Copilot Arena: A Platform for Code LLM Evaluation in the Wild

Wayne Chi*
Valerie Chen*
Carnegie Mellon University
Pittsburgh, PA, USA

Anastasios Nikolas Angelopoulos
UC Berkeley
Berkeley, CA, USA

Wei-Lin Chiang
UC Berkeley
Berkeley, CA, USA

Aditya Mittal
Carnegie Mellon University
Pittsburgh, PA, USA

Naman Jain
UC Berkeley
Berkeley, CA, USA

Tianjun Zhang
UC Berkeley
Berkeley, CA, USA

Ion Stoica
UC Berkeley
Berkeley, CA, USA

Chris Donahue†
Ameet Talwalkar†
Carnegie Mellon University
Pittsburgh, PA, USA

ACM Reference Format:
Wayne Chi, Valerie Chen, Anastasios Nikolas Angelopoulos, Wei-Lin Chiang, Aditya Mittal, Naman Jain, Tianjun Zhang, Ion Stoica, Chris Donahue, and Ameet Talwalkar. 2018. Copilot Arena: A Platform for Code LLM Evaluation in the Wild. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

As model capabilities improve, large language models (LLMs) are increasingly integrated into user environments and workflows. For example, software developers code with AI in integrated developer environments (IDEs) [39], doctors rely on notes generated through ambient listening [36], and lawyers consider case evidence identified by electronic discovery systems [50]. Increasing deployment of models in productivity tools demands evaluation that more closely reflects real-world circumstances [18, 23, 41]. While newer benchmarks and live platforms incorporate human feedback to capture real-world usage, they almost exclusively focus on evaluating LLMs in chat conversations [10, 14, 25, 52]. Model evaluation must move beyond chat-based interactions and into specialized user environments.

In this work, we focus on evaluating LLM-based coding assistants. Despite the popularity of these tools—millions of developers use Github Copilot [16]—existing evaluations of the coding capabilities of new models exhibit multiple limitations (Figure 1, bottom). Traditional ML benchmarks evaluate LLM capabilities by measuring how well a model can complete static, interview-style coding tasks [2, 8, 20, 47] and lack *real users*. User studies recruit real users to evaluate the effectiveness of LLMs as coding assistants, but are often limited to simple programming tasks as opposed to *real tasks* [32, 40, 44]. Recent efforts to collect human feedback such as Chatbot Arena [10] are still removed from a *realistic environment*, resulting in users and data that deviate from typical software development processes. We introduce Copilot Arena to address these limitations (Figure 1, top), and we describe our three main contributions below.

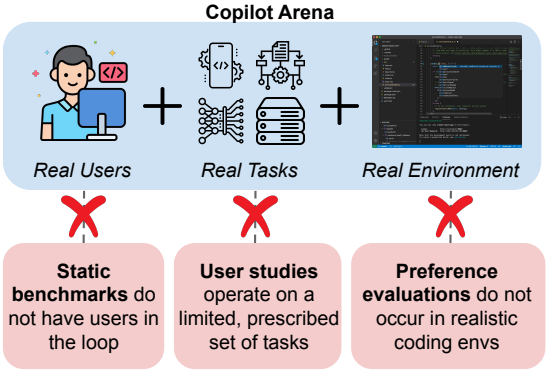

Figure 1: Copilot Arena is a platform for conducting realistic evaluations of code LLMs, collecting human preferences of coding models with real users, real tasks, and in realistic environments, aimed at addressing the limitations of existing evaluations.

We deploy Copilot Arena in-the-wild to collect human preferences on code. Copilot Arena is a Visual Studio Code extension, collecting preferences directly in a developer's IDE within their actual workflow (Figure 2). Copilot Arena provides developers with code completions, akin to the type of support provided by Github Copilot [16]. Over the past 3 months, Copilot Arena has served over 4.5 million suggestions from 10 state-of-the-art LLMs, gathering 11604 votes from 1642 users. To collect user preferences, Copilot Arena presents a novel interface that shows users paired

*Equal Contribution. Correspondence to waynechi@andrew.cmu.edu and valeriechen@cmu.edu.
†Co-senior Authors.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

