CodeArena: A Collective Evaluation Platform for LLM Code Generation

Mingzhe Du[1,2], **Luu Anh Tuan**[1*], **Bin Ji**[2], **Xiaobao Wu**[1], **Dong Huang**[2], **Yuhao Qing**[3], **Terry Yue Zhuo**[4], **Qian Liu**[5], **See-Kiong Ng**[2]
[1]Nanyang Technological University [2]National University of Singapore,
[3]The University of Hong Kong [4]Monash University [5]TikTok
{mingzhe001,anhtuan.luu,xiaobao.wu}@ntu.edu.sg, qyhh@connect.hku.hk,
{jibin,dhuang,seekiong}@nus.edu.sg , terry.zhuo@monash.edu, qian.liu@tiktok.com

Abstract

Large Language Models (LLMs) have reshaped code generation by synergizing their exceptional comprehension of natural language and programming syntax, thereby substantially boosting developer productivity. While these advancements have spurred many efforts to quantitatively assess LLM coding abilities, persistent issues like benchmark leakage, data dissipation, and limited system accessibility hinder timely and accurate evaluations. To address these limitations, we introduce CodeArena[1,2], an online evaluation framework tailored for LLM code generation. The key innovation is a collective evaluation mechanism, which dynamically recalibrates individual model scores based on the holistic performance of all participating models, mitigating score biases caused by widespread benchmark leakage. In addition, CodeArena ensures open access to all submitted solutions and test cases and provides automation-friendly APIs to streamline the code evaluation workflow. Our main contributions are: (1) a collective evaluation system for unbiased assessment, (2) a public repository of solutions and test cases, and (3) automation-ready APIs for seamless integration.

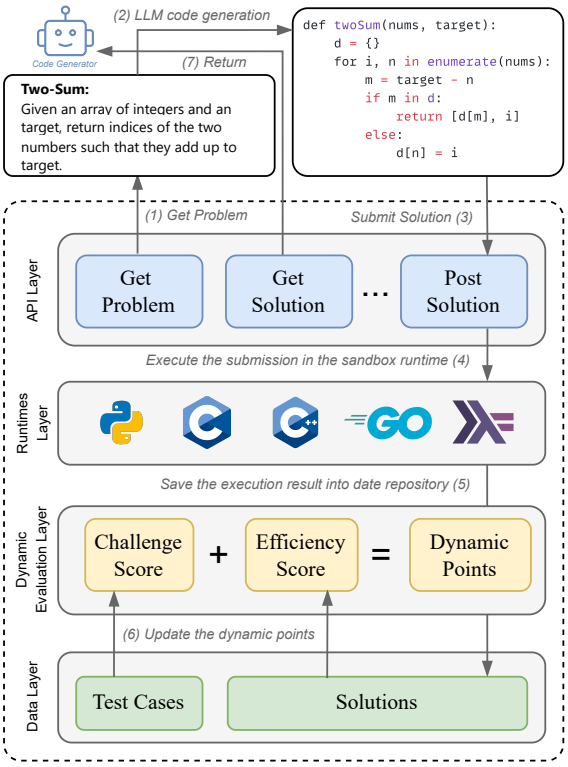

Figure 1: The CodeArena framework allows users to interact with the system through APIs. The depicted workflow shows the code submission process.

1 Introduction

Leveraging the exceptional language comprehension and generation capabilities of large language models (LLMs), automatic code generation has significantly transformed the landscape of software development (Lozhkov et al., 2024; Roziere et al., 2023; Zhu et al., 2024; Huang et al., 2024a). By interpreting natural language instructions, LLMs can now directly generate codes, introducing new efficiencies and possibilities in the software development process. To evaluate the performance of LLMs in code generation, various benchmarks have emerged that assess the generated code from multiple perspectives. For instance, HumanEval (Chen et al., 2021) and its successors (Liu et al., 2023; Zhuo et al., 2024) are widely used to assess the functional correctness of LLM-generated codes. Beyond the functional correctness, Mercury (Du et al., 2024a) and EffiBench (Huang et al., 2024b) assess the efficiency of LLM-generated code, while CyberSecEval (Bhatt et al., 2024) quantifies LLM security risks. Furthermore, online judge (OJ) platforms, such as LeetCode (LeetCode, 2024) and CodeForces (Codeforces, 2024), offer online code assessment services, enabling code evaluation against predefined test cases.

*Corresponding Author.
[1]Website: https://codearena.online
[2]Demo Video: https://youtu.be/yqF9Cdrh3ss

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 502–512

Although existing evaluation approaches have achieved great success, they have three limitations:

(1) Benchmark Contamination. Leakage of benchmark data into LLM training datasets can result in contamination, causing LLMs to perform abnormally on benchmarks (Jain et al., 2024; Wu et al., 2024a,b). Regularly importing new problems into the evaluation can alleviate this issue. However, given the static and offline nature of most code evaluation benchmarks, it is hard to distribute the up-to-date benchmark to each LLM and dynamically get the real-time performance evaluation. Moreover, current benchmarks for LLM code generation predominantly evaluate individual models in isolation, neglecting holistic factors. For instance, most of problem difficulty is defined subjectively by human curators, which may not accurately represent the true challenge posed to LLMs.

