Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review

Debalina Ghosh Paul, Hong Zhu and Ian Bayley
School of Engineering, Computing and Mathematics, Oxford Brookes University
Oxford OX33 1HX, UK. Email: hzhu@brookes.ac.uk

***Abstract*—With the rapid development of Large Language Models (LLMs), a large number of machine learning models have been developed to assist programming tasks including the generation of program code from natural language input. However, how to evaluate such LLMs for this task is still an open problem despite of the great amount of research efforts that have been made and reported to evaluate and compare them. This paper provides a critical review of the existing work on the testing and evaluation of these tools with a focus on two key aspects: the benchmarks and the metrics used in the evaluations. Based on the review, further research directions are discussed.**

***Index Terms*—Machine learning, Large language models, Code generation, Performance evaluation, Benchmarks, Metrics.**

I. Introduction

The recent years have seen a rapid growth in machine learning (ML) technology for natural language processing. Deep neural networks (DNN) in the transformer architecture [1] with billions of parameters have been developed. They have an impressive capability in conducting natural language processing (NLP) related tasks. Among the most valuable capabilities of these is code generation from natural language input. It is thought that this may fundamentally change the way software is developed; see e.g., [2]. A great amount of research efforts have been reported in the literature on uses of such general ML models and developing special purpose ones for solving programming problems; see, e.g. [3]–[5] for recent surveys of such ML models.

However, it is still an open question how well they actually perform even after many reports have been published on this issue. A large number of different benchmarks and quality metrics have been proposed and employed in the evaluations. However, the conclusions of these evaluations and comparisons often conflict with each other, and the results hardly reflect the real experiences of the users. Therefore, it is highly desirable to understand the current state of the art in the evaluation of ML models as code generation tools. This paper provides such a review and analyses the directions for future research.

The remainder of the paper is organised as follows. Section II discusses various types of coding tasks that LLMs have been applied to solve, and summarises the large language models that are used or designed for solving coding problems. Section III reviews the benchmarks used in the evaluations. Section IV focuses on the quality attributes and their metrics of code generation. Section IV-D is devoted to the performance metrics. Section V analyses the problems in the current approach and discusses the directions for further research.

II. Overview of LLMs for Coding Tasks

The research on ML for generating program code can be backdated to 1980s; see, for example, [6], [7]. However, the emergence of LLMs has been a recent breakthrough. In this context, programming tasks are regarded roughly as translations between natural language descriptions of programming problems and codes in programming languages, or vice versa. Therefore, such programming tasks can be classified into three categories: *Description to Code* (D2C), *Code to Description* (C2D) and *Code to Code* (C2C) according to the source and target languages [8].

The D2C type of programming tasks take input in natural language that specifies the coding requirements; typically this is a functional specification. It is often called *code generation* (CG) in the literature. A typical example of the ML models for code generation is Codex[1] which underlies ChatGPT. A special form of code generation is *pseudo-code implementation*, which translates pseudo-code written in natural language text into program code [9]. It is worth noting that the input may also contain code fragments to define the context of the code to be generated.

The C2D type of programming tasks take program code as the input and produce natural language text as the output. The typical examples of such tasks include *document generation*, *code summarisation* [10], and *comment generation* [11], etc.

The C2C type of programming tasks take a piece of code as the input and produce another piece of code as the output. Typical examples of this type include *code completion* (such as GitHub's copilot[2]), *code infilling* (like StarCoder[3] [12]), *code translation*, *code refactoring* [13], *automatic debugging* (which is also called *program repair* or *code repair* by many authors) [14], and *test generation* [15], [16], etc.

In this paper, our focus is on the D2C type of programming tasks, but those issues that overlap with the C2C type will also be included; the C2D type will be omitted, however.

There are two approaches to developing ML models as tools for programming tasks. The first is to employ general purpose LLMs such as ChatGPT and Gemini [17]. Although their primary purpose lies in natural language processing, they also

[1] https://openai.com/index/openai-codex/
[2] https://github.com/features/copilot
[3] https://huggingface.co/bigcode/starcoder

possess significant capability for various programming tasks since they have been trained on datasets that contain program code.

The second approach is to develop special purpose LLMs for programming tasks either through fine tuning of pre-trained LLMs or training the model from scratch.

Table I summarise the key features of the most well known LLMs for programming tasks, including their sizes, release years, the benchmarks used to evaluate their performance, and their performance as measured by $pass@100$ except for StarCoder, which was measured by $pass@1$.