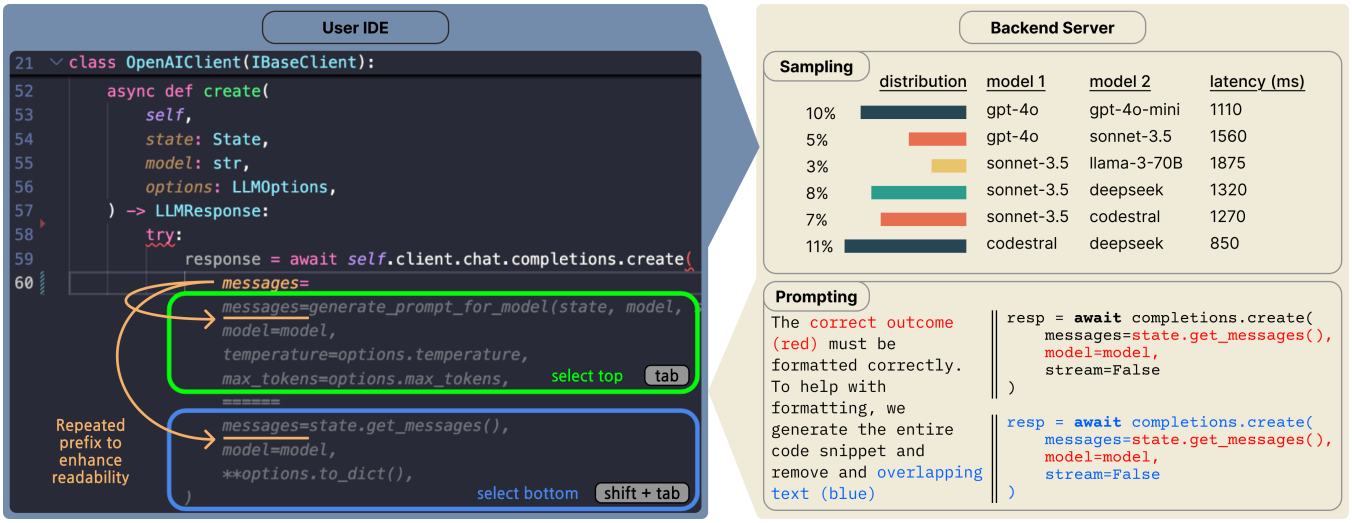

Figure 2: We introduce Copilot Arena, a VSCode extension to collect human preferences of code directly in a developer's IDE. Copilot Arena enables developers to use code completions from various models. The system comprises a) the interface in the user's IDE which presents paired completions to users (left), b) a sampling strategy that picks model pairs to reduce latency (right, top), and c) a prompting scheme that allows diverse LLMs to perform code completions with high fidelity. Users can select between the top completion (green box) using `tab` or the bottom completion (blue box) using `shift+tab`.

code completions from two different LLMs, which are determined based on a sampling strategy that aims to mitigate latency while preserving coverage across model comparisons. Additionally, we devise a prompting scheme that allows a diverse set of models to perform code completions with high fidelity. See Section 2 and Section 3 for details about system design and deployment respectively.

We construct a leaderboard of user preferences and find notable differences from existing static benchmarks and human preference leaderboards. In general, we observe that smaller models seem to overperform in static benchmarks compared to our leaderboard, while performance among larger models is mixed (Section 4). We attribute these differences to the fact that Copilot Arena is exposed to users and tasks that differ drastically from code evaluations in the past. Our data spans 103 programming languages and 24 natural languages as well as a variety of real-world applications and code structures, while static benchmarks tend to focus on a specific programming and natural language and task (e.g. coding competition problems). Additionally, while all of Copilot Arena interactions contain code contexts and the majority involve infilling tasks, a much smaller fraction of Chatbot Arena's coding tasks contain code context, with infilling tasks appearing even more rarely. We analyze our data in depth in Section 5.1.

We derive new insights into user preferences of code by analyzing Copilot Arena's diverse and distinct data distribution. We compare user preferences across different stratifications of input data (e.g., common versus rare languages) and observe which affect observed preferences most (Section 5.2). For example, while user preferences stay relatively consistent across various programming languages, they differ drastically between different task categories (e.g. frontend/backend versus algorithm design). We also observe variations in user preference due to different features related to code structure (e.g., context length and completion patterns). We open-source Copilot Arena and release a curated subset of code contexts. Altogether, our results highlight the necessity of model evaluation in realistic and domain-specific settings.

2 System Design

Copilot Arena is a VSCode extension that provides users with pairs of inline code completions from various LLMs. In return, users provide their votes on which completion is better suited for their task. To avoid interrupting user workflows, voting is designed to be *seamless*—users use keyboard shortcuts to quickly accept one of the two completions into their code, which we interpret as a vote in favor of the underlying model that produced it. Designed to allow for developer's day-to-day usage, the three core components of Copilot Arena (Figure 2) are 1) the User Interface, 2) Model Sampling, and 3) Model Prompting.

2.1 User Interface

Traditional code completion tools (e.g., GitHub Copilot [16]) only show one completion at a time. However, showing two code completions simultaneously enables us to collect preference judgments on the same context [10, 30]. We propose an interface that allows a user to view two completions in a head-to-head manner; to our knowledge, we are the first to introduce an interface that does so. We propose a design inspired by Git Diff—a well-established tool familiar to many developers—which displays code from the current commit and code from the incoming commit stacked vertically, one on top of the other. In a similar manner, given an existing code context, we also stack responses from two different model outputs. This allows users to examine both completions together (an example of how the completions are visualized is in Figure 2). The user

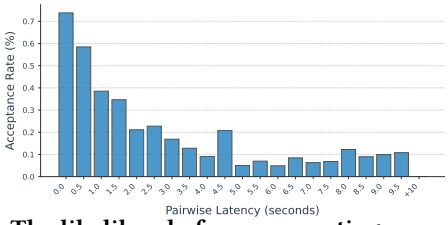

Figure 3: The likelihood of users accepting one of the two completions as a function of empirical pairwise latency (determined by the slower of the two models). As latency increases, users are less likely to accept a completion. We devise a sampling strategy described in Section 2.2 which reduces pairwise latency by 33% while also ensuring sufficient coverage of unique model pairs.