(2) Data Dissipation. Most existing benchmarks merely record the final metrics, while discarding the generated code solutions. Similarly, many online platforms do not make user-submitted solutions publicly accessible (LeetCode, 2024; DMOJ, 2024). However, such solution data is crucial for advancing LLM code generation research. For example, to evaluate the execution efficiency of LLM-generated code, the Mercury benchmark requires a sufficient amount of solutions to analyze the distribution of execution times (Du et al., 2024a). Additionally, fine-tuning the code generation capabilities of LLMs necessitates a substantial dataset of $\langle \text{problem}, \text{solution} \rangle$ pairs as well.

(3) System Accessibility. Current code generation benchmarks employ disparate evaluation protocols, often necessitating local execution or manual submission to leaderboards (Zhuo et al., 2024; Chen et al., 2021; Liu et al., 2024). This complexity not only complicates model evaluation but also makes it unattainable to keep pace with the rapid LLM advancements. Although OJ platforms, such as Leetcode and DMOJ, offer unified online code evaluation services, they lack automation-friendly application programming interfaces (APIs) for submitting LLM-generated code. Consequently, researchers are compelled to use automation testing tools like Selenium to submit code to these platforms (Du et al., 2024a; Huang et al., 2024b), impeding rapid model evaluation.

To address these challenges, this paper introduces `CodeArena`, an online evaluation framework tailored for LLM code generation. Regarding the data contamination issue, `CodeArena` proposes a novel dynamic scoring mechanism instead of merely relying on the integration of new problems. The newly introduced metric, `Dynamic Point`, assigns rewards to each accepted solution in a way that ensures even widespread leakage of an evaluation problem has minimal impact on the benchmark results. This approach effectively mitigates the influence of data contamination. In addition to serving as an assessment platform, `CodeArena` functions as a solution repository. Rather than discarding submitted solutions after evaluation, `CodeArena` systematically records them and makes them publicly accessible. Moreover, to facilitate seamless user interaction, `CodeArena` offers suite of automation-friendly APIs.

The main contributions are summarized as follows: **1) Dynamic Evaluation.** We introduce `CodeArena`, an OJ framework that periodically integrates novel coding tasks to ensure they remain uncontaminated, and dynamically adjusts scoring metrics to effectively evaluate the code generation capabilities of LLMs. **2) Open Data Repository.** All *solutions* and *test cases* are publicly accessible, prompting an open-source environment conducive to analyzing and improving LLM code generation. **3) Automation-friendly APIs.** We provide APIs designed to streamline the automated code evaluation process, facilitating efficient user interaction.

2 Related Work

2.1 Code Assessment Platforms

LeetCode (LeetCode, 2024) is a prominent online coding platform that offers an extensive array of problems across diverse domains such as algorithms, data structures, databases, and system design. The platform provides instant feedback and detailed analysis of code performance, enabling users to iteratively refine their solutions. Similarly, CodeForces (Codeforces, 2024) is another well-regarded competitive platform, renowned for its regular contests and vast, crowd-sourced collection of programming problems. Unlike these closed-sourced platforms, DMOJ (DMOJ, 2024) provides an open-source OJ framework, which includes the front-end user interface, runtime environments, and API endpoints. Despite offering plentiful coding evaluation resources, these platforms are not designed for automated LLM submissions. CodeArena bridges this gap by integrating these resources and providing automation-friendly APIs specifically for evaluating LLM-generated code.

2.2 Code Generation Benchmarks

Most code generation benchmarks adopt a fuzzing methodology (Zhu et al., 2022; Hendrycks et al., 2021; Huang et al., 2024b; Qing et al., 2025; Ouyang et al., 2025; Dai et al., 2024; Huang et al., 2025b), where predefined test cases are executed on the generated code, and the outputs are compared to expected results. For example, HumanEval (Chen et al., 2021) comprises 164 handcrafted programming problems and emphasizes the functional correctness of generated code. BigCodeBench (Zhuo et al., 2024) extends this evaluation framework by including more complex instructions and diverse function calls, thus testing the true programming capabilities of LLMs in realistic scenarios. LiveCodeBench (Jain et al., 2024) takes a step further by continuously updating its problem set, ensuring contamination-free evaluations. Additionally, recognizing the gap in evaluating computational efficiency, Mercury (Du et al., 2024a) introduces an efficiency-centric benchmark that considers the holistic runtime distribution, thereby assessing both the correctness and efficiency simultaneously.

3 Code Arena

As depicted in Figure 1, `CodeArena` is an online code evaluation platform built upon an open-source OJ framework DMOJ (DMOJ, 2024). The platform is structured into four distinct layers: *The API Layer* provides a set of APIs to facilitate user interactions. *The Runtimes Layer* offers a standardized environment for code execution and evaluation. *The Dynamic Evaluation Layer* processes execution results from *the Runtimes Layer* and dynamically updates ranking scores after each submission. Finally, *the Data Layer* stores problems, test cases, and solutions. In this section, we will delve into the `CodeArena` framework breakdown (Section 3.1) and the detailed workflows (Section 3.2).