TABLE I
PERFORMANCE COMPARISON OF LANGUAGE MODELS

Model	Base Model	Size	Year	Benchmark	Score
GPT-NEO [18]	GPT-2	125M 1.3B 2.7B	2021	HumanEval	02.97 16.30 21.37
GPT-J [19]	GPT-2	6B	2021	HumanEval	27.74
Codex [20]	GPT-3	300M 679M 2.5B 12B	2021	HumanEval	36.27 40.95 59.50 72.31
TabNine	?	?	2021	HumanEval	7.59
ChatGPT	GPT-3.5	?	2022	HumanEval	94.00
Gemini-Ultra [17]	Transformer	?	2023	HumanEval Natural2Code	74.40 74.90
Gemini-Pro [17]	Transformer	?	2023	HumanEval Natural2Code	67.70 69.60
CodeGen [21]	Auto-regressive Transformer	350M 2.7B 6.1B 16.1B	2023	HumanEval	35.19 57.01 65.82 75.00
SantaCoder [22]	Decoder	1.1B	2023	MultiPL-E	45.90
InCoder [23]	Transformer	1.1B 6.7B	2023	HumanEval	25.20 45.00
StarCoder [12]	StarCoder-Base	15.5B	2023	HumanEval MBPP	33.60 52.70

While there is much research effort and literature on the evaluation of LLM performance, many research questions remain. For example, are the evaluations and comparisons fair and are the differences significant? Do the results of performance evaluation truly reflect the usability of LLMs as practical programming tools? etc. In order to answer these questions, we will examine, in the subsequent sections, how the benchmarks were constructed, how the performances are assessed and the metrics are defined.

III. BENCHMARKS

Benchmarks play a crucial role in the evaluation of ML models and a large number of them have been proposed specifically for the evaluation of LLMs. Table II summarises the most well known such benchmarks. We will discuss first how these benchmarks are constructed and then their main characteristics.

To construct a benchmark, data must be procured, extracted and processed. We will consider each of these jobs in turn.

A. Procurement

We identify the following potential sources of data for programming tasks. Table II also shows the sources of the benchmarks that we have reviewed.

- *Code repositories*, such as GitHub.
- *Online forums* for discussing programming problems and solutions, such as Stack Overflow.
- *Coding challenge sites*, such as Codewars, AtCoder, Kattis, and Codeforces.
- *Freelancer sites*, where software development tasks were outsourced, such as Upwork.
- *Pre-existing datasets*, which can be included completely or partly in another benchmark.
- *Textbooks* on programming.
- *Online Tutorial* websites, such as W3C resources.
- *Domain experts*, who custom-write tasks and provide solutions.
- *Crowdsourcing sites*, which gather data from the crowd.

TABLE II
SOURCES OF EXISTING BENCHMARKS

Benchmark	Source
APPS [24]	Coding challenge
HumanEval [20]	Domain Experts
MBPP [25]	Crowd-sourcing
MathQA-Python [25]	Dataset: MathQA
ClassEval [26]	Repository, Datasets: HumanEval, MBPP
CoderEval [27]	Repository: Github
Multipl-E [28]	Datasets: HumanEval, MBPP
DS-1000 [29]	Forum: Stack Overflow
HumanEval+ [30]	Dataset: HumanEval
CONCODE [31]	Repository: Github
R-benchmark [32]	Text Books
JuICe [33]	Repository: Github (Jupyter, nbgrader)
Exec-CSN [34]	Repository: Github, Dataset: CodeSearchNet
EvoCodeBench [35]	Repositories: GitHub

B. Data Extraction and Processing

Manual extraction from sources, as done for datasets CoderEval and DS-1000, is labour intensive. Automated extraction is less so and possible for online platforms like code repository and Q&A forum, but work must still be done to write scripts or code for each source and to clean the data afterwards. Note that there is also the possibility of missing data. When existing benchmarks are being reused, as with MathQA-Python, MultiPL-E and HumanEval+, extraction is easier to do.

Data extracted from a source often require processing before including in the benchmark. According to the purposes, data processing tasks can be classified into the following three types.

- *Clarification*: clarifying the task description to reduce the ambiguity and incompleteness in the natural language specification of the task. Except for MultiPL-E, all benchmarks we reviewed have been manually edited after the data were extracted. For example, CoderEval employed

13 people for the task. When many people are involved, there is a risk of inconsistency between editors.

- *Deduplication*: removing duplicated data from the dataset. It is done manually for DS-1000 and with automation for APPS, which employed tf-idf features coupled with SVD dimensionality reduction and cosine similarity. However, it is not clear whether and how this was done for the other benchmarks.
- *Decontamination*: removing data used in the fine-tuning or training the LLM in the case that the benchmarks have been leaked to the LLM. This is done for APPS [24].

C. Functionality and Structure

One major difference between the benchmarks is the level of code generation: whether the task is to generate a statement, a function, a class or a whole program. This level of functionality is documented in Table III below, along with the number of tasks in the benchmark and the programming language.

