The Fault in our Stars: Quality Assessment of Code Generation Benchmarks

Mohammed Latif Siddiq*, Simantika Dristi†§, Joy Saha†§, Joanna C. S. Santos*
*Department of Computer Science and Engineering, University of Notre Dame, USA
†Department of Computer Science, BRAC University, Dhaka, Bangladesh
msiddiq3@nd.edu, {simantika.dristi,joy.saha}@bracu.ac.bd, joannacss@nd.edu

***Abstract*—Large Language Models (LLMs) are gaining popularity among software engineers. A crucial aspect of developing effective code generation LLMs is to evaluate these models using a robust benchmark. Evaluation benchmarks with quality issues can provide a false sense of performance. In this work, we conduct the first-of-its-kind study of the quality of prompts within benchmarks used to compare the performance of different code generation models. To conduct this study, we analyzed 3,566 prompts from 9 code generation benchmarks to identify quality issues in them. We also investigated whether fixing the identified quality issues in the benchmarks' prompts affects a model's performance. We also studied memorization issues of the evaluation dataset, which can put into question a benchmark's trustworthiness. We found that code generation evaluation benchmarks mainly focused on Python and coding exercises and had very limited contextual dependencies to challenge the model. These datasets and the developers' prompts suffer from quality issues like spelling and grammatical errors, unclear sentences to express developers' intent, and not using proper documentation style. Fixing all these issues in the benchmarks can lead to a better performance for Python code generation, but not a significant improvement was observed for Java code generation. We also found evidence that GPT-3.5-Turbo and CodeGen-2.5 models may have data contamination issues.**

***Index Terms*—benchmarks, code generation, data quality, data contamination**

I. Introduction

Code generation models generate code by taking as input a *prompt*, which captures the developers' intent [1]. These models are increasingly popular among software developers [2]. In fact, a recent survey with 500 US-based developers who work for large-sized companies showed that **92%** of them are using AI-based code generation tools both for work and personal use [3]. Part of this fast widespread adoption is due to the increased productivity perceived by developers [4]; AI helps them to automate repetitive tasks so that they can focus on higher-level challenging tasks [5].

As code generation models are becoming ubiquitous during software development [6], the need for reliable ***evaluation benchmarks*** is vital. Code generation benchmarks are crucial for evaluating and comparing the effectiveness of various models in producing code. These benchmarks are designed to assess the generated codes from various perspectives, such as their *correctness*, *readability*, and *security* [7].

While there are over **15** evaluation benchmarks for code generation models [7], their *quality* and *reliability* are currently unclear. First, these benchmarks are often collected in an *ad-hoc* fashion, which may not be representative of real software scenarios [8]. Second, as these benchmarks are curated from publicly available data, there is the risk that existing models include data from these benchmarks in the training set (*i.e.*, *test set contamination* [9]). In this case, the reliability of `pass@k` [10] and other performance metrics is put into question, as the models might have memorized the solutions to the prompts in the dataset [11]. Therefore, issues on these benchmarks significantly impact the trustworthiness of the evaluation results, making it crucial to thoroughly investigate and evaluate the benchmarks themselves.

In light of this research gap, we present an ***empirical study of the quality of prompts in benchmarks from multiple dimensions and compare them with issues observed in real world prompts created by developers***. Specifically, we systematically analyzed **3,566** prompts from **9** Java and Python benchmarks to check the quality issues in these prompts. We observed that issues fall into three categories: formatting, a prompt containing sentences that do not properly (or incorrectly) specify the behavior of the code, and prompts containing unnecessary tokens (*noise*). Besides identify quality issues in the benchmarks' prompts, we also explored to what extent these issues affect a model's performance. In doing so, we found that fixing spelling and grammatical issues and using standard JavaDoc and docstring style can help models to generate code. Last, but not least, we studied whether existing models are memorizing answers from existing benchmarks (*i.e.*, test set contamination [11], [12]). In our experiments, we found empirical evidence of testset contamination in two models: CodeGen-2.5 and GPT-3.5.