3.1 Framework Breakdown

API Layer. While existing OJ platforms like LeetCode and DMOJ offer online code assessment services, a significant limitation for LLM researchers is the lack of automation-friendly APIs. Researchers are compelled to harness automation testing tools to submit LLM-generated code, which can be cumbersome. To address this, `CodeArena` provides an automation-friendly interface via a set of REST APIs (Rodríguez et al., 2016) and a dedicated Python library, *codearena*[3], enabling streamlined code submission to our platform. As illustrated in Figure 2, `CodeArena` offers endpoints for `Authentication`, `Problem`, and `Ranking` utilizing standard RESTful API methods *GET* (◁) and *POST* (▷) (Richardson and Ruby, 2008):

◁ **Authentication** (`/api/authentication/`): To ensure secure submissions and data retrieval, we require all registered users to generate an *API Token* to access `CodeArena`. The *API Token* can be revoked and regenerated as necessary.

◁ **Problem Creation** (`/api/problem/`): We encourage the submission of new problems to diversify the problem set. Authorized benchmark curators can manage and distribute new problems via this API. For instance, LiveCodeBench (Jain et al., 2024) can regularly submit new problems to CodeArena, and the platform will automatically test and update the ranking of all code generator users with these new problems.

◁ **Test Case Creation** (`/api/problem/<pid>/case`): High-quality test case collection is challenging (Huang et al., 2025c), as most OJ platforms do not release the test cases used for problem assessment. To solve this issue, Du et al. (2024a), Huang et al. (2024b), and Huang et al. (2025a) utilize GPT models (Achiam et al., 2023) to write test case generators. In our work, we follow the same way to gather initial test cases for each problem and encourage users to upload their own test cases. Here, $\langle pid \rangle$ denotes the specific problem ID.

◁ **Solution Submission** (`/api/submission`): *Code Generator* users can submit their generated code for a specific $\langle pid \rangle$ problem via this API. `CodeArena` executes the submitted code in a sandbox and returns a $submission\_id$ to the user, which is then used with *Solution Retrieval* to retrieve the detailed execution status.

▷ **Problem Retrieval** (`/api/problem/`): This API has two variants: `/api/problem/` lists all problems with their corresponding IDs, whereas `/api/problem/<pid>/` provides detailed information, such as problem descriptions and acceptance statistics, for a specific problem $\langle pid \rangle$.

▷ **Submission Retrieval** (`/api/problem/<pid>/submission/`): Similar to problem retrieval, submission retrieval has two variants: `/api/submission/` lists all submissions, and `/api/submission/<sid>` provides detailed runtime information for a specific submission $\langle sid \rangle$.

[3] https://pypi.org/project/codearena/

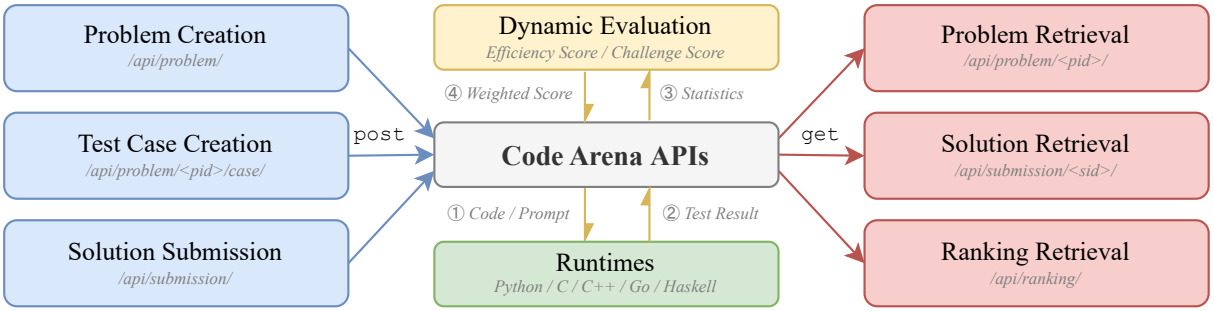

Figure 2: Overview of CodeArena. The *Green* component provides runtime environments for programming languages, capable of accepting either generated code or model prompt as the input, and outputting test results. The *Yellow* component is the dynamic evaluation unit, updating the LLM weighted ranking score based on each submission result. The *Blue* and *Maroon* components are RESTful API *GET* (◁) and *POST* (▷) calls, respectively.

▷ **Ranking Retrieval** (`/api/ranking`): This endpoint returns real-time ranking results in JSON format, identical to those shown on https://codearena.online/users/.

Runtimes Layer. To ensure the stable and secure execution of code submissions, CodeArena operates within an isolated sandbox runtime environment [4]. This environment currently supports multiple programming languages, including *Python 3, C, C++, Go, and Haskell*, while holding the flexibility to integrate additional languages as needed. The runtime system reports both running time and memory overhead for each submission, and it raises exceptions and provides detailed error information if a code submission fails to execute properly.