TABLE III
FUNCTIONALITY OF EACH BENCHMARK

Benchmark	Level	#Tasks	Language
APPS	Program	10,000	Python
R-benchmark	Program	351	R
ClassEval	Class	100	Python
HumanEval	Function	164	Python
MBPP	Function	974	Python
MathQA-Python	Function	23,914	Python
CoderEval	Function Method	230 230	Python Java
Multipl-E	Function	1138	Various
HumanEval+	Function	164	Python
CONCODE	Method	2000	Java
DS-1000	Statement	1000	Python
JuICe	Program	3700	Python
Exec-CSN	Function	1931	Python
EvoCodeBench	Function	275	Python

Another major difference between benchmarks is in the structure of the elements in the dataset. There are four types of components that have been included in existing benchmarks. Natural language descriptions are usually given as text. In addition, context code (such as function signatures) and unit test cases may be provided, both of which can be used for test automation to check the correctness of the generated solutions. In some cases, there may also be reference solutions. Table IV gives the components provided for each benchmark, where the column #Test Cases gives the average number of test cases per task if test cases are provided in the benchmark.

D. Task Classification and Metadata

In some cases, programming tasks in the dataset are classified into subsets of different difficulty levels. For example, Hendrycks et al. [24] distinguish three levels (Introductory, Interview, Competition) in APPS. Austin et al. [25] split tasks into two subsets MBPP and MathQA-Python as two levels of difficulty. Similarly, Yu et al. [27] used six levels according to the function's contextual dependency in CoderEval.

TABLE IV
STRUCTURE OF DATA FOR EACH BENCHMARK

Benchmark	Context Code	#Test Cases	Solution
APPS	-	+	+
HumanEval	function signature	7.7	-
MBPP	-	3	-
MathQA-Python	-	3	-
ClassEval	class skeleton	33.1	+
CoderEval	function signature	+	-
Multipl-E	function signature	3 to 7	-
DS-1000	signature	1.6	+
HumanEval+	function signature	774.8	-
CONCODE	function signature	-	-
R-benchmark	-	-	+
JuICe	-	-	+
Exec-CSN	function signature	+	+
EvoCodeBench	function signature	+	+

Miah and Zhu [32] also distinguish five difficulty levels but they indicate the difficulty level as a part of the metadata associated to each task together with the task source and task types, making it possible to evaluate the effect of changing both of these. This can provide strong support to scenario-based testing and evaluation as shown in their case study.

IV. QUALITY ATTRIBUTES AND METRICS

We now review the quality attributes that LLMs are assessed against and the metrics used to measure LLMs.

A. Functional Correctness

Correctness of generated code is the main quality attribute for assessing the response of an LLM to a programming task, and according to what is provided by the benchmark, it is measured using reference solutions (ConCode), test cases (HumanEval, HumanEval+, MBPP, MathQA-Python and MultiPL-E) or a combination of both (APPS, ClassEval, CoderEval, and DS-1000). Where reference solutions are provided, correctness can either be functional correctness, as measured by passing tests, or syntactic closeness, for example, measured with the BLEU metric.

Given a set $T_p = \{t_1, \cdots, t_n\}$ of test cases for a programming task p, a program code P generated by a ML model M is regarded as correct with respect to T_p, if P is correct on all test cases t_i in T_p. Let B be a benchmark dataset for evaluating a ML model M. A basic performance metric based on pass-all-tests is the percentage of tasks in B that the generated code successfully passes all tests. Note that there is randomness that an LLM generates program codes. This issue is addressed in the $pass@k$ metrics discussed in Section IV-D.

Another metric commonly used is the percentage of test cases in T_p on which the program P is correct. This is denoted by $TPR_{T_p}(P)$, where TPR stands for *test pass rate*. The corresponding overall performance of the model M on benchmark B is the average test pass rate. Formally,

$$AvgTPR_B(M) = \frac{\sum_{p \in B} TPR_{T_p}(M(p))}{\|B\|}$$

B. Syntactic Closeness

Similarity metrics have been used since the early evaluation of code generation. Some, like BLEU and ROGUE, were inherited from natural language, while others have been proposed specifically for code, such as Ruby and CodeBLEU.

1) BLEU: In 2002, Papineni et al. [36] proposed BLEU (Bilingual Evaluation Understudy) to evaluate the machine translations from one language to another. BLEU compares n-grams (n number of contiguous words) between the generated translations by the language models with the reference translations. It calculates a precision score based on the number of n-grams of generated text G that match with the reference text R, out of all generated n-grams. Then, this precision score is adjusted by a brevity penalty to account for translation length, penalising systems that generate excessively short translations.

Formally, let T be any given text that consists of a sequence of words $(\tau_1, \cdots, \tau_k)$, where $k \geq 0$. The n-gram of T can be defined as the set $Gram_n(T) = \{(\tau_i, \cdots, \tau_{i+n}) \,|\, i = 1, \cdots, k-n\}$. The n-gram precision of text G with respect to R is defined as follows.

$$p_n(T, R) = \frac{\|Gram_n(T) \cap Gram_n(R)\|}{\|Gram_n(T)\|}$$

The BLEU score is computed using the following formula.

$$BLEU(G, R) = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n(G, R)\right)$$

Here, w_n is the weight for n-gram precision, and BP is the brevity penalty, which is computed using the following formula.

