CoCo-Bench: A Comprehensive Code Benchmark For Multi-task Large Language Model Evaluation

Anonymous ACL submission

Abstract

Large language models (LLMs) play a crucial role in software engineering, excelling in tasks like code generation and maintenance. However, existing benchmarks are often narrow in scope, focusing on a specific task and lack a comprehensive evaluation framework that reflects real-world applications. To address these gaps, we introduce CoCo-Bench (Comprehensive Code Benchmark), designed to evaluate LLMs across four critical dimensions: code understanding, code generation, code modification, and code review. These dimensions capture essential developer needs, ensuring a more systematic and representative evaluation. CoCo-Bench includes multiple programming languages and varying task difficulties, with rigorous manual review to ensure data quality and accuracy. Empirical results show that CoCo-Bench aligns with existing benchmarks while uncovering significant variations in model performance, effectively highlighting strengths and weaknesses. By offering a holistic and objective evaluation, CoCo-Bench provides valuable insights to guide future research and technological advancements in code-oriented LLMs, establishing a reliable benchmark for the field.

1 Introduction

In recent years, the application of artificial intelligence in software engineering (AI4SE) (McDermott et al., 2020) has rapidly evolved, from code generation and bug detection to software testing. However, these advancements have exposed the limitations of existing code benchmarks. Many benchmarks fail to comprehensively assess large language models (LLMs), often overestimating their true performance and leading to biased conclusions.

As shown in Table 1, current benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) focus on simple test cases readily available online, making them prone to overfitting by models. Benchmarks such as CoderEval (Zhang et al., 2024b) and ClassEval (Du et al., 2023) target specific LLM capabilities but lack comprehensive coverage for a holistic evaluation. There is an urgent need for an objective, systematic, and comprehensive benchmark to support the continued development of AI4SE.

We propose that an effective benchmark should align closely with real-world scenarios and evaluate LLMs across four key dimensions of programmer capabilities: code understanding (CU, the ability to comprehend existing code), code generation (CG, the ability to generate code based on given context), code modification (CM, the ability to detect error and modify code), and code review (CR, the ability to assess and improve code quality). These four dimensions provide a robust framework for evaluating LLMs, ensuring a detailed and systematic measurement of their performance in practical development contexts.

To address these needs, we introduce CoCo-Bench, a Comprehensive Code Benchmark designed to assess LLMs across four key dimensions. It evaluates a range of programming languages and tasks of varying difficulty, with rigorous manual review to ensure quality and accuracy. Unlike existing benchmarks, CoCo-Bench features innovative task designs, such as reverse reasoning for CU and multi-level code completion for CG, offering a more comprehensive evaluation of model capabilities. CoCo-Bench ensures task diversity and practical alignment by including both simple and complex tasks. This approach prevents overfitting and better reflects model performance in real-world development environments, providing a nuanced framework that identifies strengths and weaknesses across different capability dimensions.

Our empirical results demonstrate that CoCo-Bench not only aligns well with existing benchmarks but also reveals significant variations in model performances across different capability di-

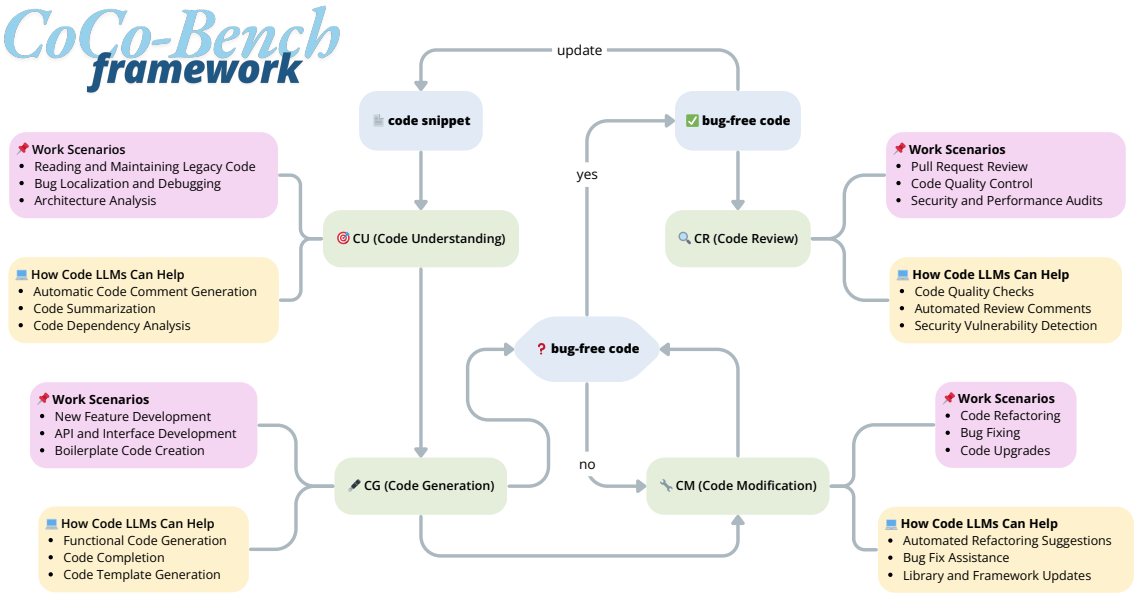

Figure 1: Overview of the core evaluation dimensions in CoCo-Bench. The framework assesses four critical capabilities of code LLMs: code understanding (CU), code generation (CG), code modification (CM), and code review (CR). The evaluation flow highlights the interconnected nature of these capabilities in real-world software development scenarios.

mensions. This effectively points out the strengths and weaknesses of various code LLMs. By offering a more holistic and objective evaluation, CoCo-Bench aims to guide future research, drive technological advancements in the development of code-oriented LLMs, and establish a reliable standard for the field of software engineering.

2 Related Work

Code benchmarks for LLMs have undergone significant evolution recently, reflecting notable advancements in the field. Early works like HumanEval (Chen et al., 2021), Mostly Basic Programming Problems (MPBB) (Austin et al., 2021) and The Code/Natural Language Challenge (CoNaLa) (Yin et al., 2018) focused on fundamental CG tasks. HumanEval evaluated models' ability to generate Python functions using real-world tasks, while MPBB included 974 tasks aimed at entry-level programmers, assessing CG from complex textual descriptions.

