderS-CL-7B	36.50	3.10	0.00	13.20
MagiCoderS-DS-6.7B	50.60	8.60	0.00	19.70
Mistral-Large	71.20	15.60	3.60	30.10
OC-DS-1.3B	20.00	0.40	0.00	6.80
OC-DS-33B	58.90	7.20	1.30	22.50
OC-DS-6.7B	50.90	6.30	0.20	19.10
Phind-34B-V2	62.00	6.50	0.90	23.10

Table 4: Self Repair Performances

Model Name	Pass@1
Claude-2	32.70
Claude-3-Haiku	32.90
Claude-3-Opus	58.70
Claude-3-Sonnet	34.10
Claude-Instant-1	25.40
CodeLlama-13b-Ins	24.40
CodeLlama-34b-Ins	23.00
CodeLlama-70b-Ins	16.10
CodeLlama-7b-Ins	15.30
Codestral-Latest	41.80
DSCoder-1.3b-Ins	12.50
DSCoder-33b-Ins	28.30
DSCoder-6.7b-Ins	26.50
GPT-3.5-Turbo-0125	35.40
GPT-3.5-Turbo-0301	32.50
GPT-4-0613	52.90
GPT-4-Turbo-1106	55.70
GPT-4-Turbo-2024-04-09	66.10
GPT-4O-2024-05-13	68.90
Gemini-Flash-1.5-May	38.10
Gemini-Pro	29.50
Gemini-Pro-1.5-April (n=1)	49.60
Gemini-Pro-1.5-May	44.80
LLama3-70b-Ins	41.40
LLama3-8b-Ins	24.40
MagiCoderS-CL-7B	21.30
MagiCoderS-DS-6.7B	27.10
Mistral-Large	46.50
Mixtral-8x22B-Ins	44.70
Mixtral-8x7B-Ins	31.80
OC-DS-1.3B	7.80
OC-DS-33B	11.30
OC-DS-6.7B	18.30
Phind-34B-V2	27.20

Table 5: Test Output Prediction Performances

Model Name	Pass@1	Pass@1 (COT)
Claude-2	31.50	43.80
Claude-3-Haiku	0.70	28.30
Claude-3-Opus	36.50	80.10
Claude-3-Sonnet	29.30	39.40
Claude-Instant-1	20.00	34.80
Cllama-13b-Ins	23.50	14.10
Cllama-34b-Ins	28.90	24.50
Cllama-7b-Ins	20.60	14.20
CodeLlama-70b-Ins	31.20	-1.00
Codestral-Latest	37.90	41.80
DSCoder-1.3b-Base	19.00	13.40
DSCoder-1.3b-Ins	18.10	17.00
DSCoder-33b-Base	29.90	29.10
DSCoder-33b-Ins	26.60	31.70
DSCoder-6.7b-Base	23.50	25.10
DSCoder-6.7b-Ins	23.10	23.80
GPT-3.5-Turbo-0301	33.90	34.80
GPT-4-0613	44.30	64.80
GPT-4-Turbo-1106	40.50	83.60
GPT-4-Turbo-2024-04-09	45.90	83.80
GPT-4O-2024-05-13	39.10	91.00
Gemini-Flash-1.5-May	21.40	57.10
Gemini-Pro	27.70	37.40
Gemini-Pro-1.5 (April) (n=1)	30.30	64.40
Gemini-Pro-1.5-May	42.10	72.10
LLama3-70b-Ins	29.60	55.50
LLama3-8b-Ins	18.40	29.40
MagiCoderS-CL-7B	21.20	-1.00
MagiCoderS-DS-6.7B	27.20	-1.00
Mistral-Large	36.60	54.40
Phind-34B-V2	26.90	-1.00
StarCoder	20.30	-1.00
WCoder-34B-V1	28.40	-1.00

Table 6: Code Execution Performances

E Qualitative Examples

E.1 Code Execution

We show 5 examples from the code execution task that GPT-4 (`gpt-4-1106-preview`) still struggles to execute, even with CoT.

```
def countWays(nums: List[int]) -> int:
    nums.sort()
    n = len(nums)
    ans = 0
    for i in range(n + 1):
        if i and nums[i-1] >= i: continue
        if i < n and nums[i] <= i: continue
        ans += 1
    return ans
assert countWays(nums = [6, 0, 3, 3, 6, 7, 2, 7]) == 3
# GPT-4 + CoT Outputs: 1, 2, 4, 5
```

Mistake 1

```
def minimumCoins(prices: List[int]) -> int:

    @cache
    def dfs(i, free_until):
        if i >= len(prices):
            return 0

        res = prices[i] + dfs(i + 1, min(len(prices) - 1, i + i + 1))

        if free_until >= i:
            res = min(res, dfs(i + 1, free_until))

        return res

    dfs.cache_clear()
    return dfs(0, -1)
assert minimumCoins(prices = [3, 1, 2]) == 4
# GPT-4 + CoT Outputs: 1, 3, 5, 6
```

Mistake 2

```
def sortVowels(s: str) -> str:
    q = deque(sorted((ch for ch in s if vowel(ch))))
    res = []
    for ch in s:
        if vowel(ch):
            res.append(q.popleft())
        else:
            res.append(ch)
    return ''.join(res)
assert sortVowels(s = 'lEetcOde') == 'lEOtcede'
# GPT-4 + CoT Outputs: "leetecode", "lEetecOde", "leetcede", "leetcEde", "leetcOde"
```

Mistake 3

```
def relocateMarbles(nums: List[int], moveFrom: List[int], moveTo: List[int]) -> List[
    ↪ int]:

    nums = sorted(list(set(nums)))
    dd = {}
    for item in nums:
        dd[item] = 1
    for a,b in zip(moveFrom, moveTo):
        del dd[a]
        dd[b] = 1
    ll = dd.keys()
    return sorted(ll)
assert relocateMarbles(nums = [1, 6, 7, 8], moveFrom = [1, 7, 2], moveTo = [2, 9, 5])
    ↪ == [5, 6, 8, 9]
# GPT-4 + CoT Outputs: [2, 6, 8, 9], [2, 5, 6, 8, 9], KeyError
```

Mistake 4

```
def minimumSum(nums: List[int]) -> int:
    left, right, ans = [inf], [inf], inf
    for num in nums:
        left.append(min(left[-1], num))
    for num in nums[::-1]:
        right.append(min(right[-1], num))
    right.reverse()
    for i, num in enumerate(nums):
        if left[i] < num and right[i + 1] < num:
            ans = min(ans, num + left[i] + right[i + 1])
    return ans if ans < inf else -1
assert minimumSum(nums = [6, 5, 4, 3, 4, 5]) == -1
# GPT-4 + CoT Outputs: 10, 11, 12
```

Mistake 5