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-\frac{r}{c})} & \text{if } c \leq r \end{cases}$$

where c is the length of the candidate translation and r is the length of the reference translation.

The BLEU score is a number from 0 to 1 where a higher score indicates better alignment between the generated and reference translations.

2) ROUGE: In 2004, Lin et al. [37] proposed ROUGE (Recall-Oriented Understudy for Gisting Evaluation) which is a family of metrics for comparing generated text with a set of reference texts.

ROUGE-N measures the overlap between the n-grams of a generated text G and the n-grams of a set of reference texts R, and is calculated using the following formula.

$$ROUGE_N(G, R) = \frac{\sum_{S \in R} \|Gram_n(S) \cap Gram_n(G)\|}{\sum_{S \in R} \|Gram_n(S)\|}$$

ROUGE-L measures the longest common subsequence (LCS) of words between the system generated and the reference texts. The calculation incorporates the precision P_{lcs} and recall R_{lcs} between the generated text G and reference text R. Let $len(T)$ be the length of text T, i.e. the number of words in T. Formally,

$$P_{\text{lcs}}(G, R) = \frac{LCS(G, R)}{len(G)}, \quad R_{\text{lcs}}(G, R) = \frac{LCS(G, R)}{len(R)}$$

$$ROUGE_L(G, R) = \frac{(1 + \beta^2) \cdot P_{\text{lcs}}(G, R) \cdot R_{\text{lcs}}(G, R)}{R_{\text{lcs}}(G, R) + \beta^2 \cdot P_{\text{lcs}}(G, R)}$$

where $LCS(G, R)$ is the length of the longest common subsequence between G and R, and β is a parameter that balances precision and recall. Typically, $\beta = 1$ for equal weighting.

There are three more metrics in the ROUGE family. ROUGE-W is a weighted variant of ROUGE-L, which assigns different weights to different LCS matches. ROUGE-S measures the overlap of skip-bigrams, which are any pair of words that occur in their sentence order, with allowance for arbitrary gaps. Skip-bigram co-occurrence statistics quantify the similarity in skip-bigrams between a generated and a set of reference texts. ROUGE-SU is a variant of ROUGE-S that also includes unigrams (single words). These metrics have not been used in the evaluation of code generation, so their definition is omitted. Readers are referred to [37] for details.

3) METEOR: In 2005, Banerjee et al. [38] proposed the METEOR (Metric for Evaluation of Translation with Explicit ORdering) metric, later extended by Denkowski et al. [39] to arbitrary target languages. The generated and reference text are compared based on several components, including exact word matches, stemmed matches, semantic similarity using WordNet and phrase matches. Exact match counts the number of exact word matches between the two texts. Stemmed match captures variations of words that have the same root. Synonymy uses WordNet to find synonyms, and phrase match utilises a paraphrase table to find paraphrases. The METEOR score is computed using the harmonic mean of precision and recall (F-mean) and adjusted by a penalty for word order errors.

$$METEOR = F_{\text{mean}} \cdot (1 - Penalty)$$

The precision of a generated text G with respect to a reference text R is defined as the ratio of the number of words in G that matches words in the reference text R (including exact, stemmed, and synonym matches) to the total number of words in G. Formally, let $MatchWords(G, R)$ denote the set of words in G that matches words in R. and $\|T\|$ be the number of words in a text T. Formally,

$$Prec(G, R) = \frac{\|MatchWords(G, R)\|}{\|G\|}$$

The recall is defined as the ratio of the number of words in G that matches words in R to the total number of words in the reference text R.

$$Rec(G, R) = \frac{\|MatchWords(G, R)\|}{\|R\|}$$

Penalty is applied for alignment fragmentation, which considers gaps and shifts in word order between the generated and reference texts. The penalty reduces the score for disorganized alignments.

4) ChrF: In 2015, Popovic et al. [40] proposed the ChrF (Character n-gram F-score) metric which measures the similarity between a generated text and a reference text by comparing character n-grams (contiguous n character) instead of word-level tokens. The ChrF score is also calculated using the harmonic mean of precision and recall.

$$ChrF = \left(1 + \beta^2\right) \cdot \frac{Prec \cdot Rec}{\beta^2 \cdot Prec + Rec}$$

where:

- $Prec$ denotes precision, which is the ratio of the number of matched character n-grams to the total number of character n-grams in the generated text.
- Rec denotes recall, which is the ratio of the number of matched character n-grams to the total number of character n-grams in the reference text.
- β is a parameter that determines the relative importance of recall over precision (commonly set to 1 for balanced importance).

Ruby and CodeBLUE are custom metrics designed to address the characteristic features of programming languages.