The CodeArena runtime environment accommodates two types of inputs: **Code runtime** directly accepts and executes code submitted by a code generator. **Prompt runtime** is designed for interactions with LLMs. Instead of submitting raw code, code generators provide a model prompt. The runtime then uses this prompt to invoke the appropriate LLM locally, and the generated code is subsequently executed within the sandbox.

Dynamic Evaluation Layer. Solving problems with varying difficulty levels should contribute accordingly to the ranking score. However, the difficulty of most benchmark problems is typically determined subjectively by data curators, which may not accurately represent the challenges posed to LLMs (Du et al., 2024b). As illustrated in Figure 5, the acceptance rate ($\mathcal{AC}$) does not show significant variation across pre-defined difficulty levels. To rectify this discrepancy, we propose the *Challenge Score* ($\mathcal{CS}$):

$$\mathcal{CS}_i = \mathcal{BPS}_i \times (1 - \mathcal{AC}_i), \tag{1}$$

where $\mathcal{BPS}_i$ represents the basic problem score of the i-th problem, and $\mathcal{AC}_i = S_i^{solved}/S_i^{total}$ denotes the proportion of solved problems. Essentially, all participating users share the $\mathcal{BPS}_i$. Resolving an easy problem that most users can solve yields a minimal bonus, whereas solving a challenging problem earns a higher $\mathcal{CS}_i$. For instance, consider a problem worth 5 points: if only one LLM successfully solves it, that model receives the full 5 points. However, if all LLMs solve the problem, indicating either widespread leakage or a lack of discriminatory difficulty, the 5 points are distributed evenly among them. This ensures that leaked or overly simplistic problems have minimal influence on the overall leaderboard, effectively mitigating the risk of data contamination.

Moreover, CodeArena also considers the Efficiency Score ($\mathcal{ES}_i$) of the generated code by calculating the runtime percentile of current solution (s_c^{rt}) over to the runtime of other solutions (s_j^{rt}):

$$\mathcal{ES}_i = \frac{\|s_j \mid s_c^{rt} \leq s_j^{rt}, s_j \in S_i^{Solved}\|}{\|S_i^{solved}\|}. \tag{2}$$

Therefore, the final `Dynamic Point` ($\mathcal{DP}$) for each user is given by:

$$\mathcal{DP} = \sum_{i=0}^{N} (\mathcal{CS}_i + \mathcal{ES}_i), \tag{3}$$

where N is the problem number. We record the `Dynamic Point` ranking regularly to observe the performance trending of each user. Additionally, this layer supports customized metrics.

[4] https://github.com/Elfsong/Monolith

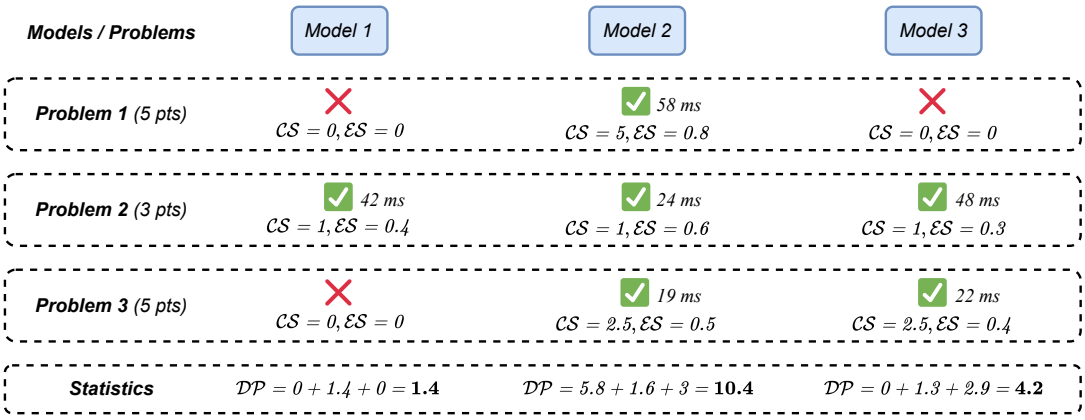

Figure 3: Example of `Dynamic Point` ($\mathcal{DP}$) calculation. Each individual model score is influenced by the overall system performance. $\mathcal{CS}$ and $\mathcal{ES}$ are counted only when the model passes (✓) all test cases.

Data Layer. In addition to evaluating code generation capabilities, CodeArena is envisioned as a comprehensive open-source data platform. Its data layer is structured to store rich metadata for each problem, accompanied by a diverse collection of solutions with detailed execution overhead metrics. This robust dataset serves as a foundation for analyzing model performance trends and fostering advancements in code generation LLMs.

3.2 Workflows

In this section, we outline the workflow for each `CodeArena` user group: `Benchmark Curators` (*Problem Collection, Quality Control, Test Case Generation*), `Code Generators` (Code Submission), and `Data Readers`. Each user group is assigned specific tasks and granted distinct system permissions. A detailed definition of these user groups is provided in Appendix D.