The contributions of this paper are:

- A thorough investigation of code generation benchmarks' prompts (**RQ1** and **RQ2**) so researchers and developers can make informed decisions about choosing a benchmark to evaluate code generation models.
- A study of how fixing quality issues in a prompt can affect a model's evaluation (**RQ3**).

– An investigation of possible test-set contamination issues in HumanEval, a popular benchmark (**RQ4**).
– A comparison of the quality issues observed in benchmarks' prompts and the real world prompts made by developers when interacting with ChatGPT [13] (**RQ5**).

This paper's replication package is available in [14].

II. Background

A. Large Language Models

Large Language Models (**LLMs**) [15] are neural networks with tens of millions to billions of parameters that were trained on large amounts of unlabeled text using self-supervised or semi-supervised learning [16]. LLMs are intended to be general purpose models for many natural language processing tasks, such as text generation, translation, summarization, *etc.* While LLMs are trained to understand *natural* language, they can be fine-tuned with source code samples to understand *programming* languages. This makes LLMs useful for a myriad of software engineering tasks, such as code completion [17]–[20], and summarization [21]. CodeBERT [20], CodeT5 [22], and Codex [10] are examples of "**code LLMs**", *i.e.*, LLMs trained on source code (henceforth, simply "LLMs").

Given a *prompt* as input, a code LLM generates code tokens, one by one, until it reaches a *stop sequence* (*i.e.*, a pre-configured token sequence) or the *maximum number of tokens* is reached. A **prompt** provides a high-level specification of a developer's intent and can include different code elements, *e.g.*, function signatures, expressions, comments, *etc.*

Transformer-based code generation models employ masked language modeling objectives or *left-to-right* (causal) autoregressive language modeling objectives [10], [16]. That is, to generate code, the generative model will use the context on the *left* side of the cursor and ignore any context on the *right*. The process of creating code that incorporates context from *both* sides is known as **code infilling**. In this work, we focus on studying **left-to-right** code generation benchmarks because the majority of benchmarks are meant to evaluate left-to-right code generation [7]. For instant, we considered 9 benchmarks in our study out of 17 benchmark studied in this survey by Daoguang *et al.* [7].

B. Code Generation Benchmarks

Code generation benchmarks are used to evaluate and compare models based on different metrics [10], [23]. Existing benchmarks usually contain coding problems captured in a natural language, comment, or combination of comment and code, referred to as a **prompt** [1]. After using the prompt for code generation, different metrics can be used to evaluate the performance. For example, `CodeBleu` [24] can be used for syntactical correctness, and `pass@k` [10] can be used for functional correctness. Benchmarks may be created for a specific purpose. For example, SALLM [25] focuses on evaluating the security of generated code and uses the `vulnerable@k` to compare the performance of the models.

C. Memorization in LLMs

Memorization refers to a model's ability to preserve and generate an identical string from its training data [9], [26]. In this work, we used a similar definition as the one presented by Carlini *et al.* [26]. Specifically, if there exists a prompt that generates a code snippet that completely matches any of its training data code snippets, then this code snippet is considered to be memorized by the code generation model, a case of **verbatim memorization** [27].

In light of this definition and similar to prior work [28], we study whether test set contamination by verifying whether the generated code is a *clone* of the solution available in the benchmark. Specifically, in our work, we search for *type-1*, *type-2*, and *type-3* code clones to pinpoint memorization [29]. A **type 1** clone occurs when two code snippets have identical code fragments except for layout, comments, and whitespace differences. A **type 2** clone arises when there are syntactically identical fragments except for comments, whitespace, literals, identifiers, and types. A **type 3** clone means that there are copied fragments that have undergone additional changes, such as additions, deletions, or changes to statements, as well as adjustments to identifiers, literals, types, whitespace, layout, and comments.

III. Methodology

In this paper, we answer the following questions:

RQ1 ***How representative are existing benchmarks of real-world code generation usage scenarios?***

Code generation models need rigorous evaluation and verification. However, existing benchmarks may not represent real-world scenarios and cover many programming languages. In this question, we compare the code generation benchmarks' *covered programming languages*, *usage scenario(s)*, *number of prompts*, and *contextual dependency complexity*.