5) Ruby: In 2019, Tran et al. [41] proposed Ruby, a similarity metric that compares the Program Dependency Graphs (PDGs) of the generated and reference codes. If a PDG cannot be constructed, then Abstract Syntax Trees (ASTs) are compared instead. If an AST cannot be constructed, the metric uses weighted string edit distance between the tokenized reference R and generated G codes.

$$RUBY(G, R) = \begin{cases} GRS(G, R) & \text{if PDGs are applicable,} \\ TRS(G, R) & \text{if ASTs are applicable,} \\ STS(G, R) & \text{otherwise.} \end{cases}$$

where:

- $GRS(G, R)$ measures the similarity between two program dependency graphs for R and G.
- $TRS(R, C)$ measures the similarity between the Abstract Syntax Trees (ASTs) for R and G.
- $STS(R, C)$ measures the string edit distance between R and G.

6) CodeBLEU: In 2020, Ren et al. [42] proposed CodeBLEU, a metric that extends the traditional BLEU metric by including program-specific features. CodeBLEU evaluates the similarity between the generated and reference codes by considering four different sub metrics: n-gram match (BLEU), weighted n-gram match, AST (Abstract Syntax Tree) match and data flow match. Weighted n-gram extends the traditional n-gram matching by assigning different weights to different types of tokens (e.g., keywords, identifiers, operators). AST match compares the Abstract Syntax Trees (ASTs) of the generated and reference codes. Data flow match evaluates the similarity in data flow graphs between the generated and reference code.

$$\begin{aligned} CodeBLEU = \alpha \cdot BLEU + \beta \cdot \text{Weighted-N-gram} \\ + \gamma \cdot \text{AST-Match} + \delta \cdot \text{DataFlowMatch} \end{aligned}$$

where α, β, γ and δ are weights that add up to 1.

In addition to the above metrics on the syntax closeness, Lai et al. used a much-relaxed form of similarity metric called surface-form constraints, which requires the presence or absence of certain specific APIs and/or the keywords in the solution code [29].

7) Validity of the Metrics: Although the use of similarity metrics in the performance evaluation of NLP models has been the standard approach in ML research, several authors have questioned if it is valid for code generation. In 2019, Kulal et al. [9] found that BLEU fails to assess functional correctness. In 2021, Hendrycks et al. [24] further showed it is inversely correlated with functional correctness.

Evtikhiev et al. [45] is perhaps the first systematic study of the applicability of six similarity metrics, BLEU, ROUGE-L, METEOR, ChrF, CodeBLEU, and RUBY, to code generation. They investigated whether evaluations that employ these metrics yield statistically significant results and whether they correlate well with human judgement. Their conclusion was that an improvement of a corpus-level metric score by less than 2 points might not be enough to warrant a statistically significant improvement in quality without additional statistical tests. Even an improvement in score by less than 5 points may not correspond to a statistically significant improvement. They found that ChrF is the closest match to human assessment but it cannot be considered the "perfect" metric for code generation and such a metric is yet to be found. Interestingly, RUBY and CodeBLEU metrics, both developed for the specific purpose of assessing code, performs no better than more generic metrics from the domain of machine translation.

Kulal's solution to the problems observed with BLEU was to judge the correctness of an LLM generated solution by whether it passes all test cases. Hendrycks et al. [24] also did this and additionally, used the test pass rate as a metric of correctness. Since then, most evaluations of code generation performances have employed test correctness.

However, more recently, in their study of GitHub Copilot, Ziegler et al. found that "while suggestion correctness is important, the driving factor for these improvements [in productivity] appears to be not correctness as such, but whether the suggestions are useful as a starting point for further development" [43]. This supports the assumption made in Miah and Zhu's study of ChatGPT's usability [32] that users may accept a generated solution if it is good enough to use even if it is not correct.

C. Usability and Productivity

There has been very little research on other quality attributes for evaluating LLMs as code generation tools. However, we are aware of the work of Miah and Zhu [32] on the usability of ChatGPT and that of Ziegler et al [43] and Xu et al [44] on productivity.

Usability is about how easy it is to use the tool to achieve the user's goal. In [32], it is assumed that LLM responses are checked by the human user to see if the solution can be used for his/her programming task. It need not be correct but it must

be good enough to use. This means a generated code to be checkable by the user. Thus, it should be understandable. This entails that the code should be *well structured* and *logically clear*. The generated code should also be easy to revise and adapt. This entails that it should be *concise*, *complete*, and *accurate* in term of close to correct, etc. Moreover, the text explanations generated by the ML model in company with the code should provide *sufficient explanation* and *readable*, etc. These were the quality attributes used by Miah and Zhu in evaluation of usability [32]. These quality attributes were manually assessed on a Likert scale of 1 to 5, with 1 the poorest and 5 the best quality, by comparing with a reference solution. Accuracy, however, may require execution of the generated code and comparison of the outputs against the standard answer and the outputs from the reference solution. In addition to these quality attributes, they also measured the *task completion time* and the *number of attempts* that the user queries ChatGPT till a satisfactory solution is obtained.