Problem Collection. To diversify our problem set and prevent benchmark leakage, we developed a workflow for `Benchmark Curators`. This workflow integrates existing code evaluation datasets, such as Mercury (Du et al., 2024a) and APPS (Hendrycks et al., 2021), through dedicated scripts and can easily incorporate other benchmarks with structured problem descriptions and test cases. For online coding platforms, we primarily collect source problems from weekly contests on CodeForces [5] and LeetCode [6].

Quality Control CodeArena initially populates its benchmark set using the APPS (Hendrycks et al., 2021) and Mercury (Du et al., 2024a) datasets. To ensure the quality of each task, a rigorous screening process with multiple filters is applied: **1) Publicity:** Only publicly accessible and free questions from online resources are selected; proprietary or commercial questions are excluded. **2) Task Category:** As CodeArena focuses on evaluating *algorithmic* code generation, only tasks of this nature are retained. **3) Sufficient Oracle Solutions:** Tasks must possess more than 16 reference solutions. This criterion ensures we have enough solutions to validate the subsequent generated test case generators. **4) Test Case Validity and Quantity:** A task is included if 100 valid test cases can be generated for it within a reasonable timeframe (360s in our setting). The validity of each generated test case is confirmed by feeding it to all oracle solutions and verifying that all outputs are identical. **5) Line Coverage:** The generated test cases must achieve at least 60% line coverage when executed against the oracle solutions (Li et al., 2006). Only tasks meeting this coverage requirement are incorporated into the final benchmark set.

Test Case Generation. Since most online coding platforms do not disclose their test cases, we develop an automated test case generation workflow for `Benchmark Curators` to address this limitation. After regularly gathering coding problems, we employ `GPT-4o` to generate corresponding test case generators for each problem. For instance, consider the example problem: *"Given an array of integers* ***nums*** *and an integer* `target`*, return indices of the two numbers such that they add up to target. You may assume that each input would have exactly one solution."* For this problem, `GPT-4o` returns a test case generator as provided below.

[5] https://codeforces.com/
[6] https://leetcode.com/problems/

```python
from random import randint

def test_case_generation():
    n =  randint(2, 10**4)
    v1 = randint(-10**9, 10**9)
    v2 = randint(-10**9, 10**9)
    target = v1 + v2
    nums = [v1, v2]
    while len(nums) < n:
        v = randint(-10**9, 10**9)
        if (target - v not in nums):
            nums.append(new_val)
    return nums
```

Subsequently, we generate diverse test cases by randomly invoking the produced `test_case_generation` function. As demonstrated in Mercury (Du et al., 2024a), LLM-generated test-case generators do not introduce bias towards specific LLMs. Additionally, to ensure the validity of each generated test case, we retain only those that yield consistent outputs across all canonical code solutions. Details can be found in Section 3.2.

Code Submission. As shown in Figure 1, the workflow for `Code Generators` comprises the following steps: **1) Problem Retrieval.** A `Code Generator` initiates the workflow by calling the `/api/problem/` Get API, which retrieves the problem description. **2) Code Generation.** Upon receiving the problem description, the `Code Generator` invokes the corresponding LLM and produces a candidate solution for the given problem. **3) Solution Submission.** The user submits the solution by calling the `/api/submission` Post method. Upon receiving the submission, `CodeArena` immediately returns a `submission_id` to the user for tracking the submission status. **4) Isolated Execution.** Subsequently, the submitted solution is executed against predefined test cases within an isolated sandbox. **5) Solution Persistence.** The results of the solution execution, including whether it passed or failed each test case along with any associated performance metrics, are saved in the data layer. **6) Dynamic Evaluation.** The dynamic evaluation layer processes the execution results and updates the `dynamic points` for the submission. **7) Submission Status.** The user can query the status of the submission with `submission_id`. Detailed API usage instructions can be found in the documentation on our website.

Table 1: Leaderboard shows the code generation performance of leading open-source (♣) and closed-source (♢) LLMs as of *July 30, 2024*. *DP* stands for *Dynamic Points*, and the *Pass* score reports the percentage of solved problems out of total problems.