RQ2 ***What are the quality issues in the prompts within code generation benchmarks?***

In this RQ, we study quality issues in the prompts of code generation benchmarks. To do so, we manually analyzed a total of **3,566** from **9** code generation benchmarks. We performed *open coding* [30] of the prompts in these benchmarks to identify and categorize quality issues.

RQ3 ***Does improving the quality of a prompt in code generation benchmarks affect the evaluation result?***

We investigate whether improving the quality of a benchmark's prompts affects the results of code generation models. To do so, we fixed quality issues identified in **RQ2** and compared the performance of LLMs when given as input the *fixed* prompts and the *original* prompts with quality issues.

RQ4 ***Are there contamination issues in existing code generation benchmarks?***

Since code generation models are fine-tuned on source code from open-source repositories, there is a risk that code from evaluation benchmarks are in the models' training set. If prompts from benchmarks are in the training set, the code generation model can perform better because it has memorized the answer [11]. Hence, this contamination issue will affect the code generation model's benchmarking process. In this RQ, we explore the possibility of contamination issues in existing code generation benchmarks.

RQ5 ***Are the quality issues in the benchmarks' prompts similar to issues observed in real world prompts?***

In this RQ, we explore whether the quality issues in the benchmarks' prompts are similar to the ones that are observed in the real world, *i.e.*, from developers using LLMs in their day-to-day development activities. To answer this question, we extracted prompts from *DevGPT* [31], a dataset that contains the chats from software developers interacting with ChatGPT [13]. This dataset was curated by finding *ChatGPT share links* that were posted on GitHub *issues*, *pull requests*, *discussions*, *commits*, *code files*, and *threads* on Hacker News.

We detail how we answer each RQ in the next sections.

A. RQ1: Code Generation Benchmarks Comparison

To answer RQ1, we first collected 17 benchmarks listed in a recent survey [7]. Since we focus on *left-to-right* code generation, we disregard benchmarks designed to evaluate code-infilling models. This way, we obtained a total of **9** benchmarks: MXEVAL [23], CODEREVAL [32], ODEX [33], MBPP [34], TORCHDATAEVAL [35], HUMANEVAL [10], PANDASEVAL [36], NUMPYEVAL [36], and JIGSAWDATASET [37]. MXEVAL [23] is a benchmark that extends the MATHQA [38], MBPP [34], and HUMANEVAL [10] benchmarks.

To verify a benchmark's potential of representing real-world code generation scenarios, we analyzed each benchmark to identify their **(i)** ***covered programming language(s)***, **(ii)** ***usage scenarios***, **(iii)** ***number of prompts***, and **(iv)** ***contextual dependency complexity***. We classify a benchmark's *contextual dependency complexity* based on the categorization scheme described by Yu *et al.* [8]. This complexity can be:

- **self-contained**: benchmarks whose solution to the prompt can be implemented using only built-in classes/modules that do not need to be imported (*e.g.*, Java's String class does not need to be imported to be used).
- **slib-runnable**: benchmarks where the solution to the prompts needs to import classes/modules that are provided by the language and do not require further installation (*e.g.*, Java's `java.util` package and Python's `re` module).
- **plib-runnable**: benchmarks in which the prompts' solutions only use libraries that are publicly available on PyPi or Maven central (*e.g.*, Apache Log4j).
- **class-runnable**: benchmarks in which the solution uses code elements (*e.g.*, methods, objects) that are declared outside the prompt's method but within the prompt's class.
- **file-runnable**: benchmarks in which the generated solution uses code elements *outside* its class, but that is still declared on the same file as the prompt.
- **project-runnable**: benchmarks in which the generated code uses code elements declared in other source files in the benchmark.

To answer RQ1, we created a ***benchmark profile*** identifying the information above for each benchmark. This benchmark profile was created by examining the benchmarks' original paper and technical documentation to identify the metadata **(i)–(iv)** listed above.

B. RQ2: Benchmark Quality Evaluation

Since there were over 8,000 prompts in total in the studied benchmarks, we first randomly sample prompts from each chosen benchmark with a **99%** confidence interval and a **5%** margin of error. As shown in Table I, we analyzed a total of **3,566** prompts from **9** benchmarks.