Another aspect of usability that Miah and Zhu studied [32] is *learnability*, which refers to how easy the user can learn to use the tool. They found that users can hardly improve their skills of using ChatGPT through experiences.

Ziegler et al. investigated various aspects of productivity, including *task completion time*, *product quality*, *cognitive load*, *enjoyment*, and *learning*, in their evaluation of GitHub's Copilot [43]. They measured objective observations on the usage of Copilot and compared this with questionnaire surveys filled out by the users. They found that the acceptance rate of shown solutions is a better productivity predictor than other metrics.

In [44], Xu et al. studied the usability of a plug-in to Python's PyCharm IDE, which enables both code generation based on a ML model and code retrieval via a search of the internet. They conducted controlled experiments with human users to compare the impact of both of these on productivity in terms of *task completion time* and quality of the result code in terms of *correctness score*, which is assessed manually according to a marking rubric. They also used the lengths of initial and final l codes and the *editing distances* to measure the quality of the generated/retrieved code.

D. Multi-Trial vs Multi-Attempt Metrics

In [9], Kulal et al. asked the ML model to generate 100 solutions on each coding task, and regarded the code generation as successful if at least one solution passed all test cases. This was later generalised to requiring $k > 0$ solutions for each task, leading to the $pass@k$ metric, which is the probability of generating at least one solution successfully in k trials. Chen et al. [20] found that a straightforward calculation of the metric by its definition produces a high variance, however, so instead of recording whether there is a successful solution or not, they count the number c of successful solutions in k and use c and k to make an unbiased estimation of the $pass@k$ metric. This approach has been used by most of the benchmarks reported above, including ClassEval, MBPP, MathQA-Python, CoderEval, and HumanEval+.

The $pass@k$ metrics is applicable to the testing and evaluation processes where the ML model is tried multiple times on each test case. Thus, it is called a *multi-trial metric*.

Miah and Zhu [32], in contrast, consider the task of code generation from an LLM model, ChatGPT specifically, to be an interactive process in which the user makes a number of attempts by entering and amending their input to the LLM until a successful solution is generated, or the user gives up after a certain number k of allowed attempts without obtain a satisfactory solution. They proposed a new metric $\#attempt_k$ that measures the average number of attempts and found in their experiments that satisfactory solutions from ChatGPT can be obtained with 1.6 attempts on average.

Table V provides detailed information regarding the benchmarks used in the evaluation, the metrics used and the main results.

V. Research Directions

In the past few years, significant progress has been made both in the development of benchmarks for code generation and in techniques for evaluation. However, there are a few problems that require further research.

First, as shown in Table II, most benchmarks were constructed from a single source, so they lack diversity and their distribution may be skewed towards certain types of questions.

Second, using tests to judge correctness of a generated solution has the advantages of being objective and automatic. However, the accuracy of the judgement depends heavily on test adequacy. This has, however, been reported in more recent benchmarks. For example, ClassEval's test suites have an average of 98.2% branch coverage and 99.7% statement coverage. CoderEval aims to provide full branch coverage. HumanEval+ claims to encompass all possible corner cases. A further problem of judging correctness by test results that has not been noticed by the research community is that prescripted test cases can detect only errors of omission, as opposed to errors of commission, which can include malicious code.

The overall performance metric $pass@k$, the probability of getting at least one correct solution in k outputs, reflects the randomness of the LLM output. However, as Miah and Zhu pointed out, users do not normally run the LLM several times so $pass@k$ does not reflect its usability. The $\#attempts_k$ metric seems to be a better fit but it heavily relies on manual assessment and subjective judgement so it is difficult to apply to large scale experiments.

The validity problems of syntax similarity metrics were studied in ML models before LLMs were introduced. It is possible that these problems still exist. Ideally, these metrics should be modified to measure usability as this is more important than correctness for LLMs [43].

Finally, with the exception of the R-benchmark [32], existing benchmarks do not support scenario-based evaluation. A single evaluation score is given and that does not tell the developer how to improve the model. The problem is how to perform scenario-based evaluation effectively and efficiently

