LiveCodeBench Paper Reading

LiveCodeBench Paper Reading

Large Language Models (LLMs) have emerged as a prominent area in code-related applications, attracting significant interest from both academia and industry. However, **With the development of new and improved LLMs, existing evaluation benchmarks (e.g., HumanEval, MBPP) are no longer sufficient to assess their ability to** . In this work, we present LiveCodeBench, a comprehensive and pollution-free benchmark for evaluating LLMs for code writing, which continuously collects new questions from events on the three competition platforms of LeetCode, AtCoder, and CodeForces. It is worth noting that, **Our benchmark also focuses on a broader range of code-related capabilities, such as self-healing, code execution, and test output prediction, beyond just code generation** . Currently, LiveCodeBench hosts four hundred high-quality coding issues, which were released between May 2023 and February 2024. We have evaluated nine base LLMs and 20 instruction-tuned LLMs on LiveCodeBench. We present questions on pollution, comprehensive performance comparisons, potential overfitting in existing benchmarks, and individual model comparisons We will publish all tips and model completion for further community analysis, as well as a common toolkit for adding new scenarios and models.

LLMs have come a long way in the code world, including many code-specific models and different tasks such as:

1. **Program repair** automatic repair program
2. **Optimization** code optimization https://openreview.net/forum?id=ix7rLVHXyY
3. **Test generation** Generate test cases
4. **Documentation generation** Generate repo documentation https://arxiv.org/abs/2402.16667
5. **Tool usage** Toolbench
6. **SQL**

In contrast to these rapid advances, assessment methods are relatively stagnant, and current benchmarks, such as HumanEval, MBPP, and APPS, may paint a biased or misleading picture.

- First, while programming is a multifaceted skill, these benchmarks focus only on natural language-to-code tasks, thus ignoring broader code-related capabilities.

- In addition, since benchmark samples exist in the training dataset, these benchmarks may be subject to potential contamination or overfitting.

Principles

Real-time updates to prevent data pollution

For the new model, we only consider problems published after the model deadline to ensure that the model has not encountered the same problem in the training dataset. In Figure 1, we find that the performance of the DeepSeek model drops sharply when evaluating LeetCode problems published after August 2023. This suggests that DeepSeek may have been trained on the old LeetCode problem. **DeepSeek (red) is heavily polluted**

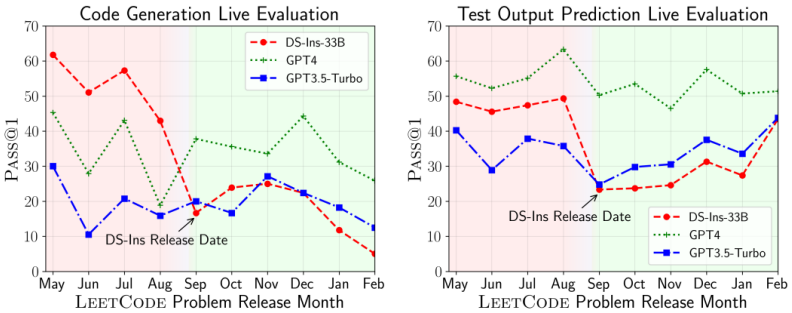

Figure 1: LIVECODEBENCH comprises problems marked with release dates, allowing evaluations over different time windows. For newer models, we can detect and avoid contamination by only evaluating on time-windows after the model's cutoff date. The figures demonstrate the performance of models on code generation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

Overall code evaluation

Current code evaluation focuses primarily on natural language to code generation. However, programming is a multifaceted task that requires capabilities that are not limited to those measured by code generation. In LiveCodeBench, we evaluate how code LLMs perform in the following three additional scenarios:

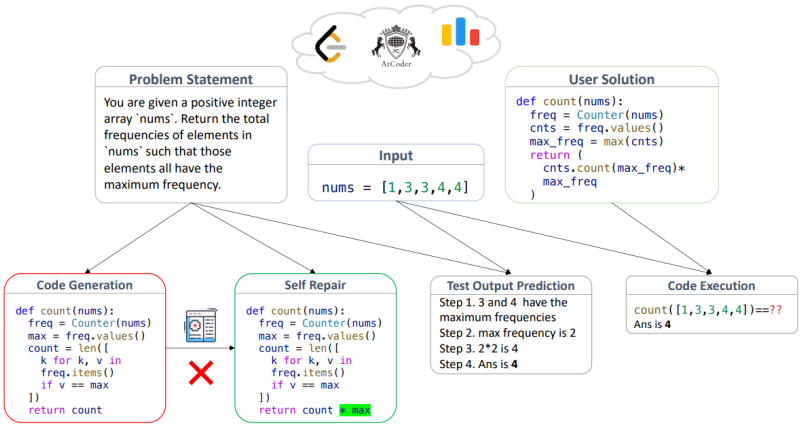

Figure 3: Overview of the different scenarios present in LiveCodeBench. Coding is multi-faceted and we propose evaluating LLMs on a suite of evaluation setups that capture various coding-related capabilities. Specifically, beyond the standard code generation setting, we consider three additional scenarios,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

- Self-Repair: Fix a faulty program based on execution information, evaluating the ability to debug code from feedback. The model is given a natural language description of the problem, the faulty program, the test cases it failed, and the execution feedback for that failure. The output should be a corrected program.
    - Large language models have shown significant capabilities in code generation, but still present difficulties when performing complex tasks. Self-repair - that is, the model debugging and fixing its own code - has recently become a popular method for improving performance in these scenarios. However, despite the growing popularity of self-repair, the scope of existing self-repair research is limited; in many settings, its effects are still not fully understood. In this paper, we analyze the ability of Code Llama, GPT-3.5, and GPT-4 to perform self-repair on issues in HumanEval and APPS. **We found that when the cost of performing the repair is taken into account, the performance gains are usually modest, vary widely between subsets of the data, and sometimes are not present at all. We hypothesized that this is because self-repair is bottlenecked by the model's ability to provide feedback about its own code** ; artificially improving the quality of feedback using a more powerful model, we observed significantly larger performance gains. Similarly, in a small study where we provided GPT-4 with feedback from human participants, we found that even for the strongest models, self-repair still lags far behind what can

- Code Execution: "Execute" a program and input to evaluate code comprehension. The model is given a program and an input, and the output should be the result.

- Test Output Prediction: Solve a natural language task on a specified input and evaluate the ability to generate test output. The model is given a natural language problem description and an input that should be the answer to the question.

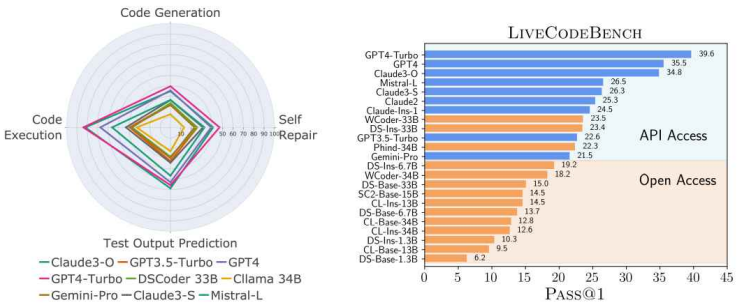

Figure 2: **Left.** We propose evaluating LLMs across scenarios capturing various coding-related capabilities. Specifically, we host four different scenarios, namely code generation, self-repair, code execution, and test output prediction. The figure depicts various model performances across the four 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 DS-Ins-33B and Phind-34B models) crossing the performance gap.

In the simple code generation scenario of LiveCodeBench, we compared the open access and (closed) API access models. We found that the closed access model consistently outperforms the open model in performance, with only large model variants tuned for strong instructions (particularly the DS-Ins-33B and Phind-34B models) able to close this performance gap.

High-quality questions and tests

High-quality questions and tests are essential to reliably evaluate LLMs. However, previous work has revealed the inadequacies of existing benchmarks. Liu et al. (2023a) found problems with inadequate testing and vague problem descriptions in HumanEval. They released HumanEval +, a benchmark variant with more tests that sometimes degraded performance by up to 8%. Similarly, Austin et al. (2021) had to create a sanitized subset of MBPP to disambiguate problem descriptions. In LiveCodeBench, we take questions from well-known contest sites whose quality has been validated by platform users. Additionally, for each question, we provide a large number of tests (over 59 on average) for meaningful and robust evaluation.