TABLE I
TOTAL NUMBER OF PROMPTS AND SAMPLED PROMPTS PER BENCHMARK.

Benchmark	# Prompts	#Sampled Prompts	Benchmark	# Prompts	#Sampled Prompts
MXEVAL [23]	6,031	2,037	**MBPP** [34]	426	261
MBPP	1,940	791	**TorchDataEval** [35]	302	270
HumanEval	325	262	**HumanEval** [10]	164	132
MathQA	3,766	984	**PandasEval** [36]	101	88
CoderEval [32]	460	342	**NumpyEval** [36]	101	88
ODEX [33]	439	265	**JigsawDataset** [37]	88	83

In our study, we focused on Java and Python prompts because not only these are popular languages among developers [39], but they are also the most supported language in benchmark datasets (§ IV-A). Thus, we systematically analyzed the benchmarks' Python/Java prompts to identify *quality issues*. This qualitative analysis was performed by two of the authors, with over two years of software development and teaching experience each. Each author independently performed open coding [30] of each prompt. The open coding started with a (initially empty) shared "code book" where we progressively captured the issue's *title* and *description* with *examples* as we analyzed prompts. Our code book was constantly refined throughout the open coding process.

After each author finished the open coding, a third author, who has over three years of professional programming experience, resolved the discrepancies through discussion and mediation. This analysis took us approximately **650** person-hours. We calculated the Cohen's Kappa coefficient to measure the inter-

rater reliability of this analysis, and it was **0.76**, which indicates ***substantial agreement*** [40].

C. RQ3: Impact on Performance

In RQ2, we identified the quality issues in the benchmarks' prompts. In RQ3, we fixed the issues to verify to what extent fixing them affects the performance of the models. As it would be time-consuming to manually fix thousands of prompts, we fixed the issues identified for the Python and Java version of the HUMANEVAL benchmark [10], [23]. We chose this benchmark because most of the code generation models are evaluated with it, as shown in a popular leaderboard published on PapersWithCode.com [41] which lists over 120 code LLMs that were evaluated with HUMANEVAL.

To conduct this investigation, the same two authors who have done the open coding in RQ2 went through all the issues identified in RQ2 for the **164** prompts from HUMANEVAL Python [10] and **161** prompts from HUMANEVAL's Java version [23]. For each identified issue, we created a set of *fix guidelines* that was shared among both researchers. Since a prompt can have *more than one* quality issue, the authors first fixed the issues *one by one* by creating a modified prompt version that does not contain *one* particular type of problem. Subsequently, they created a new prompt version that fixed *all* the quality issues in the prompt. Once both authors fixed all prompts, they peer-reviewed each other's fixes and came up with a final fix. To mitigate subjectivity when fixing issues, the senior author, who has over 10 years of experience, checked the modification and updated the prompts in multiple rounds of discussions. After that, we have 501 prompts for Java and 663 prompts for Python, including the original prompts.

After fixing these issues, we give the prompts (*fixed* and *original* ones) to five code generation LLMs:

- **CODEGEN** [42] is a code LLM that has three variants: *CodeGen-nl*, *CodeGen-multi*, and *CodeGen-mono*. CODEGEN-NL, trained with the *Pile* dataset [43], is focused on text generation. The *CodeGen-multi* is built on top of *CodeGen-nl* but further trained with a large scale-dataset of code snippets in six different languages (*i.e.*, C, C++, Go, Java, JavaScript, and Python) [44]. The *CodeGen-mono* is built from *CodeGen-multi* and further trained with a dataset of only Python code snippets [42]. They also released another (newer) version called *CodeGen-2.5* [45], which is trained on the StarCoder data from BigCode [46]. It has a mono and multi-version. We use **CODEGEN-2.5-7B-MONO** to generate Python code and **CODEGEN-2.5-7B-MULTI** to generate Java code.
- **SANTACODER** [47] is a 1.1B parameter LLM trained on the Java, JavaScript, and Python subsets of The Stack [46] dataset. It can do both *left-to-right* generation and *infilling*.
- **STARCODER** [48] is an LLM with 15.5B parameters trained with over 80 different programming languages. This model is focused on fill-in-the-middle objectives and can complete code given a code-based prompt. It has two versions for code generation: *StarCoderBase* and *StarCoder*. As the latter one is further trained with Python samples, we used that for Python and the former for Java code generation.
- **WIZARDCODER** [49] is an instruct-tuned version of STARCODER [48] model using Evol-Instruct method on the code domain. This model can generate both code and follow complex instructions.
- The **GENERATIVE PRE-TRAINED MODEL (GPT)** [16] is a family of transformer-based [50] and task-agnostic LLMs that can *understand* and *generate* natural language. We used **GPT-3.5-TURBO**, which is tuned for chat-style conversation and powers a popular chat-based question-answering tool, ChatGPT [13].