TABLE V
EVALUATION PER BENCHMARK

Benchmark	Correctness	Metric	Model	Result
APPS	pass all tests	%pass@100	GPT-2 1.5B	0.68
			GPT-Neo 2.7B	1.12
			GPT-3 175B	0.06
	test pass rate	AvgTPR	GPT-2 1.5B	7.96
			GPT-Neo 2.7B	10.15
			GPT-3 175B	0.55
HumanEval	pass all tests	%pass@100	GPT-Neo-2.7B	21.37
			GPT-J-6B	27.74
			Tabnine	7.59
			Codex-12B	72.31
MBPP	pass all tests	%pass@1	Decoder only-8B	79.0
			Transformer-68B	82.8
			Lang. Model-137B	83.8
MathQA -Python	pass all tests	%pass@1	Decoder only-8B	74.7
			Transformer-68B	79.5
			Lang. Model-137B	81.2
ClassEval	pass all tests	%pass@5	GPT-4	42.0
			GPT-3.5	36.0
			WizardCoder	23.0
			StarCoder	14.0
			SantaCoder	10.0
			CodeGen	13.0
			CodeGeeX	10.0
			InCoder	8.0
			Vicuna	4.0
			ChatGLM	3.0
			PolyCoder	3.0
CoderEval (Python)	pass all tests	%pass@10	CodeGen	23.48
			PanGu-Coder	27.39
			ChatGPT	30.00
CoderEval (Java)	pass all tests	%pass@10	CodeGen	33.48
			PanGu-Coder	43.04
			ChatGPT	46.09
Multipl-E (HumanEval)	pass all tests	%pass@1	Codex	$\approx$ 36
			CodeGen	$\approx$ 9
			InCoder	$\approx$ 6
Multipl-E (MBPP)	pass all tests	%pass@1	Codex	$\approx$40
			CodeGen	$\approx$14
			InCoder	$\approx$ 15
DS-1000	pass all tests	%pass@1	Codex-002	41.25
			CodeGen-6B	8.4
			InCoder-6B	7.45
HumanEval+	pass all tests	%pass@100	CodeGen	$\approx$64.0
			SantaCoder	40.6
			InCoder	$\approx$29.8
			PolyCoder	13.6
			ChatGPT	89.8
			Vicuna	40.25
			StableLM-α	11.9
			GPT-J	25.9
			GPT-Neo	16.8
ConCode	BLEU	Avg BLEU	Retrieval	20.27
			Seq2Seq	23.51
			Seq2Prod	21.29
			User-designed	22.11
R-benchmark	Satisfactory	Avg #attempt$_k$	ChatGPT	1.6

so that better feedback can be given. Including metadata in benchmarks seems to be a promising approach. However, manually assigning this metadata to coding tasks in a large scale dataset like in [32] is labour intensive, time consuming and costly. The challenge is to do it automatically.

VI. CONCLUSION

Evaluation of LLMs as intelligent code generation tools is still a grave challenge. There are many open problems in the construction of benchmarks and the definition and implementation of performance metrics despite of the great efforts reported recently in the literature. Among the most important problems to be solved are the development of performance metrics that reflect ML model's usability, the validation of the metrics, the construction of benchmarks that are versatile and feasible to use, and the techniques and tools that enable the automation of evaluation.

REFERENCES

[1] A. Vaswani, et al., "Attention is all you need,"*Advances in Neural Information Processing Systems*, vol. 30, 2017.

[2] B. Marr, "How Generative AI Will Change The Jobs Of Computer Programmers And Software Engineers, "*FORBES, INNOVATION, ENTERPRISE TECH*, May 30, 2024.

[3] Y. Wan, et al., "Deep Learning for Code Intelligence: Survey, Benchmark and Toolkit," *ACM Computing Surveys*, Dec. 30, 2023.

[4] A. Odeh, N. Odeh, and A. S. Mohammed, "A Comparative Review of AI Techniques for Automated Code Generation in Software Development: Advancements, Challenges, and Future Directions," *TEM Journal*, vol. 13, no. 1, pp. 726-739, Feb. 2024.

[5] J. L. Espejel, et al., "A comprehensive review of state-of-the-art methods for Java code generation from natural language text," *Natural Language Processing Journal*, vol. 3, article 100013, 2023.

[6] R. Balzer, "A 15 year perspective on automatic programming," *IEEE Trans. Softw. Eng.*, vol. 11, pp. 1257–1268, 1985.

[7] H. Zhu and L. Jin, "A knowledge-based system to synthesize FP programs from examples," in *Proc. of the 4th Portuguese Conference on Artificial Intelligence (EPIA 1989)*, Springer LNCS, vol. 390, pp. 234-245, Sept., 1989.

[8] E. Dehaerne, et al., "Code generation using machine learning: A systematic review," *IEEE Access*, vol. 10, Aug. 2022, pp. 82434-55.

[9] S. Kulal. et al., "Spoc: Search-based pseudocode to code," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[10] C.Y. Su and C. McMillan, "Distilled GPT for source code summarization," *Automated Software Engineering*, vol. 31, no. 1, p. 22, May 2024.

[11] L. Zhao, L. Zhang, and S. Yan, "A survey on research of code comment auto generation," *Journal of Physics: Conference Series*, vol. 1345, no. 3, p. 032010, Nov. 2019.

[12] R. Li, et al., "StarCoder: may the source be with you!," *arXiv: 2305.06161*, 2023.

[13] P. Naik, S. Nelaballi, V.S. Pusuluri, and D.K. Kim, "Deep learning-based code refactoring: A review of current knowledge," *Journal of Computer Information Systems*, vol. 64, no. 2, pp. 314-328, Mar. 2024.

