Poster: Benchmarking of Code Generative LLMs

Mirza Masfiqur Rahman
Cisco Research
California, USA
mirrahma@cisco.com

Ashish Kundu
Cisco Research
California, USA
ashkundu@cisco.com

Elisa Bertino
Purdue University
Indiana, USA
bertino@purdue.edu

***Abstract*—Generative LLMs have proven to be valuable code generators, thus enabling code copilots and meeting several requirements in software engineering. However, several questions arise: How good an LLM is as a software engineer? How secure is the code generated or fixed by an LLM? These are complex questions to address; however, addressing them is critical for enabling trustworthy software development ecosystems. Addressing those questions requires a designing rigorous benchmark to evaluate: (i) the code that is generated/completed, and (ii) the generative LLMs themselves. In this paper, we propose an automated benchmarking system covering the different aspects of the generated code and the LLMs that generate the code. We also propose the concept of a benchmark dependency graph, coupled with an automated benchmark scoring process that provides a vector of scores on how "good" or how "bad" an LLM is, or the code generated/modified by it is, additionally the artifacts generated or modified around the code.**

***Index Terms*—Language models, LLM, Code generation, Benchmark, Automation, NLP**

I. Introduction

Large Language Models (LLMs) have demonstrated significant potential in code generation and subsequently been powering code engines such as Copilot. Currently, there are numerous LLMs–some specialize in code generation such as Codex, Code Llama, StarCoder, CodeGen, etc., while others such as GPT-3.5, GPT 4.0, Llama-2/3, Mistral, etc. can generate codes alongside solving general tasks.

Unlike other tasks, code generation is more challenging for autoregressive LLMs. Past research has designed benchmarks for LLMs on their code-generating capabilities where manual references are mandated. Some works have either created benchmark datasets by hand to evaluate one LLM or benchmarked based on diversity and concept difficulty in programming problems in a single language [1], [2]. However, because of the dynamically improving ecosystem of open-sourced LLMs and their increasing use in complex software projects, these static (and often manual) benchmarks are insufficient and often outdated. Moreover, there is no structured evaluation to understand the capability of these models in working as a complete programmer or in a collaborative manner.

We address such a gap by proposing a framework for automatic benchmarking, addressing the following questions - how good is an LLM as a software engineer, and how good is a set of LLMs as a software engineering team? Automated evaluation of such LLMs and code is essential.

II. Benchmarking Framework

Our framework automatically evaluates LLMs for their capability as complete code generators and code copilots. It consists of the three components: (1) benchmark workflow graph, (2) benchmark and evaluation bipartite graph, and (3) benchmark evaluation model.

(1) ***Benchmark workflow graph***: It comprises of vertices, each representing an entity being evaluated by one or more benchmarks, and edges representing the flow of the evaluation process. For each vertex, a set of evaluation metrics is also selected (a subset of such evaluation metrics falling under each benchmark, described using a bipartite relationship, see Fig. 2). Thus, to compute an arbitrary set of evaluation metrics or a set of evaluation metrics that characterize the vertex, one may need to select a set of benchmarks. In our framework, we consider the following entities:

A. Single code segment C_i: How good a piece of code generated by an LLM is, based on a comprehensive set of properties?
B. Set of Code segments $S(C_i)$: Collectively how good a set of code segments is, with respect to a task mentioned in a prompt?
C. LLM as a Co-pilot CP_i: How well does an LLM act as a co-pilot and augment human-written code with its own generated code, such as for fixing bugs?
D. LLM ($\mathcal{L}_i$): As a code generator, how good an LLM is, based on the various complete codes it can generate?
E. LLMs as a team (τ_i): How good a team of LLMs is, when working together in generating and updating code?

Fig. 1 presents the workflow graph, including the input, such as a predefined code repository, a human, or an autonomous agent, and the dependency among the entities. The input from code repositories is associated with a pre-defined text prompt to ensure conversational style.

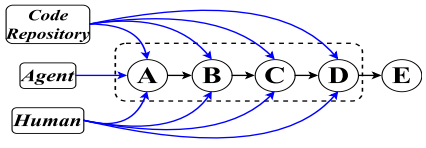

Fig. 1. Benchmark workflow graph. Blue edges denote prompts to evaluate an entity.

(2) ***Benchmark and Evaluation Bipartite Graph***: It is a graph representing dependencies between a set of benchmarks

and evaluation metrics. Briefly, a benchmark is expressed as a set of metrics, whereas one might want to calculate another arbitrary set of metrics that falls under more than one benchmark. Figure 2 presents such a scenario.

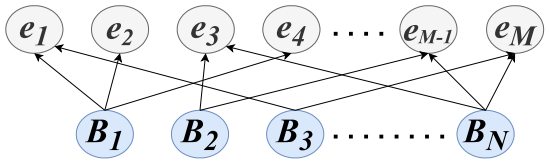

Fig. 2. Benchmark and evaluation bipartite graph. A set of metrics can fall under one benchmark and an arbitrary set of metrics calculation requires multiple benchmark to be run.

Each software engineering requirement r_i is associated with one evaluation metric e_i. A set of evaluation metrics are computed from running one or more benchmarks.