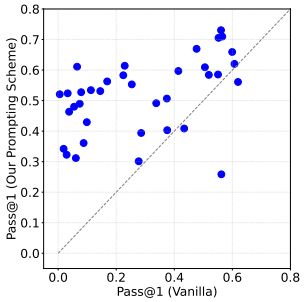

Figure 4: We evaluate the effectiveness of our prompting scheme by comparing LLM performance on infilling tasks (using pass@1) before and after applying it. We evaluate 9 different models of varying performance across 4 different prompt templates (i.e., ways of encoding the prefix and suffix in the prompt): each point represents one model and one prompt template pair. We observe that, across the board, the overwhelming majority of pairs benefit from our prompting scheme (e.g., lie above the diagonal line).

can accept the top suggestion using `tab` and the bottom suggestion using `shift+tab`, or decide neither is appropriate and continue typing. The only distinction between our system and conventional inline completion systems is the inclusion of a second suggestion, resulting in a user experience that is familiar overall.

We make several other notable design decisions. First, we repeat the first line in the ghost text of the top completion so that both top and bottom completions are entirely ghost text. Not repeating the text—as is the case with a single completion—was an alternative we considered, but our initial pilot studies indicated that the discrepancy between top (partial ghost text) and bottom (full ghost text) completions was more likely to confuse users. Second, we always wait for both completions to finish generating before showing them to the user to reduce the effects of latency on user preference, which we aim to study separately in Section 5.2. Lastly, we randomize the ordering of the completions to remove top-bottom bias from our preference evaluation.

2.2 Model Sampling

A key challenge in building a realistic environment for coding assistance is providing responsive code completion. Developer expectations for low latencies impact not only user satisfaction and retention, but also directly affect their likelihood to provide preference data. The slower the completions are returned to the user, the *less likely* users are to vote (i.e. users select neither completion) (Figure 3). However, many model providers do not optimize their API endpoints for low-latency use cases, requiring us to explore a sampling strategy that improves our system-wide latency.

Since the Copilot Arena interface shows two code completions together, the slowest completion determines the latency. Thus, given a set of M models $\{1, \ldots, M\}$, we let $F_{\max}(l; i, j)$ denote the cumulative density function (CDF) for the maximum latency between models i and j. Because latencies tend to be long-tailed, we model $F_{\max}(l; i, j)$ as a log-normal CDF with parameters estimated from our historical data. Our objective will then be to minimize the expected latency of the chosen model pair under the distribution induced by our observed data,

$$\mathcal{L}(\theta) = \mathbb{E}_{(i,j)\sim p_\theta, L\sim F_{\max}(l;i,j)}[L], \tag{1}$$

where p_θ is a distribution over model pairs,

$$p_\theta(i, j) = \frac{\exp(\theta_{ij}/\tau)}{\sum_{k<l}\exp(\theta_{kl}/\tau)}. \tag{2}$$

Above, τ is a temperature parameter that interpolates between a latency-optimized distribution and a uniform distribution, allowing us to trade off latency and coverage of unique model pairs. The parameters $\theta \in \mathbb{R}^{\binom{M}{2}}$ are optimized via gradient descent to minimize (Eq. 1). In practice, we set τ to values between 5 and 10 to ensure sufficient coverage. By deploying our algorithm, we observed a decrease in median experienced latency by 33% (from 1.61 to 1.07 seconds) compared to a uniform distribution.

2.3 Model Prompting

During real development processes, developers frequently modify or expand upon existing code which requires models to *infill* between code segments. However, many popular coding models such as GPT-4o or Sonnet 3.5 are instruction-tuned [46] and trained to output text left-to-right autoregressively, rather than to "fill-in-the-middle" (FiM) [15, 17]. In preliminary experiments, we observed poor, essentially unusable performance of instruction-tuned models on FiM tasks. Accordingly, we use offline datasets to improve chat models' infilling capabilities.

Offline Evaluation Set-up. Our set-up uses the HumanEval-infilling dataset [5] which consists of 1640 examples where random spans in a completed portion of code are masked out to simulate FiM behavior. To incorporate prefix and suffix information, we began with several prompt templates from Gong et al. [17] with modifications to align the prompts with chat models (e.g., initial instruction and few-shot examples). The templates capture different ways to encode information about the given code context. For example, prefix-suffix-middle presents the code context in the order of prefix and then suffix, and the LLM is asked to output the middle.

Vanilla performance on FiM tasks. We find that the success of standard prompt templates varies greatly between models. This is not necessarily an indication that models cannot code as clearly many state-of-the-art chat models are proficient coders [20, 27]. Instead, the vast majority of the errors result in formatting issues or duplicate code segments rather than logical errors, indicating that FiM performance is inhibited more by low-level formatting issues than high-level coding capabilities.