The difficulty of the balance problem

Competition programming can be a challenge for even the best performing LLMs, and most current state-of-the-art (SoTA) models perform close to zero on most problems. As a result, they may not be suitable for meaningful comparisons of today's LLMs, as the difference in performance is small. Furthermore, differences between different models are artificially minimized by averaging evaluation scores across problems at different difficulty levels. In contrast, we use problem difficulty ratings from competition sites to ensure a balanced distribution of problem difficulty and allow for fine grain comparisons of models.

Based on these principles, we built LiveCodeBench, a continuously updated benchmark that avoids data pollution. In particular, we collected 400 competition questions from three competition platforms - LeetCode, AtCoder, and CodeForces - from May 2023 to the present (February 2024), and used them to build different LiveCodeBench scenarios.

Empirical findings

Empirical findings. We have evaluated 9 basic models and 20 instruction adjustment models in different LiveCodeBench scenarios. The following are the empirical findings we evaluated, which were not revealed in previous benchmark tests.

1. Pollution. We observed a significant decrease in performance on LeetCode issues released by DeepSeek after August 2023, highlighting the possible pollution in old issues.

2. Comprehensive evaluation. Our evaluation shows that the performance of the model is correlated between different tasks, but the relative differences do vary. For example, in tasks such as self-repair or test output prediction, the gap between open and closed models further increases. Similarly, some models, such as Claude-3-Opus and Mistral-L, perform significantly better than code generation in code execution and test output prediction, with Claude-3-Opus surpassing GPT-4 in test output prediction. This emphasizes the importance of comprehensive evaluation.

3. HumanEval Overfitting. When comparing LiveCodeBench with HumanEval, we found that the models were divided into two groups. One group performed well in both benchmarks, while the other group performed well on HumanEval but not on LiveCodeBench (see Figure 5). The latter group mainly includes fine-tuned open access models, while the former group includes basic and closed models. This indicates that these models may be overfitting for HumanEval.

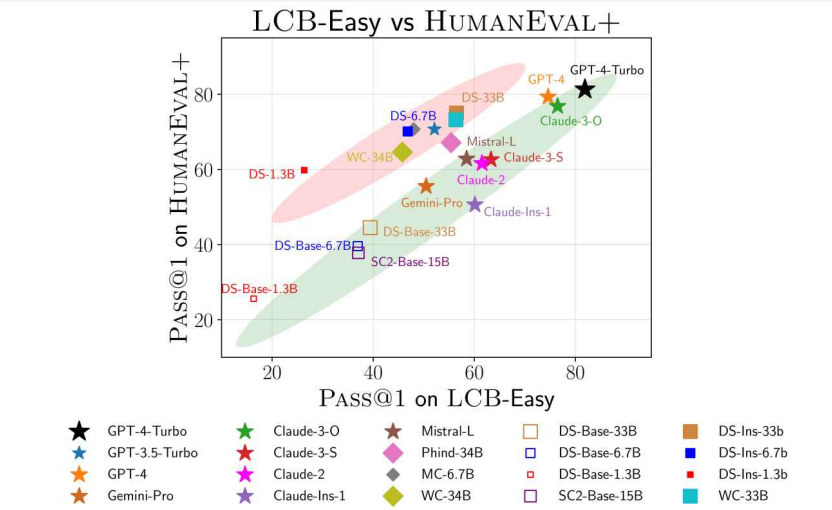

Figure 5: Scatter plot comparing Pass@1 of models on HumanEval+ versus Pass@1 on the easy subset of LiveCodeBench code generation scenario. Star markers denote the closed-access models while other markers denote different open model families. We find that the models are separated into two groups – the 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 Gemini-Pro and Claude-Ins-1 on HumanEval+, it performs significantly worse on the LCB-Easy subset. Similarly, DS-Ins-6.7B outperforms Claude-2 on HumanEval+ but is 14.7 points behind on LCB-Easy subset.

Concurrent work

TACO:

Proposed large datasets of competitive programming problems with additional generative tests, but did not study pollution or tasks other than code generation.