(3) ***Benchmark evaluation model***: Let $T = \{t_1, t_2, \dots t_n\}$ denote the set of code generation tasks. For each task, t_i let I_{t_i} represent the set of input specifications. Thus, for each task t_i and its corresponding input specification I_{t_i}, the LLM $\mathcal{L}$ generates a code snippet C_{t_i}. Now assume $E = \{e_1, e_2, \dots, e_M\}$ be the set of evaluation metrics used to assess the quality of the generated code snippets. Thus, we need to define an evaluation function $f : T \times E \to R$ that maps each task t_i and evaluation metric e_j to a real-valued score representing the performance of $\mathcal{L}$ on task t_i with respect to the metric e_j. Let w_{ij} denote the weight assigned to evaluation metric e_j for task t_i, where $w_{ij} \geq 0$ and $\sum_{j=1}^{m} w_{ij} = 1$ for each task t_i. The weights w_{ij} reflect the relative importance of each evaluation metric for each task. To define the benchmark score $B_{\mathcal{L}} : T \to R$ for $\mathcal{L}$ as a function of the performance scores obtained across all tasks and evaluation metrics:

$$B_{\mathcal{L}}(t_i) = \sum_{j=1}^{m} w_{ij} \hat{f}(t_i, e_j), \quad B_{\mathcal{L}} = \frac{1}{n} \sum_{i=1}^{n} B_{\mathcal{L}}(t_i)$$

$B_{\mathcal{L}}(t_i)$ represents the overall performance of $\mathcal{L}$ on task t_i considering all evaluation metrics, weighted according to w_{ij} For the aggregated benchmark score $B_{\mathcal{L}}$ for $\mathcal{L}$ is calculated by aggregating the individual scores across all tasks. The aggregation can be naively weighted sum or can be moderated to characterize more complex relationship.

III. Automated Benchmarking Process

A comprehensive set of properties can be evaluated based on two primary software engineering requirements: functional and non-functional. Functional requirements mandate semantic correctness and completeness of the code [3]. It may also include advanced capabilities, such as symbolic execution or fuzzing [4] while debuggability, reliability, security, privacy, compliance, optimality, code maturity, etc., that fall under non-functional requirements. It may also include aspects related to cost and maintainability. Algorithm 1 presents our approach for the automated benchmark and for brevity, we include the benchmarking process based on entities A and D.

Algorithm 1 Holistic Benchmarking Algorithm

Input: Properties $\Phi : \{\phi_1, \phi_2, \dots \phi_d\}$, Generated codes $\zeta : \{C_1, C_2, \dots C_n\}$, LLMs $\Lambda : \{\mathcal{L}_1, \mathcal{L}_2, \dots \mathcal{L}_p\}$
Require: Score Aggregators $\Rightarrow$ Code: $\mathcal{A}^C(\cdot,\cdot)$, LLM: $\mathcal{A}^{\mathcal{L}}(\cdot,\cdot)$; Property verifier $\mathcal{F}(\cdot,\cdot)$
Output: Code score set $[e]_n$, LLM score set $[\beta]_p$
  for $a \leftarrow 1$ to p **do**
    $\beta^{\mathcal{L}_a} \leftarrow 0$ ▷ Initially all LLM scores are 0
  end for
  for $i \leftarrow 1$ to n **do**
    $e^i \leftarrow 0$ ▷ Initially i-th code score is 0
    for $j \leftarrow 1$ to d **do**
      $S^i_j \leftarrow \mathcal{F}(C^{\mathcal{L}_a}_i, \phi_j)$, $e^i \leftarrow \mathcal{A}^C(e^i, e^i_j)$
      $\beta^{\mathcal{L}_a} \leftarrow \mathcal{A}^{\mathcal{L}}(\beta^{\mathcal{L}_a}, e^i)$
    end for
  end for

A. Benchmark dependency graph.

Benchmarks may have a dependency relationship with each other. This can be expressed as a directed acyclic graph. As we mentioned, to calculate a set of metrics, one might need to run multiple benchmarks. The relative ordering of benchmarks that cover the metrics can be delineated through the dependency and these benchmarks can be run using a depth-first approach.

IV. Conclusion

In this paper, we propose a benchmarking framework for LLMs based on not only their arbitrary code generation capabilities but also their ability as a comprehensive development tool–the ability to generate property-comprehensive code(s), to work as a co-pilot, to work in a team, and so on. Taking into account a wide array of aspects, we show that the varied benchmarking scheme may have significant potential over traditional code quality scores, and their interconnectivity can bring them under a common standard.

Acknowledgment

This work has been partially supported by NSF Grant No. 2112471.

References

[1] A. Yadav and M. Singh, "Pythonsaga: Redefining the benchmark to evaluate code generating llm," 2024.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[3] J. Liu, C. S. Xia, Y. Wang, and L. ZHANG, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *Advances in Neural Information Processing Systems* (A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, eds.), vol. 36, pp. 21558–21572, Curran Associates, Inc., 2023.

[4] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, (New York, NY, USA), p. 423–435, Association for Computing Machinery, 2023.