Our prompting scheme. While it is not feasible to retrain these models because many of them offer API access only, we explore alternative approaches via prompting to improve chat models' abilities to complete FiM tasks. Specifically, we allow the model to generate code snippets, which is a more natural format, and then post-process the snippets into a FiM completion. Our approach is as follows: the model is prompted with the same prompts as above (e.g. prefix-suffix-middle) but with instructions to begin by repeating a portion of the prefix and similarly end by repeating a portion of the suffix. Then, we remove any portion of the output code that already exists in the input, similar to recent agentic search-replace tools [1]. As shown in Figure 4, we found that, relative to the baseline, our prompting scheme provides robust performance gains for infilling: performance improved in 93% of the conditions. High-performing models improve substantially (e.g., Claude-3.5-Sonnet improves from 56.1% to 73.0%), while initially struggling models improve dramatically (e.g., Llama-3.1-70B from 7.4% to 49%). While offline evaluation is not a perfect metric, we find that these drastic improvements enable these models for FiM tasks.

3 System Deployment

Deployment Details. The Copilot Arena extension is advertised in online open-source communities and made available on the VSCode extension store, where it is free to download. Similar to the set-up employed by Chiang et al. [10] and Lu et al. [30], participants are not compensated for using the extension, as in a traditional user study, but instead receive free access to state-of-the-art models. In addition to logging all preference judgments made by users of Copilot Arena, we also log the latency of each model response, the type of file the user is writing, the prefix and suffix length (characters and tokens), each completion length, which model was in the top versus bottom position, and a unique userID—all of which allows users to utilize the extension without revealing the content of what the user is working on. Given the sensitive nature of programming, we established clear privacy controls to give users the ability to restrict our access to their data. Our data collection process was approved by the Institutional Review Board.

Data collection process. We select 10 state-of-the-art models including open and commercial models, as well as generalist and code-specific models. Across 1642 users, we have served over 4.5 million suggestions and collected 11604 votes over the course of 3 months. Overall, we find that all models received between 2-5K votes, providing sufficient coverage. In general, the median time to vote—the time taken after the completion is displayed to the user—was 7 seconds, suggesting that users did not accept all suggestions immediately and considered both completions.

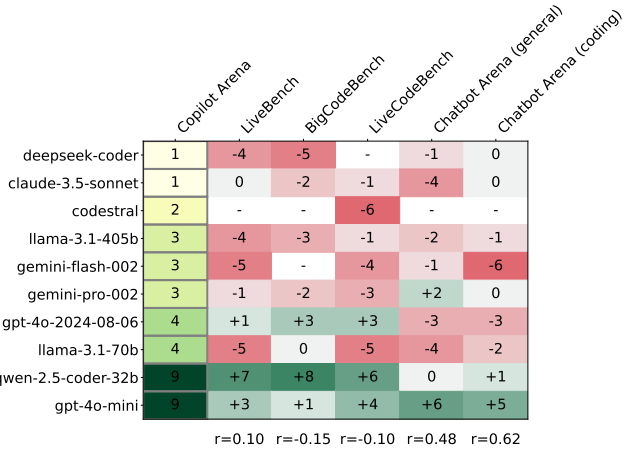

Figure 5: We compare model rankings in Copilot Arena (1st column) to existing evaluations, both static benchmarks (2nd-4th column) and live preference evaluations (last two columns), along with Spearman rank correlation coefficients. For existing evaluations, we show the *change* in rank relative to Copilot Arena rank, with positive values in green denoting models performing better on existing evaluations, negative values in red denoting models performing worse, and a dash indicating that the model is not present in the live leaderboard.

4 Model Rankings

4.1 Copilot Arena Leaderboard

We construct a leaderboard using our user preference judgements. Let n denote the number of judgments and M the number of models. For each battle $i \in [n]$, we define: $X_i \in \{-1, 0, 1\}$: $X_{i,m} = 1$ if model m is presented in the top position, $X_{i,m} = -1$ if presented in the bottom position, and 0 otherwise. The outcome $Y_i \in \{0, 1\}$, where 1 indicates the top model won. Akin to prior work on pairwise preference evaluation [10, 30], we apply a Bradley-Terry (BT) model [6] to estimate the relative strengths of models $\beta \in \mathbb{R}^M$, where the probability p_{ij} that model i beats model j can be modeled as:

$$p_{ij} = \frac{e^{\beta_i}}{e^{\beta_i} + e^{\beta_j}}.$$

We bootstrap the battles in the BT calculation to construct a 95% confidence interval for the rankings, which are used to create a leaderboard that ranks all models, where each model's rank is determined by which other models' lower bounds fall below its upper bound.

Constructing our leaderboard (Figure 5, 1st column), we find that our leaderboard is segmented into multiple tiers based on the estimated β_i values. In the first tier, DeepSeek Coder and Claude Sonnet-3.5 are at the top, with Codestral following closely behind. In general, we observe that code-specific models (e.g., DeepSeek Coder and Codestral) are competitive with general-purpose state-of-the-art models (e.g. Claude Sonnet-3.5), especially if they are trained to infill. In the second tier, there are 5 models of varying sizes and from different model providers that have relatively similar strengths. In the final tier, users preferred two models the least. In