Most code benchmarks primarily concentrate on tasks related to CG. APPS (Hendrycks et al., 2021) challenges models to generate Python code from natural language, simulating real-world developer tasks. CoderEval (Zhang et al., 2024b) expanded evaluations by focusing on non-standalone

Code Benchmarks	CU	CG	CM	CR
CoNaLA	✓	✗	✗	✗
Concode	✓	✗	✗	✗
HumanEval	✗	✓	✗	✗
MBPP	✓	✗	✗	✗
APPS	✓	✗	✗	✗
PandasEval	✓	✗	✗	✗
NumpyEval	✓	✗	✗	✗
AixBench	✓	✗	✗	✗
ClassEval	✓	✗	✗	✗
CoderEval	✓	✗	✗	✗
CodeFuseEval	✓	✓	✗	✗
UltraEval	✗	✓	✗	✗
CodeXGLUE	✓	✓	✓	✗
NaturalCodeBench	✓	✗	✗	✗
CodeScope	✓	✓	✓	✗
Mercury	✓	✗	✗	✗
ENAMEL	✓	✗	✗	✗
CoCo-Bench	✓	✓	✓	✓

Table 1: Task coverage across various code benchmarks. ✓ indicates coverage of certain type of task, while ✗ indicates no coverage.

functions commonly found in open-source projects, providing a platform for assessing functional correctness. ClassEval (Du et al., 2023) introduced a benchmark for class-level CG, addressing gaps in existing evaluations by focusing on more complex tasks. Concode (Iyer et al., 2018) targets generating Java class member functions from English documentation, addressing challenges in class member function generation. CodeXGLUE (Lu et al., 2021) and CodeEditorBench (Guo et al., 2024) advance research in code understanding and editing, evaluating tasks like debugging and requirement switching.

As task complexity increased, new benchmarks emerged to cover a broader range of scenarios. DyPyBench (Bouzenia et al., 2024) is the first comprehensive benchmark for dynamic program analysis of Python projects. SWE-BENCH (Jimenez et al., 2024) evaluates models' ability to generate patches that pass real tests by linking GitHub issues with merged pull requests. CRUXEval (Gu et al., 2024) tests models on practical coding tasks using 800 different Python functions, while Debugbench (Tian et al., 2024) focuses on assessing debugging capabilities, reflecting the need for more complex evaluations beyond CG. LiveCodeBench (Jain et al., 2024) adopts a dynamic approach by continuously sourcing new programming challenges from competitive platforms to evaluate models' real-world capabilities, particularly in code self-repair and test output prediction. CodeMind (Liu et al., 2024) introduces dimensions like independent execution reasoning and specification reasoning to assess models' performance in complex tasks beyond simple CG.

Multi-task benchmarks have also gained importance. CodeFuseEval (Di et al., 2024) combines the standards of HumanEval-x and MPBB, introducing multi-task scenarios like code completion and cross-language translation. UltraEval (He et al., 2024) provides a lightweight, comprehensive framework to assess LLMs across various tasks, offering a unified evaluation platform.

3 Task Definition

In CoCo-Bench, we define four primary tasks to comprehensively evaluate the capabilities of LLMs in software engineering. Each task is formally defined using mathematical notation to provide clarity and precision in assessment. Figure 2 shows some demonstrations.

3.1 Code Understanding (CU)

CU is formalized as a bidirectional inference problem. Let $\mathcal{C}$ denote the set of all possible code snippets, $\mathcal{I}$ represent the set of all possible input parameters, and $\mathcal{O}$ denote the set of all possible outputs. The CU task comprises two functions: $f_{\text{CU}} : \mathcal{C} \times \mathcal{I} \rightarrow \mathcal{O}$ and $f_{\text{CU}}^{-1} : \mathcal{C} \times \mathcal{O} \rightarrow \mathcal{P}(\mathcal{I})$. Here, $f_{\text{CU}}(C, I) = O$ predicts the output O given the code snippet C and input parameters I, and $f_{\text{CU}}^{-1}(C, O) = \mathcal{I}'$ deduces the set of possible input parameters $\mathcal{I}' \subseteq \mathcal{I}$ that could produce the output O when executed with code C.

Formally, for a given code snippet $C \in \mathcal{C}$, input $I \in \mathcal{I}$, and output $O \in \mathcal{O}$, $f_{\text{CU}}(C, I) = O$ if and only if $\texttt{execute}(C, I) = O$, and $f_{\text{CU}}^{-1}(C, O) = \{I' \in \mathcal{I} \mid \texttt{execute}(C, I') = O\}$.

CU emphasizes the model's deep understanding of code logic and its ability to perform both forward and reverse inferences. To ensure wide applicability, the collected code snippets span multiple technical domains, including artificial intelligence and machine learning, data processing and analysis, web development, and database management. For example, in web development and data processing, we utilize key tools such as `PyTorch` (Paszke et al., 2019), `TensorFlow` (Abadi et al., 2016), `Keras` (Chollet, 2015), `Scikit-learn` (Pedregosa et al., 2018), `NumPy` (Harris et al., 2020), `Pandas` (McKinney, 2010), and `Matplotlib` (Hunter, 2007).

Specifically, the CU task consists of two subtasks: predicting the code output (CU_F) and deducing the code input (CU_R). For instance, given the function `def add(a, b): return a + b` and inputs $(3, 5)$, the model should accurately predict the result 8. Conversely, based on the output 8, the model should deduce potential inputs such as $\{(3, 5), (4, 4), (6, 2)\}$. By assessing the model's accuracy in both CU_F and CU_R, we can comprehensively measure its understanding of code logic.

3.2 Code Generation (CG)

CG is defined as the transformation of natural language descriptions into executable code. Let $\mathcal{D}$ represent the set of all possible natural language descriptions, and $\mathcal{C}$ denote the set of all possible code snippets. The CG task is represented by the function $f_{\text{CG}} : \mathcal{D} \rightarrow \mathcal{C}$, where $f_{\text{CG}}(D) = C$ generates the code snippet C corresponding to the natural language description D. Formally, for a given descrip-