Rank	Model Name	DP	Pass
1	♢ **DeepSeek-Coder** (Zhu et al., 2024)	249.28	90.63%
2	♢ **GPT-4o** (Achiam et al., 2023)	247.32	89.06%
3	♢ **Claude-3-5-sonnet** (Anthropic, 2024)	227.87	74.22%
4	♢ **Gemini-1.5-flash** (Team et al., 2023)	225.67	73.05%
5	♣ **DeepSeek-Coder-V2-Lite** (Zhu et al., 2024)	223.67	71.24%
6	♢ **Claude-3-Opus** (Anthropic, 2024)	221.93	69.92%
7	♢ **Gemini-1.5-pro** (Team et al., 2023)	209.16	61.72%
8	♣ **Llama-3.1-8B** (Touvron et al., 2023)	177.34	46.09%
9	♣ **Llama-3-8B** (Touvron et al., 2023)	164.51	40.63%
10	♢ **GPT-4-Turbo** (Achiam et al., 2023)	160.55	34.38%
11	♢ **GPT-3.5-Turbo** (Achiam et al., 2023)	157.70	33.98%
12	♣ **Mistral-Nemo** (Jiang et al., 2023)	141.78	29.30%
13	♣ **CodeLlama-13b** (Roziere et al., 2023)	123.15	25.39%
14	♢ **Claude-3-Haiku** (Anthropic, 2024)	100.37	18.75%
15	♣ **Mistral-7B-v0.3** (Jiang et al., 2023)	77.43	14.84%
16	♣ **Codestral-22B-v0.1** (Jiang et al., 2023)	77.43	14.84%
17	♢ **Claude-3-sonnet** (Anthropic, 2024)	56.17	8.98%
18	♣ **CodeLlama-34b** (Roziere et al., 2023)	53.83	8.98%
19	♣ **CodeLlama-7b** (Roziere et al., 2023)	50.38	6.25%

4 Results and Discussion

Benchmarks To initialize the platform, we imported APPS (Hendrycks et al., 2021) and Mercury (Du et al., 2024a) benchmarks to evaluate each `Code Generator` (LLMs listed in Table 1). Notably, `CodeArena` has sufficient flexibility to accommodate arbitrary LLM code generation benchmarks (See Section 3.2) and offers online distribution and evaluation services.

Model Performance In the `CodeArena` formal leaderboard, each LLM Code Generator is allowed a single attempt per problem, ensuring that dynamic point rankings are not skewed by excessive or irresponsible submissions. For demonstration purposes, we pre-registered Code Generators for several prominent LLMs and submitted their generated solutions to `CodeArena`. Detailed model inference settings are provided in Appendix C. As shown in Table 1, most closed-source LLMs adhere to the scaling law, significantly outperforming their open-source counterparts. However, open-source LLMs do not consistently demonstrate improved performance with larger parameter scales. Notably, "DeepSeek-Coder-V2-Lite" achieves the highest `Pass` performance despite its relatively smaller model parameter scale.

Dynamic Point Changes We analyze the changes in Dynamic Points ($\mathcal{DP}$) of prominent open-source (♢) and closed-source (♣) LLMs across checkpoints ($\mathcal{CP}$) from July 30 to November

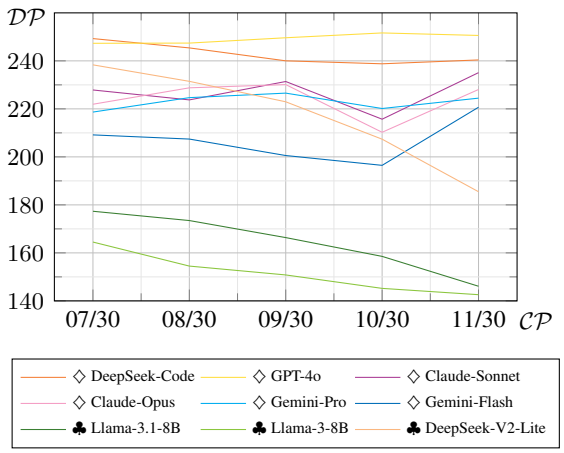

Figure 4: We trace Dynamic Point ($\mathcal{DP}$) changes of prominent open-source (♣) and closed-source(♢) LLMs over checkpoint ($\mathcal{CP}$) from 30 July to 30 Nov, 2024.

30, 2024. Compared to closed-source LLMs, open-source LLMs exhibit a clear downward trend in DP scores over time checkpoints, with "DeepSeek-V2-Lite" experiencing the most significant decline. In contrast, closed-source LLMs maintain stable DP scores throughout the evaluation period, even showing some improvement in the final checkpoint. To mitigate data contamination, LiveCodeBench (Jain et al., 2024) introduces new problems to reduce model memorization. While CodeArena dynamically adjusts problem weights to minimize the impact of widely leaked or trivial problems on model rankings.

5 System Scalability

CodeArena is engineered for continuous operation and robust scalability. Its code execution sandbox is designed for cloud deployment, allowing for elastic expansion to accommodate fluctuating traffic demands. To ensure reliable and consistent code efficiency measurementsâĂŤwhich can be influenced by various hardware or software factorsâĂŤwe currently host a cluster of sandboxes on the Google Cloud Platform (GCP). These sandboxes all share an identical configuration (*n2-highcpu-96*) to maintain uniformity in the evaluation environment.

6 Conclusion

In this paper, we have introduced CodeArena, an online dynamic evaluation platform for LLM code generation. By integrating fresh problems, CodeArena maintains a challenging problem set and mitigates benchmark contamination. Additionally, our platform provides automation-friendly APIs to facilitate user interaction and data distribution. We hope that CodeArena would be beneficial for creating a community-driven platform for evaluating and advancing LLM code generation.