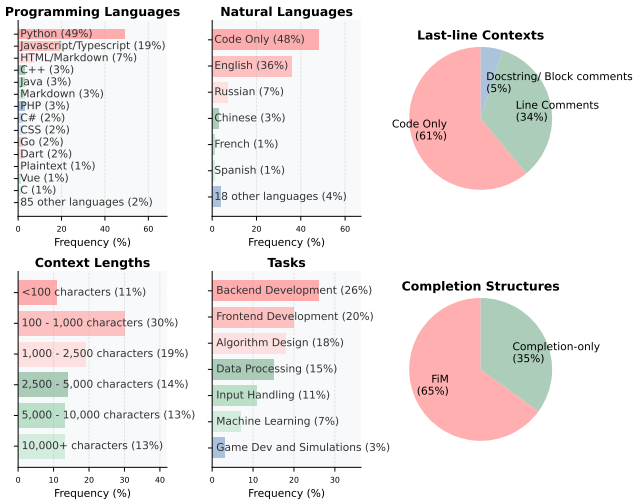

Figure 6: Copilot Arena data is diverse in programming and natural languages, downstream tasks, and code structures.

particular, Qwen-2.5-coder is an exception, performing notably worse than other code-specific models.

4.2 Comparison against prior evaluations

We compare our leaderboard to existing evaluations which encompass both live preference leaderboards with human feedback and static benchmarks (Figure 5, 2nd-5th column). For human preferences, we compare against Chatbot Arena [10] across both the general leaderboard and the coding subset. For static coding benchmarks, we select three that are recent and continue to be maintained (of which we have at least 8 out of 10 overlapping models): LiveBench [47], LiveCodeBench [20], and BigCodeBench [54]. We do not compare to rankings from any user studies because they are difficult to keep updated in comparison to both static benchmarks and live comparative systems.

We find the highest correlation (Spearman's rank correlation (r_s) of 0.62) with Chatbot Arena (coding) [10] and similarly high correlation ($r_s = 0.48$) with Chatbot Arena (general). However, we find a low correlation ($r_s \leq 0.1$) with most static benchmarks. The stronger correlation with human preference evaluations compared to static benchmarks likely indicates that human feedback captures distinct aspects of model performance that static benchmarks fail to measure. We notice that smaller models tend to overperform (e.g., GPT-4o mini and Qwen-2.5-Coder 32B), particularly in static benchmarks. We attribute these differences to the unique distribution of data and tasks that Copilot Arena evaluates over, which we explore in more detail next.

5 Data Analysis

5.1 Exploring Copilot Arena Data

Evaluating models in real user workflows leads to a diverse data distribution in terms of programming and natural languages, tasks, and code structures—e.g., context lengths, last-line contexts, and

Table 1: We compare Copilot Arena with prior evaluations in terms of scale, context length, task type, and code structure. Copilot Arena provides broad coverage across programming languages (PL), natural languages (NL), context length in characters, multiple task types, and structural dimensions—whether the context contains code and fill-in-middle (FiM) tasks are present. Chatbot Arena (code), which is a subset of Chatbot Arena (general), only contains code in 40% and infilling in 2.6% of its input and is denoted by ✓. In Figure 5, we compare against benchmarks that are updated with the latest models (denoted by *).

	Scale		Context Len		Task	Structure	
Benchmark	# PL	# NL	p50	p95	Multi	Code	FiM
Copilot Arena	103	24	1.6k	18k	✓	✓	✓
HumanEval	1	1	0.4k	0.9k	✗	✓	✗
HumanEval-XL	12	23	0.4k	0.9k	✗	✓	✗
SAFIM	4	1	3k	5.9k	✓	✓	✓
LiveCodeBench*	1	1	1.4k	2.5k	✗	✓	✗
LiveBench*	1	1	2.3k	3.9k	✓	✓	✗
BigCodeBench*	1	1	1.1k	1.9k	✓	✓	✗
Chatbot Arena (general)*	≥ 17	≥ 49	0.7k	2.9k	✓	✓	✓
Chatbot Arena (code)*	≥ 17	≥ 39	1.4k	7.8k	✓	✓	✓

completion structures (Figure 6). We discuss how our data distribution compares against those considered in prior evaluations (Table 1).

Programming and natural language: Previous benchmarks such as HumanEval [8] cover a limited number of languages, primarily focusing on Python and English [5, 20, 47, 54]. While recent work such as HumanEval-XL [38] and SAFIM [17] has expanded coverage to up to a dozen programming languages, Copilot Arena covers 103 programming languages which is an order of magnitude more than most other benchmarks. Similarly, while the majority of Copilot Arena users (36%) write in English, we also identify 24 different natural languages which is comparable to Chatbot Arena (general) [10] and benchmarks with multilingual generation [38].

Downstream tasks: Existing benchmarks tend to source problems from coding competitions [20, 47], handwritten programming challenges [8], or from a curated set of GitHub repositories [17]. In contrast, Copilot Arena users are working on a diverse set of realistic tasks, including but not limited to frontend components, backend logic, and ML pipelines. Coding style problems (i.e., algorithm design) comprise a much smaller portion—18%—of Copilot Arena's data. Further, the distribution of downstream tasks for our in-editor suggestions differs from questions raised by chat conversations, e.g., in Chatbot Arena [10], where coding questions also focus on code explanation or suggesting commands.