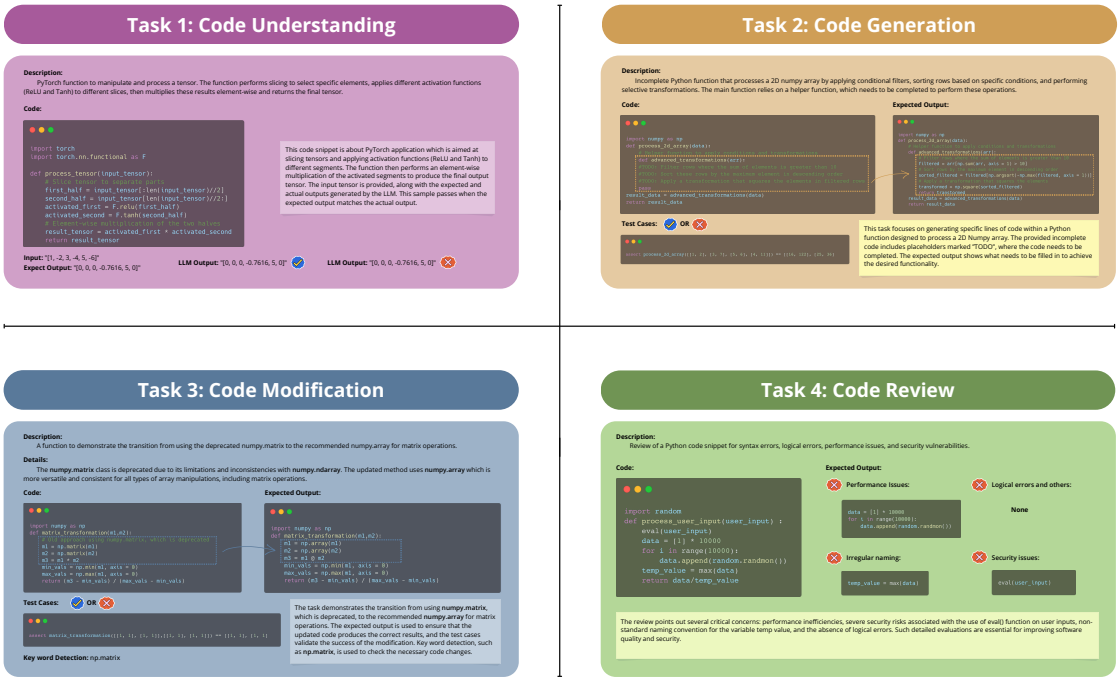

Figure 2: Illustration of the four primary tasks in CoCo-Bench—Code Understanding (CU), Code Generation (CG), Code Modification (CM), and Code Review (CR)—each defined to evaluate the capabilities of large language models (LLMs) in software engineering.

tion $D \in \mathcal{D}$, $C = f_{\text{CG}}(D)$ such that $\texttt{execute}(C)$ performs the task described by D.

This function enables programmers to quickly translate ideas into code and allows non-technical individuals to contribute to software development. By automating routine programming tasks, CG reduces human errors, improves code quality and consistency, and accelerates the product iteration cycle. We design both function-level CG tasks and sentence-level CG tasks to test the models' capabilities.

3.3 Code Modification (CM)

CM involves altering existing code to meet specific requirements or to improve its functionality. Let $\mathcal{C}_{\text{old}}$ represent the set of original code snippets, $\mathcal{R}$ denote the set of all possible modification requests or requirements, and $\mathcal{C}_{\text{new}}$ denote the set of modified code snippets. The CM task is defined by the function $f_{\text{CM}} : \mathcal{C}_{\text{old}} \times \mathcal{R} \to \mathcal{C}_{\text{new}}$, where $f_{\text{CM}}(C_{\text{old}}, R) = C_{\text{new}}$ generates the modified code snippet C_{new} that satisfies the modification request R. Formally, for given $C_{\text{old}} \in \mathcal{C}_{\text{old}}$ and $R \in \mathcal{R}$, $C_{\text{new}} = f_{\text{CM}}(C_{\text{old}}, R)$ such that $\texttt{execute}(C_{\text{new}})$ meets the requirements R.

This task is crucial for maintaining software reliability and adapting to evolving needs. Our samples include code modification, where the model updates code based on error messages, and API updates, where the model revises code to reflect the latest API changes. These scenarios test whether code LLMs can enhance software reliability and keep applications up-to-date with the latest technological advancements.

3.4 Code Review (CR)

CR is structured as a multi-label classification problem, where each code snippet is evaluated across several criteria. Let $\mathcal{C}$ represent the set of all possible code snippets and $\mathcal{E}$ denote the set of evaluation criteria. The CR task is defined by the function $f_{\text{CR}} : \mathcal{C} \to \mathcal{P}(\mathcal{E})$, where $f_{\text{CR}}(C) = \mathcal{E}'$ assigns a subset of evaluation criteria $\mathcal{E}' \subseteq \mathcal{E}$ that the code snippet C satisfies or violates. Formally, for a given code snippet $C \in \mathcal{C}$, $\mathcal{E}' = f_{\text{CR}}(C)$ where $\mathcal{E}' = \{e \in \mathcal{E} \mid C \text{ exhibits characteristic } e\}$.

The key areas of evaluation include security issues, performance problems, adherence to naming conventions, and logical errors. Specifically, security issues involve vulnerabilities such as SQL injections, buffer overflows, and insecure data handling. Performance problems pertain to code seg-

ments that may lead to inefficiency, including unnecessary computations, suboptimal algorithms, and resource-heavy operations. Adherence to naming conventions means the compliance with established coding standards and naming practices, which enhance code readability and maintainability. Logical errors involve flaws that might cause incorrect execution, such as infinite loops, improper condition checks, and erroneous data manipulation.

4 Analysis

In this section, we first analyze the overall scores of all the LLMs. Then, in Section 4.2, we examine the correlations between different sub-tasks and datasets. In Section 4.3, we analyze the impact of context length on model inference. In Section 4.4, we focus on the impact of decoding strategies on the models, primarily including top-k, top-p, and Max New Tokens.

4.1 Comprehensive Performance Evaluation

We present the leaderboard results in Table 2, in which the empirical results reveal that DeepSeek-Coder-V2-Instruct and ChatGPT-4.0 outperform other leading LLMs across various code-related tasks. This superior performance primarily stems from the scaling up of model parameters and the volume of data used during their pre-training processes.

Model scaling plays a pivotal role in enhancing performance. Larger models, such as DeepSeek-Coder-V2-Instruct, consistently outperform smaller models like DeepSeek-Coder-1.3b and DeepSeek-Coder-6.7b. This scaling trend is not uniformly observed across all model families, suggesting that factors such as training data diversity and fine-tuning strategies play critical roles in maximizing the benefits of larger model architectures. For instance, while DeepSeek-Coder-33b-instruct achieves significant performance gains, similar scaling within the CodeLlama series does not yield comparable improvements, pointing to potential inefficiencies or bottlenecks in their respective training paradigms.

The superiority of instruction-tuned models over their base counterparts is consistently observed across both CoCo-Bench and HumanEval benchmarks. This trend underscores the critical importance of instruction fine-tuning in enhancing models' ability to follow complex directives and perform specialized tasks. The enhanced performance of larger instruction-tuned models further suggests that combining scale with targeted fine-tuning yields synergistic benefits, enabling models to achieve higher levels of proficiency in intricate code-related tasks.

4.2 Correlation Analysis

Correlation between tasks: We visualize the Spearman correlation heatmap among the tasks in Figure 3, which provides further insight into the interdependencies between different code-related tasks. Notably, a strong correlation is observed between CU_F and CG (0.94), as well as between CU_R and CG (0.84). These high correlation values indicate that proficiency in code understanding (CU) significantly enhances a model's ability to generate (CG) code effectively. This interconnectedness suggests that improvements in one task—such as CU—can lead to better performance in related tasks like CG. It highlights the integrated nature of these tasks in real-world applications, where advances in one area may catalyze progress in others, ultimately fostering more versatile and capable code models.