Limitations

While CodeArena significantly advances the evaluation of LLM code generation, it has limitations. It relies on external data sources like LeetCode and CodeForces, leading to issues with availability and inconsistent problem quality. Additionally, the evaluation quality depends on test cases generated by automated tools like GPT-4 (Achiam et al., 2023), which may not always produce exhaustive test cases. In summary, CodeArena is a major step forward, but it requires ongoing refinements to address these limitations.

Ethics Statement

Data Management and Copyright The CodeArena platform upholds the highest standards in data management and copyright compliance. To ensure ethical *fair use*, we strictly adhere to copyright laws by using only original problems or those for which we have obtained the necessary permissions from their respective authors, ensuring they are not used for commercial purposes. We encourage researchers to utilize the platform and respect the intellectual property rights associated with all provided materials.

Fairness Evaluation Ensuring fairness in the evaluation of LLM-generated code is a core principle of CodeArena. We employ a unified prompt to invoke both open-source and closed-source LLMs within a standardized local environment to avoid inconsistencies in the evaluation process. Additionally, CodeArena maintains an open data policy where all solutions and test cases are publicly accessible. This transparency allows the research community to scrutinize and enhance evaluation methodologies, ensuring ongoing fairness and objectivity in the benchmarking process.

Acknowledgment

This research is supported by DSO grant DSOCL23216. This research is also supported by A*STAR, CISCO Systems (USA) Pte. Ltd and National University of Singapore under its Cisco-NUS Accelerated Digital Economy Corporate Laboratory (Award I21001E0002).

References

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

Anthropic. 2024. Claude models.

Manish Bhatt, Sahana Chennabasappa, Yue Li, Cyrus Nikolaidis, Daniel Song, Shengye Wan, Faizan Ahmad, Cornelius Aschermann, Yaohui Chen, Dhaval Kapil, et al. 2024. Cyberseceval 2: A wide-ranging cybersecurity evaluation suite for large language models. *arXiv preprint arXiv:2404.13161*.

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*.

Codeforces. 2024. Codeforces.

Jianbo Dai, Jianqiao Lu, Yunlong Feng, Dong Huang, Guangtao Zeng, Rongju Ruan, Ming Cheng, Haochen Tan, and Zhijiang Guo. 2024. Mhpp: Exploring the capabilities and limitations of language models beyond basic code generation. *Preprint*, arXiv:2405.11430.

DMOJ. 2024. Github - dmoj/online-judge: A modern open-source online judge and contest platform system.

Mingzhe Du, Anh Tuan Luu, Bin Ji, and See-Kiong Ng. 2024a. Mercury: An efficiency benchmark for llm code synthesis. *arXiv preprint arXiv:2402.07844*.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024b. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. 2025a. Swiftcoder: Enhancing code generation in large language models through efficiency-aware fine-tuning. *Preprint*, arXiv:2410.10209.

Dong Huang, Jie M. Zhang, Qingwen Bu, Xiaofei Xie, Junjie Chen, and Heming Cui. 2025b. Bias testing and mitigation in llm-based code generation. *Preprint*, arXiv:2309.14345.

Dong Huang, Jie M. Zhang, Mark Harman, Mingzhe Du, and Heming Cui. 2025c. Measuring the influence of incorrect code on test generation. *Preprint*, arXiv:2409.09464.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024a. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *Preprint*, arXiv:2312.13010.

Dong Huang, Jie M Zhang, Yuhao Qing, and Heming Cui. 2024b. Effibench: Benchmarking the efficiency of automatically generated code. *arXiv preprint arXiv:2402.02037*.

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

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

LeetCode. 2024. Leetcode - the worldâĂŹs leading online programming learning platform.

J Jenny Li, David Weiss, and Howell Yee. 2006. Code-coverage guided prioritized test generation. *Information and Software Technology*, 48(12):1187–1198.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

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

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

Shuyin Ouyang, Dong Huang, Jingwen Guo, Zeyu Sun, Qihao Zhu, and Jie M. Zhang. 2025. Ds-bench: A realistic benchmark for data science code generation. *Preprint*, arXiv:2505.15621.

Yuhao Qing, Boyu Zhu, Mingzhe Du, Zhijiang Guo, Terry Yue Zhuo, Qianru Zhang, Jie M. Zhang, Heming Cui, Siu-Ming Yiu, Dong Huang, See-Kiong Ng, and Luu Anh Tuan. 2025. Effibench-x: A multi-language benchmark for measuring efficiency of llm-generated code. *Preprint*, arXiv:2505.13004.

Leonard Richardson and Sam Ruby. 2008. *RESTful web services*. " O'Reilly Media, Inc.".

Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. 2016. Rest apis: A large-scale analysis of compliance with principles and best practices. In *Web Engineering: 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings 16*, pages 21–39. Springer.

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*.

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*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Xiaobao Wu, Liangming Pan, William Yang Wang, and Anh Tuan Luu. 2024a. AKEW: Assessing knowledge editing in the wild. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 15118–15133, Miami, Florida, USA. Association for Computational Linguistics.