Code structures and context lengths: Most coding benchmarks follow specific structures, e.g., taking structured docstrings as input [8, 20, 47, 54] or infilling tasks [5, 17]. This means that most benchmarks have relatively short context lengths (e.g., all HumanEval [8] problems are less than 2k characters). Similarly, Chiang et al. [10] focuses on natural language input collected from chat conversations, with many prompts not including any code context (e.g., 40% of Chatbot Arena's coding tasks contain code context and only 2.6% focus on infilling). Input prompts are also relatively

short, with 95% of prompts falling between 1-3k characters. Unlike any existing evaluation, Copilot Arena is structurally diverse, comprising a mixture of infilling versus code completion and forms of docstring tasks. Since users are working in actual IDEs, they work on significantly longer inputs: the median context length is around 1.6k characters and 95% of inputs fall within 18k characters.

5.2 Understanding User Preferences of Code

Given our diversity of input features, we evaluate how each impacts user preference. We partition each feature into contrasting subsets (e.g. FiM vs non-FiM), which we refer to as X and $\tilde{X}$. For each subset, we compute the win-rate[1] matrix $W \in \mathbb{R}^{M \times M}$ where $W(X)$ represents the win-rate matrix of subset X. For each feature, we compute a win-rate difference matrix $\Delta \in \mathbb{R}^{M \times M}$, which represents the number of substantial differences in the win-rate between $W(X)$ and $W(\tilde{X})$.

$$\Delta_{i,j} = \mathbb{1}\left[(W_{i,j}(X) - W_{i,j}(\tilde{X})) > \epsilon\right]$$

In our analysis, substantial changes are those in the top 90th percentile of win-rate changes ($\epsilon = 0.166$). Since $M = 10$, the maximum amount of significant changes is 90 ($|\Delta| \leq 90$).

We compute Δ for four input features—task type, context length, FiM, and programming language—where contrasting strata are present in sufficient quantity ($\geq 10\%$) within our dataset. We stratify the data as follows: For tasks, we compare frontend/backend against algorithm design. For context length, we compare the top 20% against the bottom 20%. For FiM, we compare FiM against completion only. For programming languages, we compare all other programming languages against Python. We stratify these input features to highlight differences between the data distribution in Copilot Arena compared to static benchmarks (Table 1), where a positive win-rate indicates increased model performance on data considered out of the distribution of typical static benchmarks.

Downstream task significantly affects win-rate, while programming languages have little effect. Changing task type significantly affects relative model performance, with 28 significant win-rate changes (31.1% of all possible changes). This gap may indicate that certain models are overexposed to competition-style algorithmic coding problems. On the other hand, the effect of programming language on win-rates was remarkably small, resulting in only 6 (6.6%) significant changes, meaning that models that perform well on Python will likely perform well on another language. We hypothesize that this is because of the inherent similarities between programming languages, and learning one improves performance in another, aligning with trends reported in prior work [38]. Context length and FiM have moderate effects to win-rate, which lead to 16 (17.8%) and 14 (15.6%) significant changes respectively.

Smaller models tend to perform better on data similar to static benchmarks, while the performance of larger models is mixed. For example, Qwen-2.5 Coder performs noticeably worse on frontend/backend tasks (-2), longer contexts (-3), and non-Python settings (-2). We observe similar trends for the two other small models (Gemini Flash and GPT-4o mini) across multiple features. We hypothesize that overexposure may be particularly problematic for smaller models. On the other hand, performance amongst larger models is mixed. For example, Gemini-1.5 Pro performs noticeably better (+3) on long context which aligns with its goal of long context understanding [43]. However, Llama-3.1 405B underperforms on frontend/backend tasks (-4).

[1]Inspecting win-rates helps circumvent potential issues that may arise from applying BT regression to slices with fewer votes.

Figure 7: Significant win-rate changes (Δ) as a result of different data partitions: frontend/backend versus algorithmic problems, long versus short contexts, FiM vs non-FiM, non-Python vs Python. We report the number of positive and negative changes (e.g., +1/-2 means that a model improved over 1 model and worsened against 2 models). In general, we observe the largest percentage of total changes as a result of differences in task (e.g., frontend/backend versus algorithmic problems), while the smallest effects as a result of differences in programming language.

6 Related Work

Human Preferences for Evaluations. A diverse set of human preferences—including binary preferences [3], fine-grain feedback [25, 48], and natural language [42]—are increasingly used for training and fine-tuning LLMs [37]. Preferences are also important for human-centric evaluation, especially as LLMs are deployed in contexts that involve human interaction. Platforms like Chatbot Arena [10] and Vision Arenas [11, 30] provide a way for users to interact with LLMs and provide paired preference judgments. However, existing arenas lack integration into actual user environments to reflect the diverse data that may appear in a user's workflow. We study the use case of LLMs as coding assistants and introduce Copilot Arena to ground preference evaluations in a developer's working environment.