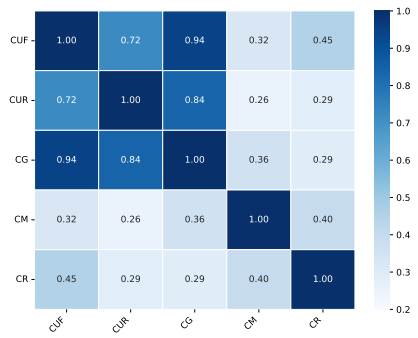

Figure 3: The correlation between each two of the five tasks (CU_F, CU_R, CG, CM, CR) on CoCo-Bench.

However, the relatively lower correlation between CU and CR indicates that CR involves additional competencies beyond mere code comprehension, such as assessing code quality, efficiency, and adherence to best practices. We also observe a relatively low correlation between CG and CM (0.36), suggesting that the performance in CG does not necessarily contribute to CM. This distinction underscores the necessity for models to possess not only a deep understanding of code but also the capability to evaluate and improve it critically, reflecting the multifaceted nature of software engineering.

Model	CU		CG	CM	CR	CoCo-Score
	CU_F	CU_R				
CodeLlama-7b-hf	12.93	3.33	11.45	30.00	39.39	16.87
CodeLlama-7b-Instruct-hf	9.92	5.17	16.41	15.00	25.71	12.94
CodeLlama-13b-hf	19.01	4.17	19.23	20.00	32.26	16.67
CodeLlama-13b-Instruct-hf	13.45	6.78	18.60	25.00	31.43	17.14
CodeLlama-34b-hf	15.83	6.67	21.71	15.00	20.00	14.50
CodeLlama-34b-Instruct-hf	14.88	4.24	19.08	20.00	29.41	15.48
DeepSeek-Coder-1.3b-base	14.88	3.31	15.27	20.00	31.43	14.84
Deepseek-Coder-1.3b-instruct	15.13	5.98	19.85	20.00	25.71	15.62
DeepSeek-Coder-6.7b-base	26.05	5.08	25.78	25.00	28.57	19.68
Deepseek-Coder-6.7b-instruct	35.65	**11.93**	**46.92**	<u>44.44</u>	28.12	30.62
DeepSeek-Coder-33b-base	21.37	8.40	23.81	35.00	32.35	21.91
Deepseek-Coder-33b-instruct	33.88	<u>10.74</u>	39.53	50.00	<u>37.14</u>	<u>31.05</u>
ChatGPT4	<u>53.72</u>	15.70	<u>44.27</u>	15.00	45.71	**32.06**
DeepSeek-R1-Distill-Qwen-7B	**57.02**	9.09	35.88	20.00	34.29	28.26
o1-mini	66.12	9.09	55.73	**45.00**	45.71	39.60

Table 2: Leaderboard of model performance comparison on CoCo-Bench, with the first-place models highlighted in shadow , the second-place models in **bold**, and the third-place models <u>underlined</u>. The table compares the performance across different tasks: **CU**, **CG**, **CM**, and **CR** for five major model series: DeepSeek-Coder, CodeLlama, R1, GPT and o1. $\mathbf{CU_F}$ and $\mathbf{CU_R}$ are two kinds of sub-tasks of **CU**. The metric CoCo-Score (see C) provides an aggregated evaluation of model capabilities.

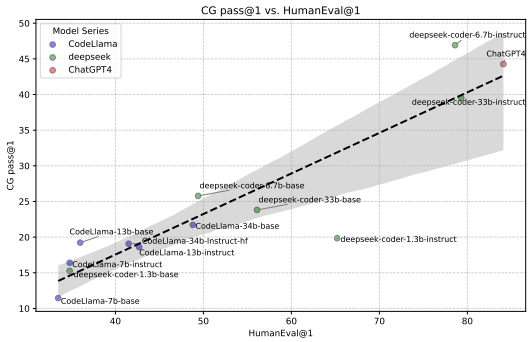

Figure 4: Comparative analysis of model performance on CoCo-Bench and HumanEval benchmarks: This figure illustrates the relationship between model performance on the CoCo-Bench (CG pass@1) and HumanEval (HumanEval@1) benchmarks. Colors distinguish the training methods, with blue representing base models and orange representing instruct-tuned models. The size of each point reflects the model's parameter count, where larger points correspond to larger models. Different shapes indicate the model family: circles for CodeLlama, crosses for DeepSeek-Coder and plus signs for ChatGPT models. The dashed trend line and shaded area indicate the general correlation between performance on the two benchmarks.

Correlation between datasets: Figure 4 and Figure 3 collectively demonstrate that CoCo-Bench provides a more stringent and discriminative evaluation compared to simpler benchmarks like HumanEval. The positive correlation between CoCo-Bench and HumanEval scores indicates that foundational capabilities are consistent across benchmarks. Moreover, the analysis reveals that increasing model size generally correlates with enhanced performance, particularly within the DeepSeek-Coder series.

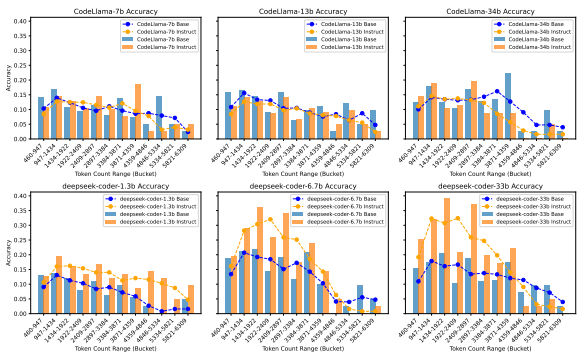

Figure 5: Performance differences between instruct and base versions of the same models on CoCo-Bench