[14] J. Renzullo, P. Reiter, W. Weimer, and S. Forrest, "Automated Program Repair: Emerging trends pose and expose problems for benchmarks," *arXiv:2405.05455*, May 8, 2024.

[15] S. Ajorloo, et al., "A systematic review of machine learning methods in software testing," *Applied Soft Computing*, vol. 162, p. 111805, Sept. 2024.

[16] D.M. Zapkus and A. Slotkienė, "Unit Test Generation Using Large Language Models: A Systematic Literature Review," *Vilnius University Open Series*, May, 2024, pp. 136-144.

[17] Gemini Team Google, "Gemini: A Family of Highly Capable Multimodal Models," *arXiv:2312.11805*, 2024.

[18] S. Black, et al., "GPT-Neo: Large scale autoregressive language modeling with Mesh-Tensorflow," 2021.

[19] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 billion parameter autoregressive language model," 2021.

[20] M. Chen, et al., "Evaluating large language models trained on code," *arXiv:2107.03374*, 2021.

[21] E. Nijkamp, et al., "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2023.

[22] L. B. Allal, et al., "SantaCoder: don't reach for the stars!," *arXiv: 2301.03988*, 2023.

[23] D. Fried, et al., "Incoder: A generative model for code infilling and synthesis," in *The Eleventh International Conference on Learning Representations*, 2023.

[24] D. Hendrycks, et al., "Measuring Coding Challenge Competence With APPS," in *Proc. of 35th Conference on Neural Information Processing Systems - Datasets and Benchmarks Track*, 2021. Also available as *arXiv:2105.09938*.

[25] J. Austin, et al., "Program synthesis with large language models," *arXiv:2108.07732*, 2021.

[26] X. Du, et al., "ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation," *arXiv:2308.01861*, Aug. 2023.

[27] H. Yu, et al., "CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models," In *Proc. of the IEEE/ACM 46th International Conference on Software Engineering* (ICSE'24), Article 37, pp1-12, Feb. 2024.

[28] F. Cassano, et al., "MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, July 2023.

[29] Y. Lai, et al., "DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation," in *Proc. of the 40th International Conference on Machine Learning*, PMLR 202, Honolulu, Hawaii, USA, 2023.

[30] J. Liu, et al., "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," *arXiv:2107.03374*, Jul, 2021.

[31] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping Language to Code in Programmatic Context," in *Proc. of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp.1643-1652, 2018.

[32] T. Miah and H. Zhu, "User-Centric Evaluation of ChatGPT Capability of Generating R Program Code," *arXiv:2402.03130*, Feb. 2024.

[33] R. Agashe, S. Iyer, and L. Zettlemoyer, "JuICe: A Large Scale Distantly Supervised Dataset for Open Domain Context-based Code Generation," in *Proc. of the 2019 Conference on Empirical Methods in Natural Language Processing and The 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Association for Computational Linguistics, 2019, pp.5436-5446.

[34] Y. Xie, et al., "CodeBenchGen: Creating Scalable Execution-based Code Generation Benchmarks," *arXiv:2404.00566*, May, 2024.

[35] J. Li, et al., "EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories," *arXiv:2404.00599*, Mar. 2024.

[36] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a Method for Automatic Evaluation of Machine Translation," in *Proc. of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, Philadelphia, PA, July 2002, pp.311-318.

[37] Chin-Yew Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," in *Text Summarization Branches Out*, Barcelona, Spain, Association for Computational Linguistics, 2004, pp. 74-81.

[38] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proc. of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, 2005, pp. 65-72.

[39] M. Denkowski and A. Lavie, "Meteor universal: Language specific translation evaluation for any target language," in *Proc. of the Ninth Workshop on Statistical Machine Translation*, 2014, pp. 376-380.

[40] M. Popović, "Chrf: character n-gram F-score for automatic MT evaluation," in *Proc. of the 10th Workshop on Statistical Machine Translation*, 2015, pp. 392-395.

[41] N. Tran, et al., "Does BLEU score work for code migration?" in *Proc. of IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 165-176.

[42] S. Ren, et al., "CodeBLEU: a method for automatic evaluation of code synthesis," *CoRR 2020. arXiv:2009.10297*, Sept. 2020.

[43] A. Ziegler, et al., "Measuring GitHub Copilot's Impact on Productivity," *Communications of the ACM*, vol. 67, no. 3, pp. 54-63, Feb. 2024.

[44] F. F. Xu, B. Vasilescu, and G. Neubig, "In-IDE code generation from natural language: Promise and challenges," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1-47, Mar. 2022.

[45] M. Evtikhiev, E. Bogomolov, Y. Sokolov, and T. Bryksin, "Out of the BLEU: how should we assess quality of the code generation models?," *Journal of Systems and Software*, vol. 203, article 111741, Sept. 2023.