Evaluations of LLM Coding Capabilities. Static benchmarks, e.g., HumanEval [8] and MBPP [2], largely focusing on interview-style programming problems have been the most commonly used to evaluate coding capabilities [7, 13, 22, 24, 28, 29, 33, 35, 45, 49, 51, 53], measured using `pass@k`. Recent benchmarks aim to create more realistic problems, which include multi-turn program evaluations [35] and repository-level challenges [21, 22], and create live benchmarks that reduce contamination risks [20, 47]. Our evaluation platform complements the existing suite of benchmarks by contextualizing model evaluations in an actual user's workflow as coding assistants, measuring a model's quality based on user preferences. Preference data retains signal when models output slightly incorrect, but still

useful answers as opposed to a strict or all or nothing when evaluating using test cases.

A growing set of user studies aim to study human interactions with LLMs [26], particularly how programmers use LLM assistance for software development [4, 9, 31, 34, 39, 40, 44]. A notable work by Cui et al. [12] conducted a field study on GitHub Copilot with many users. However, these studies generally face challenges of scale in terms of the number of users and the models considered, primarily relying on commercial tools like GitHub Copilot or ChatGPT. Mozannar et al. [32] conducted a study to evaluate six different LLMs of varying performance and Izadi et al. [19] similarly conducted a study with three different LLMs, but the models evaluated in both studies are no longer considered state-of-the-art. Our platform aims to address these challenges by building and deploying an actual coding assistant that allows for scalable and adaptable evaluation as new models emerge.

7 Limitations

Although we have a diverse set of users and use cases, it is unclear to what extent our results encapsulate all real-world use cases. We run extensive pilot tests to ensure platform usability, but we recognize that certain aspects—specifically our pairwise completions and slower latency—do not perfectly mirror real-world platforms such as Github Copilot. Further, while we rank models based on user preferences, this should not be treated as the sole defining metric of model quality, but instead an informative one. In this work, we evaluate multiple LLMs with strong coding capabilities; however, we are unable to include Github Copilot because the model powering Github Copilot is not available via API. Finally, due to privacy considerations, we choose not to release all code contexts collected in the study without careful post-processing. We strive to make more data open through periodic releases.

References

[1] Anthropic. 2024. Raising the bar on SWE-bench Verified with Claude 3.5 Sonnet. https://www.anthropic.com/research/swe-bench-sonnet

[2] 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* (2021).

[3] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862* (2022).

[4] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[5] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255* (2022).

[6] Ralph Allan Bradley and Milton E Terry. 1952. Rank analysis of incomplete block designs: I. The method of paired comparisons. *Biometrika* 39, 3/4 (1952), 324–345.

[7] 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. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* (2023).

[8] 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* (2021).

[9] Valerie Chen, Alan Zhu, Sebastian Zhao, Hussein Mozannar, David Sontag, and Ameet Talwalkar. 2024. Need Help? Designing Proactive AI Assistants for Programming. *arXiv preprint arXiv:2410.04596* (2024).

[10] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. 2024. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference. *arXiv preprint arXiv:2403.04132* (2024).

[11] Christopher Chou, Lisa Dunlap, Koki Mashita, Krishna Mandal, Trevor Darrell, Ion Stoica, Joseph E. Gonzalez, and Wei-Lin Chiang. 2024. VisionArena: 230K Real World User-VLM Conversations with Preference Labels. (2024). arXiv:2412.08687 [cs.LG] https://arxiv.org/abs/2412.08687

[12] Kevin Zheyuan Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. 2024. The Productivity Effects of Generative AI: Evidence from a Field Experiment with GitHub Copilot. (2024).

[13] Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. 2023. Large language models of code fail at completing code with potential bugs. *Advances in Neural Information Processing Systems* 36 (2023).

[14] Yann Dubois, Chen Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy S Liang, and Tatsunori B Hashimoto. 2023. Alpacafarm: A simulation framework for methods that learn from human feedback. *Advances in Neural Information Processing Systems* 36 (2023).

[15] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999 [cs.SE] https://arxiv.org/abs/2204.05999

[16] Github. 2022. GitHub copilot - your AI pair programmer. https://github.com/features/copilot

[17] Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. 2024. Evaluation of LLMs on Syntax-Aware Code Fill-in-the-Middle Tasks. *arXiv preprint arXiv:2403.04814* (2024).

[18] Ben Hutchinson, Negar Rostamzadeh, Christina Greer, Katherine Heller, and Vinodkumar Prabhakaran. 2022. Evaluation gaps in machine learning practice. In *Proceedings of the 2022 ACM conference on fairness, accountability, and transparency*. 1859–1876.

[19] Maliheh Izadi, Jonathan Katzy, Tim van Dam, Marc Otten, Razvan Mihai Popescu, and Arie van Deursen. 2024. Language Models for Code Completion: A Practical Evaluation. arXiv:2402.16197 [cs.SE] https://arxiv.org/abs/2402.16197

[20] 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* (2024).

[21] Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. 2024. R2E: Turning any Github Repository into a Programming Agent Environment. In *Forty-first International Conference on Machine Learning*.

[22] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*.

[23] Sayash Kapoor, Benedikt Stroebl, Zachary S Siegel, Nitya Nadgir, and Arvind Narayanan. 2024. AI agents that matter. *arXiv preprint arXiv:2407.01502* (2024).

[24] 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* (2023).