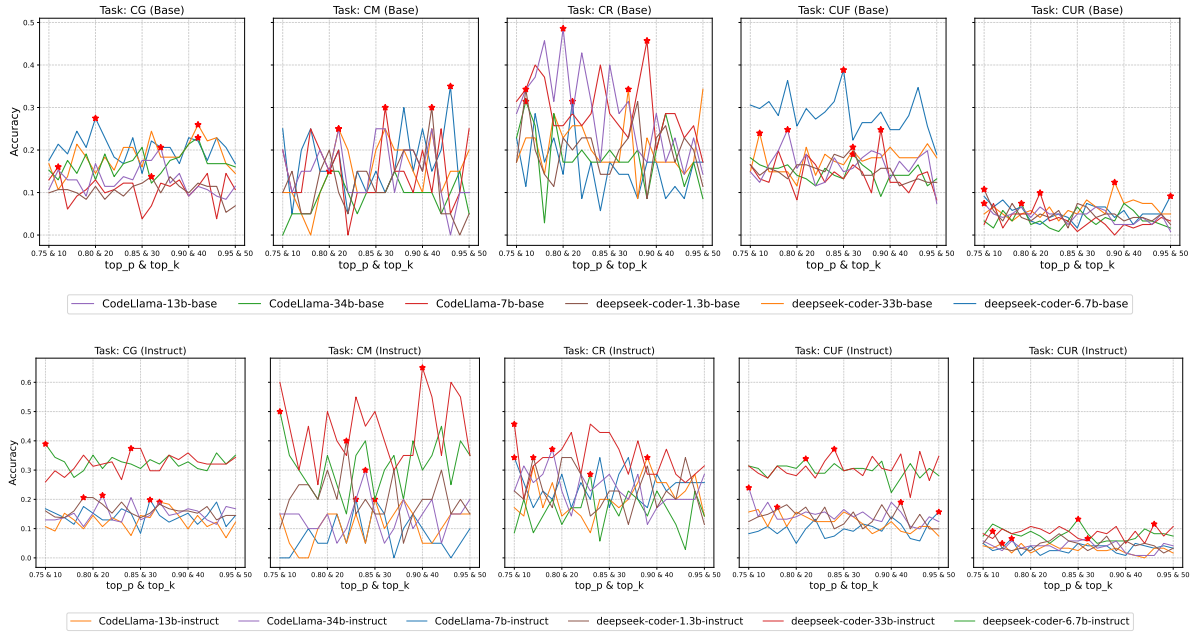

Figure 6: Performance differences under various `top_p` and `top_k` configurations

4.3 Context Length and Model Performance

Figure 8 and Figure 5 illustrate how model accuracy varies across different token count ranges of input. All models, regardless of size and type, face challenges in maintaining attention across longer contexts due to the quadratic cost of self-attention in transformers. Instruction tuning likely helps focus attention on task-relevant segments of the input, mitigating some of this drop in performance for instruction models.

4.4 Decoding Strategies

Top-p and Top-k: As is illustrated in Figure 6, top-p and top-k sensitivity varies by task. Structured tasks like CU tasks benefit from deterministic outputs since code has stricter correctness requirements. A narrower token distribution (lower top-p and top-k) is often sufficient for accurate results. In contrast, open-ended tasks like CM and CG require creativity and diversity from LLMs, as there are multiple plausible ways to modify or improve code. It's worth mentioning that LLMs are more sensitive to top-p and top-k variance as they generate a wider range of plausible token predictions due to their richer token distribution and greater expressive power. When using high top-p and top-k for inference, the outputs of LLMs can become overly diverse or less coherent, potentially introducing syntactic errors or irrelevant code snippets. Smaller models, by contrast, have less expressive token distributions, making them inherently more deterministic and less impacted by high top-p and top-k settings, thereby maintaining code correctness more effectively. So, we can draw the following empirical conclusion:

Conclusion 1. *The sensitivity to top-p and top-k parameters varies across different tasks. Structured tasks, such as CU, tend to benefit from more deterministic outputs, while open-ended tasks, such as CM and CG, demand greater creativity and diversity.*

Instruction models consistently outperform base models in code-related tasks. We believe this is due to the exposure of instruction models to examples with task-specific instructions during training, enabling them to better understand and follow coding guidelines and requirements. As a result, we can derive the following empirical conclusion:

Conclusion 2. *Instruction-tuned models excel in code-related tasks due to their close alignment with task-specific coding objectives, which significantly enhances both performance and robustness across various decoding configurations.*

Medium token ranges align well with the pre-training datasets and tokenization schemes for most models. Shorter tokens (e.g., 460–1191) may lead to sparse representation, while longer tokens may overload the model's capacity. It's also noted how scaling laws reconcile with the outcome. Larger

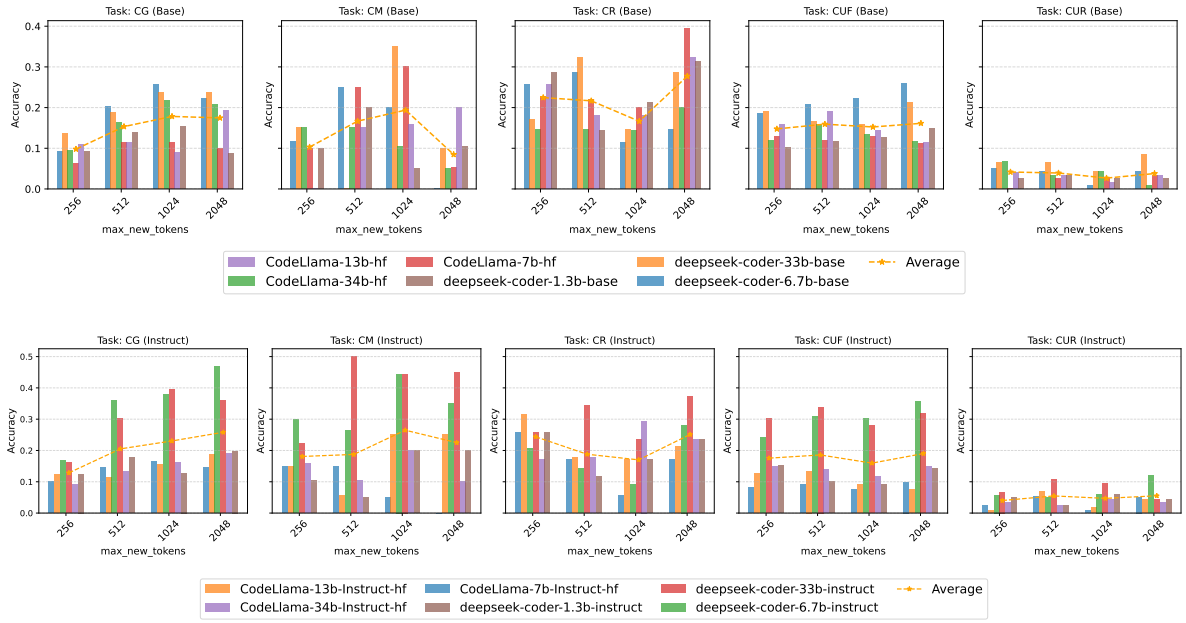

Figure 7: Performance differences under various `max_new_tokens` Configurations

models are inherently better at capturing long-range dependencies due to their larger parameter space and richer latent representations. Smaller models lack the capacity to encode such complexity effectively. This leads us to the following empirical conclusion:

Conclusion 3. *LLMs encounter challenges with longer code contexts, but instruction tuning helps maintain performance by focusing on relevant code segments. Larger models handle long-range dependencies in code more effectively than smaller ones.*

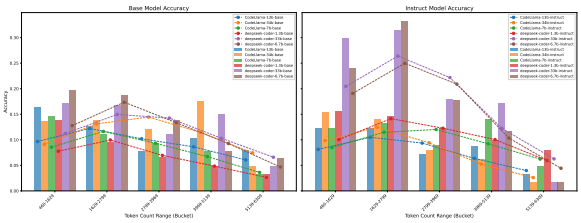

Figure 8: Performance differences between different models of the same type on CoCo-Bench