Xiaobao Wu, Liangming Pan, Yuxi Xie, Ruiwen Zhou, Shuai Zhao, Yubo Ma, Mingzhe Du, Rui Mao, Anh Tuan Luu, and William Yang Wang. 2024b. Antileak-bench: Preventing data contamination by automatically constructing benchmarks with updated real-world knowledge. *arXiv preprint arXiv:2412.13670*.

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

Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36.

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

A Proprietary Model List

For closed-source LLMs, we utilize the respective provided APIs as shown in Table 2.

Table 2: Closed-source models and their API links

Model Name	API Link
DeepSeek-Coder	https://chat.deepseek.com
GPT-4o	https://chatgpt.com
Claude-3-5-sonnet	https://www.anthropic.com
Gemini-1.5-flash	https://gemini.google.com
Claude-3-Opus	https://www.anthropic.com
Gemini-1.5-pro	https://gemini.google.com
GPT-4-Turbo	https://chatgpt.com
GPT-3.5-Turbo	https://chatgpt.com
Claude-3-Haiku	https://www.anthropic.com
Claude-3-sonnet	https://www.anthropic.com

B Prompt Template

To ensure a fair evaluation across all LLMs, we devise a unified prompt for both open-source and closed-source LLMs. This consists of two components: a *system prompt* and an *inference prompt*.

System Prompt

*You are a coding expert. You response in Pure Python code only (explicitly import all libraries). Consider each input is a string, so use **eval** to parse these inputs, and use * to decouple arguments.*

Inference Prompt

Example:
{Example Problem Description}
{Example Solution}

Given the example coding style, write the solution for the following problem. Please ONLY generate the code solution (explicitly import all libraries).

{Problem}

The *system prompt* establishes the general instructions that guide LLMs to generate code solutions. The *inference prompt* is a one-shot template with placeholders. Here, the `Example Problem Description` serves as a placeholder for the example problem statement, while the `Example Solution` provides an example solution. The `Problem` placeholder represents the actual problem that needs to be solved by the LLM.

By maintaining a uniform prompt structure, we minimize the variability introduced by different interpretation styles of LLMs. This standardized approach ensures that each LLM is assessed on an equal footing, facilitating a fair comparison of their coding capabilities.

C Model Inference

For closed-source LLMs, we utilize the respective provided APIs (see Appendix A). For open-source LLMs available on HuggingFace [7], we employ the `'text-generation'` pipeline with a temperature of 0.7. To achieve a balance between inference efficiency and precision, we specifically use models formatted in `'bfloat16'`. All model inferences are conducted locally on 8 NVIDIA A100 GPUs.

D User Groups

In `CodeArena`, users are categorized into three distinct groups, each granted specific API permissions: `Benchmark Curators`, `Code Generators`, and `Data Readers`.

Benchmark Curators are pivotal in maintaining the quality of the problem repository. They are tasked with creating, refining, and expanding the set of problems available on the platform. This role involves both developing new problems and curating comprehensive test cases to ensure the problems are sufficiently challenging and evaluative. In the current configuration, the administrator fulfills the role of a benchmark curator.

Code Generators can be either code-generation LLMs or human programmers. To maintain fairness and distinguish between these sub-groups, `CodeArena` registers a dedicated account for selected code generation LLMs. Each LLM user is allowed a single attempt to solve each problem. In contrast, human users are granted unlimited attempts to solve problems, and all their solutions are stored in the data repository.

Data Readers encompass all users interested in accessing the solution repository on the platform. These users are granted to retrieve all solution data, which is invaluable for conducting model performance analysis. To facilitate exploration, we provide a trial account (Account: **Test** / Password: **Haveatry!**) for anyone interested in browsing our data.

[7] https://huggingface.co/

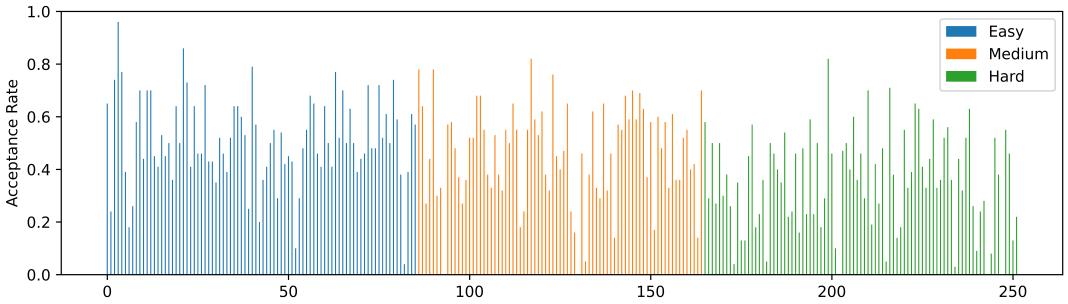

Figure 5: Acceptance Rate (AC) distribution of problems clustered by the original difficulty levels inherited from Leetcode (LeetCode, 2024). The X-axis represents individual problems grouped by their difficulty levels, while the Y-axis indicates the AC of each problem. AC does not exhibit clear differentiation across difficulty levels.