CodeMind

Introduced three types of reasoning tasks:

- Independent Execution Reasoning (IER): Provide input to the model and evaluate the model's prediction of the execution output of any given code
- Dependency Execution Inference (DER): Provide input to the model and predict the code that the model can correctly synthesize for execution output
- Normative Reasoning (SR): Evaluate the degree to which LLMs achieve the specified expected behavior, such as giving specific test cases in prompts

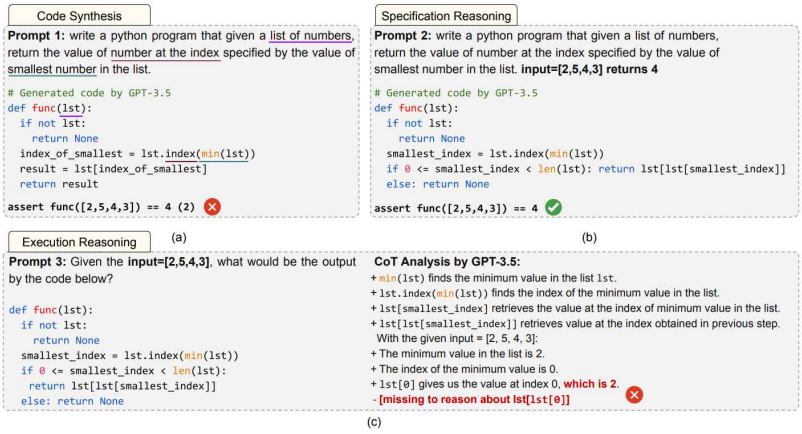

Figure 1. An example illustrating the importance of evaluating LLMs on code reasoning

In addition to the pass rate of the code, an additional evaluation indicator CRR is introduced to represent the accuracy of reasoning

Table 1. CRR performance of subject LLMs on IER task through CoT prompting. We highlight the top three best-performing models with red (1st), green (2nd), and blue (3rd).

Dataset	Programming Language	# Subjects	General LLMs				Code LLMs				
			GPT-4	GPT-3.5	Llama 2	Mistral	CodeLlama	DeepSeekCoder	MagicCoder	StarCoder	WizardCoder
MBPP	Python	408	80.88%	71.32%	45.59%	31.37%	42.40%	57.84%	59.80%	43.63%	46.08%
HumanEval	Python	162	79.01%	64.20%	30.86%	32.72%	45.06%	41.98%	52.47%	38.89%	40.12%
CruxEval	Python	800	80.50%	65.13%	25.38%	34.13%	37.75%	44.38%	46.50%	35.50%	35.88%
CodeNet	Python	1914	70.43%	49.06%	18.97%	17.35%	27.95%	26.65%	33.28%	26.28%	24.87%
	Java	1939	71.17%	51.93%	23.99%	18.15%	28.52%	32.13%	36.46%	29.34%	29.35%
Avatar	Python	86	52.33%	39.53%	24.42%	16.28%	23.26%	18.60%	24.42%	19.77%	24.42%
	Java	86	48.84%	34.88%	23.26%	11.63%	27.91%	23.26%	24.42%	13.95%	13.95%
Total	Java and Python	5395	72.60%	54.24%	24.26%	21.54%	30.40%	33.85%	38.68%	30.14%	29.99%

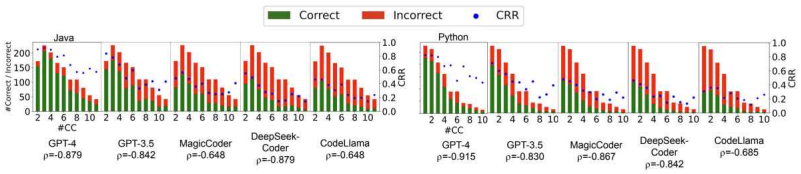

Figure 3. Impact of CC on CRR performance of LLMs in IER

In the normative reasoning (SR) task, the model's drop point is given correct test cases, no test cases, and incorrect test cases

Table 3. Performance of LLMs on SR task. Symbol ↓ indicates a drop from the *previous* setting (row above), and ↑ indicates an increase from the *previous* setting (row above).