Max New Tokens: Figure 7 shows that higher max new tokens directly correlates with better accuracy in tasks requiring coherent and extended outputs (e.g., CG and CU). It allows the model to generate longer sequences, potentially capturing more context and completing more complex outputs. Some tasks like CM benefit less due to their inherent requirements, where longer sequences add minimal value. Comparably, larger models and instruction-tuned models are better equipped to utilize extended token generation due to their ability to manage more extensive contexts and dependencies. Based on this, the following empirical conclusion can be drawn:

Conclusion 4. *Increasing max new tokens improves performance in code tasks that require extended and coherent outputs, particularly benefiting larger and instruction-tuned models by enabling them to manage more complex code structures and dependencies.*

5 Limitations and Future Work

While CoCo-Bench provides an effective and comprehensive evaluation framework for LLMs in code-related tasks, there are areas for expansion. Currently, the benchmark lacks multimodal tasks that require models to integrate code with other data types, such as images or natural language, which are becoming increasingly relevant in modern development environments. Recognizing this gap, we plan to introduce multimodal tasks in the future, allowing us to evaluate models on more complex projects and providing a more holistic assessment of LLMs' capabilities. To ensure CoCo-Bench remains at the cutting edge of LLM evaluation, we will regularly update the benchmark by incorporating new programming languages and adapting to evolving development practices.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *Preprint*, arXiv:1603.04467.

Open AI. 2024. Learning to reason with llms.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Islem Bouzenia, Bajaj Piyush Krishan, and Michael Pradel. 2024. Dypybench: A benchmark of executable python software. *Proceedings of the ACM on Software Engineering*, 1(FSE):338–358.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

François Chollet. 2015. keras. `https://github.com/fchollet/keras`.

Peng Di, Jianguo Li, Hang Yu, Wei Jiang, Wenting Cai, Yang Cao, Chaoyu Chen, Dajun Chen, Hongwei Chen, Liang Chen, Gang Fan, Jie Gong, Zi Gong, Wen Hu, Tingting Guo, Zhichao Lei, Ting Li, Zheng Li, Ming Liang, Cong Liao, Bingchang Liu, Jiachen Liu, Zhiwei Liu, Shaojun Lu, Min Shen, Guangpei Wang, Huan Wang, Zhi Wang, Zhaogui Xu, Jiawei Yang, Qing Ye, Gehao Zhang, Yu Zhang, Zelin Zhao, Xunjin Zheng, Hailian Zhou, Lifu Zhu, and Xianying Zhu. 2024. Codefuse-13b: A pretrained multilingual code large language model. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '24. ACM.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A code efficiency benchmark for code large language models. *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. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *Preprint*, arXiv:2308.01861.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *Preprint*, arXiv:2401.03065.

Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi LI, Ruibo Liu, Yue Wang, Shuyue Guo, Xingwei Qu, Xiang Yue, Ge Zhang, Wenhu Chen, and Jie Fu. 2024. Codeeditorbench: Evaluating code editing capability of large language models. *Preprint*, arXiv:2404.03543.

Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. Aixbench: A code generation benchmark dataset. *Preprint*, arXiv:2206.13179.

Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with numpy. *Nature*, 585(7825):357–362.

Chaoqun He, Renjie Luo, Shengding Hu, Yuanqian Zhao, Jie Zhou, Hanghao Wu, Jiajie Zhang, Xu Han, Zhiyuan Liu, and Maosong Sun. 2024. Ultraeval: A lightweight platform for flexible and comprehensive evaluation for llms. *Preprint*, arXiv:2404.07584.

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. *Preprint*, arXiv:2105.09938.

J. D. Hunter. 2007. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *Preprint*, arXiv:1808.09588.

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. *Preprint*, arXiv:2403.07974.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. 2024. Codemind: A framework to challenge large language models for code reasoning. *Preprint*, arXiv:2402.09664.

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. *Preprint*, arXiv:2102.04664.

Tom McDermott, Dan DeLaurentis, Peter Beling, Mark Blackburn, and Mary Bone. 2020. Ai4se and se4ai: A research roadmap. *Insight*, 23(1):8–14.

Wes McKinney. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, pages 51 – 56.

Tom Minka, Ryan Cleven, and Yordan Zaykov. 2018. Trueskill 2: An improved bayesian skill rating system. *Technical Report*.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. *Preprint*, arXiv:1912.01703.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel Nothman, Gilles Louppe, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2018. Scikit-learn: Machine learning in python. *Preprint*, arXiv:1201.0490.

Ruizhong Qiu, Weiliang Will Zeng, Hanghang Tong, James Ezick, and Christopher Lott. 2024. How efficient is llm-generated code? a rigorous & high-standard benchmark. *Preprint*, arXiv:2406.06647.

Charles Spearman. 1961. The proof and measurement of association between two things.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. 2024. Debugbench: Evaluating debugging capability of large language models. *Preprint*, arXiv:2401.04621.

Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Hari Sundaram, and Shuiguang Deng. 2024. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. *Preprint*, arXiv:2311.08588.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. *Preprint*, arXiv:1805.08949.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. Cert: Continual pre-training on sketches for library-oriented code generation. *Preprint*, arXiv:2206.06888.

Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. 2024a. Naturalcodebench: Examining coding performance mismatch on humaneval and natural user prompts. *Preprint*, arXiv:2405.04520.

Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. 2024b. Learning-based widget matching for migrating gui test cases. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24. ACM.

A Data Statistics Details

A.1 Data Statistics

CoCo-Bench, meticulously curated through fine-grained manual review, comprises 705 high-quality samples developed with the assistance of several seasoned developers, each with over 10 years of experience in SE. Half of CoCo-Bench samples are quite challenging. As shown in Figure 9, The tasks are diversified, with 56.7% of the samples focused on CU, 21.3% on CG, 17.0% on CM, and 5.0% on CR. This distribution emphasizes a comprehensive approach to testing the various aspects of code comprehension and generation.

In addition to the challenging nature of the tasks, CoCo-Bench also supports a wide range of programming languages. Python leads the way, accounting for 54.6% of the samples, followed by Java at 20.4%, C++ at 19.0%, and SQL at 6.0%. The diversity ensures that the models are evaluated across a broad spectrum of real-world coding scenarios.

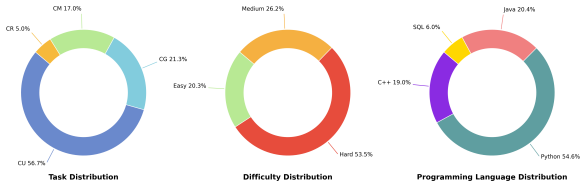

Figure 9: Distribution of tasks, difficulty levels, and programming languages in CoCo-Bench.