We chose these models because they are representative of code generation LLMs. GPT-3 is used on popular code generation tools, such as GitHub Copilot [51] and ChatGPT [13]. CodeGen-2.5, SantaCoder, StarCoder, and WizardCoder are open-source top-performing code LLMs [41], [49].

For each model, we generated **20** codes with a maximum of *t* new tokens for each prompt. To choose a suitable value of maximum numbers of tokens *t*, we calculated the size of canonical solutions for the HUMANEVAL's problems [10]. We found that the average solution has 54 tokens (maximum of 240 tokens). Hence, we asked each LLM to generate **512** new tokens (*i.e.*, *t* is $10\times$ the average canonical solution's length). Then, we calculated the `pass@k` metric by running the test cases for each output.

Computing pass@k: Code LLMs are commonly evaluated using `pass@k` [10], [52]. This metric estimates the probability that *at least one* out of k generated samples is correct (*i.e.*, passes all the prompt's test cases). This metric is computed by generating n samples per prompt ($n \geq k$), counting the number of samples c that are correct ($c \leq n$), and calculating the unbiased estimator from Kulal *et al.* [52]:

$$pass@k = \mathbb{E}_{prompts}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \quad (1)$$

We set k to 1, 3, and 10 and generated $n = 20$ outputs for each prompt. We used temperature **1.0** for GPT-3.5-Turbo [16] for all `pass@k`, **0.2** for `pass@1` and **0.6** for `pass@3` and `pass@10` for CodeGen model [45], and **0.2** for `pass@1` and **0.8** for `pass@3` and `pass@10` for the other open source models. We chose these temperatures as these were the ones that were reported in the models' corresponding papers. In this evaluation, we compared the models' `pass@k` when provided with the *original* prompt and its *fixed* versions.

D. RQ4: Contamination Issues

In this question, we checked the contamination issue of the widely used HumanEval's Python version benchmark [10]. We used this benchmark because it is the only one that has the canonical solution, *i.e.*, it is at a higher risk of the contamination issue [11]. To answer RQ4, we ran NiCad

TABLE II
CHARACTERISTICS OF THE STUDIED BENCHMARKS

Name	# Prompts	Target Code Language	Usage Scenario	Contextual Dependency Complexity
CoderEval [32]	460	Python, Java	Pragmatic Code Generation	self-contained
HumanEval [10]	164	Python	Code Exercise	slib-runnable
JigsawDataset [37]	87	Python	Public Library	
MBPP [34]	974	Python	Code Exercise	self-contained
MXEVAL [23]	16,171	C#, C++, Go, Java, JavaScript, Kotlin, Perl, PHP, Python, Ruby, Scala, Swift, TypeScript	Code/Math Exercises	slib-runnable, self-contained
MBPP	8,588	*All of the 13 languages listed above*	Code Exercises	slib-runnable, self-contained
HumanEval	1,934	*All except C++*	Code Exercises	slib-runnable
MathQA	5,649	*Only Python, Java, and JavaScript*	Math Exercises	slib-runnable, self-contained
NumpyEval [36]	101	Python	Single Public Library	plib-runnable
ODEX [33]	945	Python	Open Domain	self-contained
PandasEval [36]	101	Python	Single Public Library	plib-runnable
TorchDataEval [35]	50	Python	Private Library	plib-runnable

(Automated Detection of Near-Miss Intentional Clones), a state-of-the-art code clone detection tool [53], on all code generated by each model. NiCad can detect different types of clones, including *Type-1*, *Type-2*, and *Type-3* clones.