Dataset	Setting	General LLMs			Code LLMs				
		GPT-4	GPT-3.5	Mistral	CodeLlama	DeepSeekCoder	MagicCoder	StarCoder	WizardCoder
MBPP	With Test	90.69%	85.05%	50.74%	63.73%	78.68%	75.25%	51.47%	67.89%
	No Test	72.13% ↓	78.87% ↓	48.28% ↓	53.68% ↓	67.65% ↓	69.61% ↓	41.67% ↓	52.21% ↓
	Misleading Test	68.14% ↓	74.02% ↓	50.74% ↑	59.07% ↑	68.63% ↑	67.40% ↓	40.20% ↓	58.09% ↑
HumanEval	With Test	91.98%	74.07%	57.41%	70.37%	87.04%	81.48%	56.17%	76.54%
	No Test	88.27% ↓	70.37% ↓	54.32% ↓	65.43% ↓	82.10% ↓	80.86% ↓	38.89% ↓	76.54%
	Misleading Test	83.95% ↓	65.43% ↓	53.70% ↓	61.73% ↓	79.63% ↓	74.69% ↓	27.04% ↓	66.05% ↓

Classify the framework formats of the code: For, While, If, Try, Switch, Nested Loop, Nested If, and Basic. And calculate the reasoning accuracy of the model on specific category frameworks

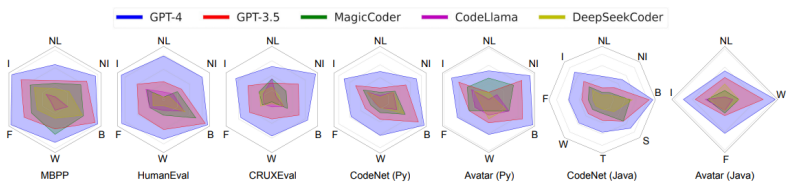

Figure 5. CRR of top five best-performing LLMs per specific code constructs across all datasets. We abbreviate the tags with B (Basic), F (For), I (If), NI (Nested If), NL (Nested Loop), S (Switch), T (Try), and W (While)

NoFunEval

Verify the non-functional-correctness aspects of programming (time and space complexity, etc.)