	Task Types				
Difficulty	CU	CG	CM	CR	Total
Easy	74	41	17	11	143
Medium	94	54	22	15	185
Hard	232	55	81	9	377
Total	400	150	120	35	705

Table 3: Distribution of samples by difficulty levels. This table presents the distribution of samples across different difficulty levels in the dataset. The tasks are categorized into four types: CU, CG, CU and CR. Each sample is further divided into easy, medium and hard difficulty levels.

The dataset, as summarized in Table 3, shows a clear emphasis on hard examples, which constitute the majority of the dataset with 377 tasks. Additionally, as shown in Table 4, each task type is well-represented across different languages, with Python being the most frequently used language across all categories.

	Task Types				
Code Language	CU	CG	CM	CR	Total
python	208	87	55	35	385
java	80	29	35	\	144
C++	70	34	30	\	134
SQL	42	\	\	\	42
Total	400	150	120	35	705

Table 4: Distribution of task types by programming languages. Each row represents a programming language, such as Python, Java, C++ and SQL, and lists the number of tasks in each category for that language. The total number of tasks for each language is provided, along with the overall total for all languages.

B Benchmark Construction

The sample generation process of CoCo-Bench is strategically organized into three main stages, as shown in Figure 10: raw data collection, task-specific data transformation and sample review. Each part contributes to the overall effectiveness and quality of the datasets.

Raw Data Collection: The code collection phase of our dataset construction pipeline is designed to gather a diverse set of programming samples that adhere to strict timeliness criteria, preventing data contamination and ensuring the quality and relevance of the data for training and validating machine learning models. We source code from Leetcode and various project repositories, selected under stringent conditions to ensure data freshness. Leetcode provides a controlled environment with frequently reviewed and updated code, incorporating current coding practices and algorithms. This is crucial for training models that must stay up-to-date with the latest programming trends. Project code, particularly from repositories with contributions from less experienced developers, serves a dual purpose. It captures common coding errors and suboptimal practices, valuable for training models aimed at CR and CM tasks. Additionally, it aligns the dataset with real-world programming tasks, as project code often involves complex, operationally-driven designs. By carefully selecting sources like Leetcode and project repositories, we ensure that our dataset avoids outdated or irrelevant data while encompassing a broad spectrum of real-world coding scenarios.

Task-Specific Data Transformation: The sec-

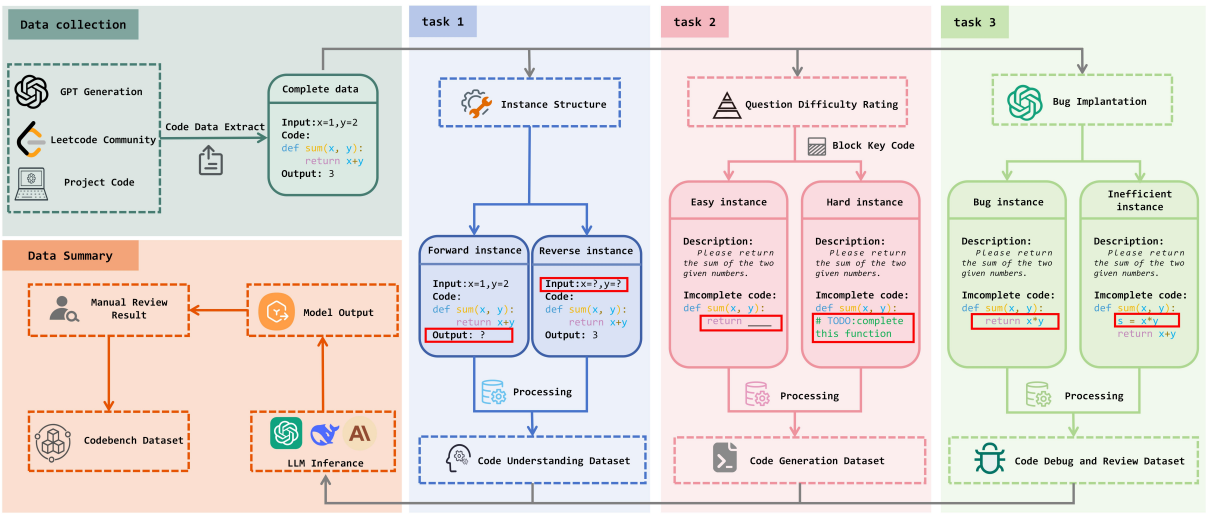

Figure 10: The construction pipeline of CoCo-Bench. The process starts with data collection from three primary sources: GPT generation, Leetcode community solutions, and various projects. The collected data is then structured into samples, which are adapted to different tasks based on their characteristics. **CU:** the construction of this dataset involves structuring the collected data into forward and reverse instances. **CG:** this dataset involves the question difficulty rating task, where the collected code is categorized into easy and hard instances. Easy instances are characterized by simple, straightforward tasks with minimal code, whereas hard instances involve more complex tasks with additional comments or incomplete parts that require completion. **CM** and **CR** datasets start with a bug implantation process, where code snippets are intentionally altered to introduce bugs and inefficiencies. The final stage of the pipeline involves inference and human check which ensure the correctness and relevance of CoCo-Bench.

ond step in our dataset construction pipeline involves task-specific data transformation, tailoring code samples for different computational tasks to optimize model training and evaluation. For CU, the process creates forward instances (input-output pairs) and reverse instances (output-inference challenges). In CG tasks, instances are categorized by difficulty, ranging from simple completions to complex corrections. For CM and CR tasks, bugs and inefficiencies are strategically inserted into code samples, using models like GPT to generate realistic scenarios, enhancing the dataset's robustness for debugging and optimization tasks. The final step includes a thorough CR, starting with initial evaluations using high-performance open-source models like GPT, followed by manual review, and final incorporation into the benchmark library. Following the automated inference, the samples undergo a detailed manual review by experienced developers. The manual review focuses on several critical aspects:

1. **Sample Correctness**: Verifying that the sample code performs the intended function without errors. This involves running the code against a set of test cases to ensure accurate outputs.

2. **Reasonable Difficulty**: Assessing whether the difficulty level of the sample is appropriate. Ensuring that the samples are neither too easy nor excessively difficult, thereby maintaining a balanced difficulty level across the dataset.

3. **Practical Applicability**: Evaluating the real-world relevance and usefulness of the sample. Ensuring that the code samples reflect practical scenarios and challenges that developers are likely to encounter.

4. **Easy Readability**: Checking if the sample code is easy to understand and maintain. This includes verifying clear and concise variable names, appropriate use of comments, and adherence to coding standards.

During the manual review, each aspect is meticulously examined to ensure that the benchmark samples meet high standards of quality and robustness.