[25] Hannah Rose Kirk, Alexander Whitefield, Paul Röttger, Andrew Michael Bean, Katerina Margatina, Rafael Mosquera, Juan Manuel Ciro, Max Bartolo, Adina Williams, He He, Bertie Vidgen, and Scott A. Hale. 2024. The PRISM Alignment Dataset: What Participatory, Representative and Individualised Human Feedback Reveals About the Subjective and Multicultural Alignment of Large Language Models. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=DFr5hteojx

[26] Mina Lee, Megha Srivastava, Amelia Hardy, John Thickstun, Esin Durmus, Ashwin Paranjape, Ines Gerard-Ursin, Xiang Lisa Li, Faisal Ladhak, Frieda Rong, Rose E Wang, Minae Kwon, Joon Sung Park, Hancheng Cao, Tony Lee, Rishi Bommasani, Michael S. Bernstein, and Percy Liang. 2023. Evaluating Human-Language Model Interaction. *Transactions on Machine Learning Research* (2023). https://openreview.net/forum?id=hjDYJUn9l1

[27] Bill Yuchen Lin, Yuntian Deng, Khyathi Chandu, Faeze Brahman, Abhilasha Ravichander, Valentina Pyatkin, Nouha Dziri, Ronan Le Bras, and Yejin Choi. 2024. WILDBENCH: Benchmarking LLMs with Challenging Tasks from Real Users in the Wild. *arXiv preprint arXiv:2406.04770* (2024).

[28] 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. *Advances in Neural Information Processing Systems* 36 (2023).

[29] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation.

In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*. https://openreview.net/forum?id=6lE4dQXaUcb

[30] Yujie Lu, Dongfu Jiang, Wenhu Chen, William Yang Wang, Yejin Choi, and Bill Yuchen Lin. 2024. WildVision: Evaluating Vision-Language Models in the Wild with Human Preferences. *arXiv preprint arXiv:2406.11069* (2024).

[31] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–16.

[32] Hussein Mozannar, Valerie Chen, Mohammed Alsobay, Subhro Das, Sebastian Zhao, Dennis Wei, Manish Nagireddy, Prasanna Sattigeri, Ameet Talwalkar, and David Sontag. 2024. The RealHumanEval: Evaluating Large Language Models' Abilities to Support Programmers. *arXiv preprint arXiv:2404.02806* (2024).

[33] Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. OctoPack: Instruction Tuning Code Large Language Models. In *The Twelfth International Conference on Learning Representations*.

[34] Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, Nachiappan Nagappan, and Peter C Rigby. 2024. AI-assisted Code Authoring at Scale: Fine-tuning, deploying, and mixed methods evaluation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1066–1085.

[35] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=iaYcJKpY2B_

[36] Michael Oberst, Davis Liang, and Zachary C. Lipton. 2024. The Science of AI Evaluation at Abridge. https://www.abridge.com/ai/science-ai-evaluation.

[37] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[38] Qiwei Peng, Yekun Chai, and Xuhong Li. 2024. HumanEval-XL: A Multilingual Code Generation Benchmark for Cross-lingual Natural Language Generalization. *arXiv preprint arXiv:2402.16694* (2024).

[39] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590* (2023).

[40] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 491–514.

[41] Michael Saxon, Ari Holtzman, Peter West, William Yang Wang, and Naomi Saphra. 2024. Benchmarks as Microscopes: A Call for Model Metrology. In *First Conference on Language Modeling*. https://openreview.net/forum?id=bttKwCZDkm

[42] Jérémy Scheurer, Jon Ander Campos, Jun Shern Chan, Angelica Chen, Kyunghyun Cho, and Ethan Perez. 2022. Training language models with language feedback. *arXiv preprint arXiv:2204.14146* (2022).

[43] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, Soroosh Mariooryad, Yifan Ding, Xinyang Geng, Fred Alcober, Roy Frostig, and Mark Omernick and. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv:2403.05530 [cs.CL] https://arxiv.org/abs/2403.05530

[44] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–7.

[45] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2023. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 13818–13843.

[46] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. 2022. Finetuned Language Models are Zero-Shot Learners. In *International Conference on Learning Representations*.

[47] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, et al. 2024. Livebench: A challenging, contamination-free llm benchmark. *arXiv preprint arXiv:2406.19314* (2024).

[48] Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A Smith, Mari Ostendorf, and Hannaneh Hajishirzi. 2023. Fine-grained human feedback gives better rewards for language model training. *Advances in Neural Information Processing Systems* 36 (2023), 59008–59033.

[49] 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* (2023).

[50] Eugene Yang, Roshanak Omrani, Evan Curtin, Tara Emory, Lenora Gray, Jeremy Pickens, Nathan Reff, Cristin Traylor, Sean Underwood, David D Lewis, et al. 2024. Beyond the Bar: Generative AI as a Transformative Component in Legal Document Review. In *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 4779–4788.

[51] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2024. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems* 36 (2024).

[52] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=uccHPGDlao

[53] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022. XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence. arXiv:2206.08474 https://arxiv.org/abs/2206.08474

[54] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. 2024. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. arXiv:2406.15877 [cs.SE] https://arxiv.org/abs/2406.15877

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009