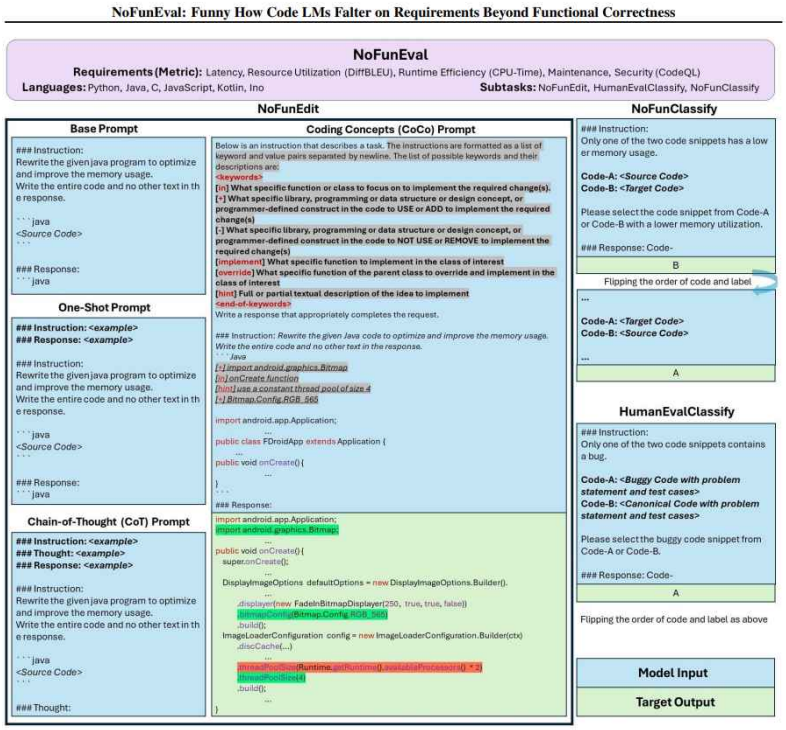

Figure 2: Overview of the NoFunEval benchmark. NoFunEval consists of three subtasks – NoFunEdit, NoFunClassify, HumanEvalClassify, spanning multiple programming languages. NoFunEdit (§ 2.1) involves editing a given source code as per a user-specified non-functional requirement (e.g., improving memory usage). We design four different prompting techniques (§ 2.2) for eliciting LMs to perform the required editing. The Base prompt contains minimal task-related information, while Coding-Concepts prompt is augmented using human-specified high-level hints to complete the task. The second task – NoFunClassify (§ 2.3) involves distinguishing between two code snippets based on a non-functional property (e.g., selecting the code with lower memory utilization). NoFunClassify is constructed by reformulating problems in NoFunEdit. Similarly, we construct HumanEvalClassify (§ 2.4) by reformulating HumanEvalFix (Muennighoff et al., 2023), which involves distinguishing two code snippets based on their functional correctness (i.e., bug detection).

Overall code review

The coding ability of language models is often evaluated and compared through natural language to code generation tasks. However, this only captures one dimension of code-related abilities. **In reality, the expertise required for software engineering in the real world is not limited to generating tasks, but also includes tasks such as synthesizing information-rich test cases, debugging BigInt, understanding existing code, and writing documentation** . These tasks are not just additional documentation work; they are a crucial part of the software development process, helping to improve the quality, maintainability, and reliability of code (Boehm, 2006). This also applies to language models (LLMs), and adopting a similar workflow can make the model perform better in code generation. For example, AlphaCodium (Ridnik et al., 2024) is a complex LLM pipeline used to solve competition coding problems. By combining natural language inference, test case generation, code generation, and self-repair, they achieve significant improvements on the simple baseline of directly generating code, demonstrating the importance of these broader abilities. Inspired by this, we propose a more comprehensive evaluation method for LLMs in this work, using a series of evaluation settings

Bench construction

We wrote HTML crawlers for each of the above websites to collect questions and corresponding metadata. To ensure quality and consistency, we parsed mathematical formulas and excluded questions that contained images. We also excluded questions that were not suitable for scoring through input/output examples, such as those that accepted multiple correct answers or required building a data structure. In addition to parsing problem descriptions, we also collected directly available relevant real solutions and test cases. **Therefore, for each question, we collected tuples of natural language problem statement P, test case T, and real solution S. Finally, we associated match date D with each question to mark the publication date of each question, and used the collected attributes to build the questions for our four scenarios.**

Over time, we associated match date D for each question. The release date allowed us to measure the performance of LLMs in different time windows (called "rolling" time) by filtering questions. This is crucial for evaluating and comparing models trained at different times. **Specifically, for a new model and its corresponding deadline (normalized to release date if no training deadline is published), we can measure the performance of the model on benchmark questions published after the deadline** . We have developed a User Interface that allows models to be compared on questions published in different time windows (as shown in Figure 9).

- **Test set** . Testing is crucial for evaluating the correctness of generated output and is used in all four scenarios. We collect public-facing tests as much as possible and use them for benchmarking. Otherwise, following the method of Liu et al. (2023b), we use LLM (here referring to GPT-4-Turbo) to generate tests for the problem. The key difference between our test generation method and theirs is that we do not directly use LLM to generate inputs, but build generators that sample inputs based on problem specifications using a few sample prompts. Details and examples of such input generators can be found in Section A.2.

- **Problem Difficulty** . Competition programming has always been a challenge for LLMs. The average rating (ELO) of GPT-4 on CodeForces is 392, which is in the lowest 5th percentile (OpenAI, 2023). This makes it difficult to compare LLMs because there is little variation in performance between models. In LiveCodeBench, we collected problems marked with different difficulties on the competition platform, excluding those that were rated as too difficult for even the best models. In addition, we used these ratings to classify problems into easy, medium, and difficult for more detailed model comparisons.

Conclusion

In this work, we propose LiveCodeBench, a new benchmark for evaluating the coding capabilities of LLMs. Our benchmark addresses contamination issues in existing benchmarks by introducing real-time evaluation and emphasizing scenarios beyond code generation, and considers the broader coding capabilities of LLMs. LiveCodeBench is an extensible framework that will continuously update new problems, scenarios, and models. Our evaluation reveals some new findings, such as contamination detection and potential overfitting issues on HumanEval. We hope that LiveCodeBench can drive understanding of current code LLMs through our findings and guide future research in this field.