C Benchmark Metric

To evaluate the performance of Code LLMs on the CoCo-Bench, we incorporate a weighting scheme

based on the difficulty of each sample, ensuring a more accurate assessment of a model's capabilities.

We first apply TrueSkill2 (Minka et al., 2018) approach to assess the difficulty coefficient of each task, marked as μ_i. We then calculate w_i using inverse normalization:

$$w_i = \frac{1/\mu_i}{\sum_{j=1}^{5} 1/\mu_j}$$

ensuring $\sum_{i=1}^{5} w_i = 1$. Let w_i represent the weight assigned to the i-th sample, reflecting the contribution of each task to the overall score. Higher weights correspond to more complex samples. For all tasks, we define the difficulty-aware pass rate (DAPR) as follows:

$$\text{DAPR} = \frac{\sum_{i=1}^{n}(\text{Pass Rate}_i \times w_i)}{\sum_{i=1}^{n} w_i}$$

where Pass Rate$_i$ represents the pass rate for the i-th sample—indicating how often the model successfully passes this specific test case—w_i is the difficulty weight of the i-th sample, and n is the total number of samples in the task.

To compute the overall CoCo-Score, we combine DAPR of each task:

$$\text{CoCo-Score} = \sum_{j=1}^{4} \left(\frac{n_j}{N} \times \text{DAPR}_j \right)$$

where n_j is the number of samples in the j-th task, N is the total number of samples across all tasks, DAPR$_j$ is the difficulty-aware pass rate for each respective task.

Our approach ensures that the CoCo-Score not only measures the average performance of the models but also emphasizes their ability to handle more complex coding challenges by using difficulty-aware scores, providing a comprehensive assessment of a model's practical effectiveness and robustness across various coding tasks.

D Prompts for Inference

In specific tasks within CoCo-Bench, the input typically consists of a prefix prompt and a suffix prompt. The prefix prompt includes several examples to assist the large model in understanding the specific task requirements and the expected output format. These examples effectively guide the model, ensuring it performs the task correctly. The suffix prompt appears at the end of the input and serves to reinforce the required output structure, ensuring that the model's output is consistent with the examples provided, which facilitates subsequent automated processing.

D.1 Prompts for CU Inference

Prefix prompt for CU_F :

Please deduce the output of the following code based on the code snippet and the input.

Infix prompt for CU_F :

The code snippet is as follows:

```
import numpy as np
def power_sum(arr1,arr2):
    powered_arr = np.power(arr1,
        arr2)
    result_sum = np.sum(powered_arr
        )
    return result_sum
```

Infix prompt for CU_F :

The input is as follows:

```
[[2,3,4], [1,2,3]]
```

Suffix prompt for CU_F :

Give only the deduced output of the code snippet. Do not output any additional information.

Prefix prompt for CU_R :

Please deduce the input of the following code based on the code snippet and the output.

Infix prompt for CU_R :

The code snippet is as follows:

```
import numpy as np
def power_sum(arr1,arr2):
    powered_arr = np.power(arr1,
```

```
        arr2)
    result_sum = np.sum(powered_arr
        )
    return result_sum
```

Infix prompt for CU_R :

The output is as follows:

```
102
```

Suffix prompt for CU_R :

Give only the deduced input of the code snippet. Do not output any additional information.

D.2 Prompts for CG Inference

Prefix prompt for CG :

Please fill in the following incomplete code according to the description. The description is as follows:

You are given an array of positive integers nums. Alice and Bob are playing a game. In the game, Alice can choose either all single-digit numbers or all double-digit numbers from nums, and the rest of the numbers are given to Bob. Alice wins if the sum of her numbers is strictly greater than the sum of Bob's numbers. Return true if Alice can win this game, otherwise, return false.

Infix prompt for CG :

The incomplete code is as follows:

```
def canAliceWin(self, nums: List[
    int]) -> bool:
    single=0
    double=0
    for it in nums:
        if it>=10:
            double=____
        else:
            single=____
    return single !=double
```

Suffix prompt for CG :

Give only the completed code. Do not output any additional information.

D.3 Prompts for CM Inference

Preffix prompt for CM:

Please correct the following code according to the description. The description is as follows:

You are given a 0-indexed string s typed by a user. Changing a key is defined as using a key different from the last used key. For example, s = "ab" has a change of a key while s = "bBBb" does not have any. Return the number of times the user had to change the key. Note: Modifiers like shift or caps lock won't be counted in changing the key that is if a user typed the letter 'a' and then the letter 'A' then it will not be considered as a changing of key.

Infix prompt for CM :

The code to be corrected is as follows:

```
def countKeyChanges(self, s: str)
    -> int:
    if len(s) == 1:
        return 0
    s = s.upper()
    count = 0
    for i in range(len(s)-1):
        if s[i] == s[i + 1]:
            count += 1
    return count
```

Suffix prompt for CM:

Give only the corrected code. Do not output any additional information.

D.4 Prompts for CR Inference

Prefix prompt for CR:

Please find errors in the following code according to the description. The description is as follows:

Function uses the 'eval' function to execute dynamic expressions from user inputs, posing serious security risks.

Infix prompt for CR :

The code with errors is as follows:

```
def execute_expression(user_input):
    result = eval(user_input) #
        Dangerous use of eval
    return result
```

Suffix prompt for CR:

There are four types of errors: performance_issues, security_issues, syntax_errors and logical_errors. Please give accurate error types and correct the code, in the form of

```
{
    "performance_issues":
        "data = request.get(user_url)",
    "security_issues":
        "password = getpass.getpass()",
    "syntax_errors":
        "print(a + b)",
    "logical_errors":
        "continue if a > b else break"
}
```

E More Hyper Parameter Analysis

An interesting phenomenon can be observed from these Figure **??** and Figure **??**: models with lower performance seem to be more sensitive to decoding temperature. Specifically, CodeLlama-7b-hf achieves its highest score at a lower temperature, while DeepSeek-Coder-6.7b-Instruct shows greater score fluctuations at higher temperatures.

This can be explained by the relationship between a model's generation capability and the decoding temperature. Higher-performing models typically have stronger generation capabilities, allowing them to maintain more stable performance across a wider range of temperatures. In contrast, lower-performing models may rely on specific decoding temperatures to enhance the quality of their outputs when tackling complex tasks. For instance, CodeLlama-7b-hf might produce more deterministic outputs at lower temperatures, avoiding the randomness introduced at higher temperatures, which leads to better scores in certain tasks.

On the other hand, lower-performing models are more sensitive to temperature changes, possibly because they struggle to maintain coherence and quality in their outputs at higher temperatures. As the temperature increases, the generated text may become more random, leading to a decline in task performance. This also explains why CodeLlama-7b-hf achieves its highest score at a lower temperature: at lower temperatures, the model produces more deterministic and consistent content, avoiding the noise introduced by unnecessary randomness.