We first removed all the prompt's comments (docstring and in-line) from both the canonical solutions and the generated code. Then, we used NiCad cross-clone detection mechanism, which can find clones between two systems. In our case, the first system is the source codes from canonical solutions, and the second is the generated codes from the models. We compared the canonical solutions not only with the generated codes from the *original* prompts but also with the generated codes from the modified prompts that fix *all* issues in RQ3. As the prompts were modified by us, they would not be a part of the training set, though they are similar to the original prompts. Hence, comparing the results with the original and the modified can provide us with more insights into the data contamination issue.

We configured NiCad [53] to find clones in the function labels, as the HumanEval dataset consists of prompts for completing functions. For Type-2 and Type-3 clone detection, we kept the default maximum difference threshold to 30%. We configured the minimum line number of clones based on the size of the canonical solution line number. That is, the *minimum* number of lines is set to be half of the number of lines in the canonical solution. It is worth highlighting that since NiCad can detect clones with at least 5 lines, we kept the threshold set to 5 in case the canonical solution had less than five lines.

Similar to a prior work [28], we used code clones as a means to identify cases where the generated code is identical (*i.e.*, a clone) to the solution. If a code clone is detected, the model likely has memorized the solution. Hence, to identify potential contamination issues, we computed the percentage of different types of clones, including clones from the original and the modified prompts in the previous research questions. Notably, the Type-2 clone results from NiCad include Type-1 and Type-3 clone results, which include Type-1 and Type-2, and the result is kept as it is.

E. RQ5: Quality Assessment of Developers' Prompts

To investigate whether the quality issues observed in benchmark datasets (RQ2) are similar to the ones observed in real prompts, we have collected from the DevGPT dataset **3,995** publicly shared unique ChatGPT conversation links that are mentioned in code comments, commits, pull requests, discussions on GitHub, and threads on HackerNews [31]. Next, we discarded 217 links that were no longer accessible. Then, we manually inspected each conversation to keep only those in which developers asked for code and ChatGPT generated one or more code snippets according to the developers' prompts. That is, we discarded conversations in which ChatGPT did not generate code and/or the generated code was not in Java or Python. As a result, we had **371** ChatGPT conversation links that had generated Python/Java code. Two authors (the same ones from RQ2 & RQ3) did the open coding of these 371 conversations. This open coding process employed the same methodology as RQ2 (§ III-B). Then, a senior author, who has more than ten years of experience, resolved any discrepancies. The Cohen's Kappa score to measure our inter-rater agreement was **0.60**, which indicates *substantial agreement* [40].

IV. RESULTS

This section presents the results for each RQ.

A. RQ1: Code Generation Benchmarks Comparison

As shown in Table II, the studied benchmarks have an average of **516** prompts per language. In terms of *supported programming languages*, we found that **all** benchmarks included prompts to generate Python code, and only **2** out of **9** benchmarks included other languages besides Python. The MXEVAL [23] is a benchmark that extends three other benchmarks

TABLE III
QUALITY ISSUES IN EACH BENCHMARK (THE PERCENTAGES ARE THE PERCENT PROMPTS WITH THE ISSUE IN THE BENCHMARK).

Type	Quality Issue	# Prompts	Benchmark	%	Benchmark	%	Benchmark	%
Intent	**Q.I.1: Function/method's name mismatches with its intent**	1,321	CODEREVAL	0.6%	HUMANEVAL	9.1%	MBPP	3.1%
			MXEVAL$_{\text{HUMANEVAL}}$	8.8%	MXEVAL$_{\text{MBPP}}$	3.2%	MXEVAL$_{\text{MATHQA}}$	100.0%
			NUMPYEVAL	1.1%	ODEX	100.0%	TORCHDATAEVAL	0.4%
	Q.I.2: Spelling and grammatical errors	303	CODEREVAL	8.2%	HUMANEVAL	17.4%	JIGSAWDATASET	9.6%
			MBPP	2.7%	MXEVAL$_{\text{HUMANEVAL}}$	17.6%	MXEVAL$_{\text{MBPP}}$	4.7%
			MXEVAL$_{\text{MATHQA}}$	6.7%	NUMPYEVAL	10.2%	ODEX	4.2%
			PANDASEVAL	13.6%	TORCHDATAEVAL	20.7%		
	Q.I.3: The prompt description is unclear	124	CODEREVAL	2.9%	HUMANEVAL	1.5%	MXEVAL$_{\text{HUMANEVAL}}$	0.4%
			MXEVAL$_{\text{MBPP}}$	6.3%	MXEVAL$_{\text{MATHQA}}$	4.2%	NUMPYEVAL	5.7%
			ODEX	1.9%	PANDASEVAL	3.4%	TORCHDATAEVAL	2.6%
	Q.I.4: Partial or incomplete sentence	132	MXEVAL$_{\text{MATHQA}}$	13.3%	ODEX	0.4%		
	Q.I.5: Incorrect input/output pair example	15	HUMANEVAL	1.5%	MXEVAL$_{\text{HUMANEVAL}}$	1.5%	MXEVAL$_{\text{MBPP}}$	1.1%
Formatting	**Q.I.6: JavaDoc/docstring has formatting issues**	1,835	HUMANEVAL	22.0%	MXEVAL$_{\text{HUMANEVAL}}$	81.7%	MXEVAL$_{\text{MBPP}}$	89.5%
			MXEVAL$_{\text{MATHQA}}$	89.2%	NUMPYEVAL	1.1%	PANDASEVAL	2.3%
			TORCHDATAEVAL	1.1%				
	Q.I.7: Not using JavaDoc (Java) or docstring (Python) on the prompt	439	HUMANEVAL	2.3%	MXEVAL$_{\text{HUMANEVAL}}$	0.8%	NUMPYEVAL	97.7%
			PANDASEVAL	94.3%	TORCHDATAEVAL	98.1%		
	Q.I.8: Inconsistent prompt style	311	HUMANEVAL	88.6%	MXEVAL$_{\text{HUMANEVAL}}$	66.4%	MXEVAL$_{\text{MBPP}}$	1.4%
			NUMPYEVAL	3.4%	PANDASEVAL	2.3%	TORCHDATAEVAL	1.5%
Noise	**Q.I.9: Interrogation questions in the prompt**	152	MXEVAL$_{\text{MATHQA}}$	0.1%	NUMPYEVAL	44.3%	ODEX	0.4%
			PANDASEVAL	36.4%	TORCHDATAEVAL	29.3%		
	Q.I.10: URL or reference in the comment	18	MBPP	6.1%	MXEVAL$_{\text{MBPP}}$	0.3%		

(MATHQA [38], MBPP [34], and HUMANEVAL [10]) to offer support to other **12** languages besides Python. CODEREVAL is a benchmark created by mining Python and Java projects on GitHub and contains 230 prompts for each language.

The benchmarks did not have a variety of *use scenarios*; their prompts were mostly crafted from coding exercises. Among these benchmarks, CODEREVAL [8] and ODEX [33] cover problems from more diverse use cases; they had prompts that were based on GitHub repositories and StackOverflow questions which are more similar to real use cases.

In terms of *contextual dependency complexity*, the benchmarks were mostly *self-contained*, *slib-runnable*, and *plib-runnable*. This means the structure of the problem described in the prompt is simple, *i.e.*, it does not take context from different files under a project.

RQ1 Summary of Findings:
- **Python** is the most supported language. Only **2** (out of **9**) benchmarks supported other languages besides Python.
- Most benchmarks (**6** out of **9**) had prompts whose solution would mostly require **built-in classes**.

B. RQ2: Benchmark Quality Evaluation

From our open coding of benchmarks' prompts, we found **10** quality issues that can be classified into **3** main categories. Figure 1 shows the quality issue types we found and their counts, while Table III enumerates the quality issue types we found. Most issues were related to the prompt's *format* and *intent*; there were **2,413** (**68%**) prompts improperly formatted and **1,621** (**53%**) with issues that may affect the LLM's ability to understand the prompt. Only **659** (18%) of the analyzed prompts did not have any issues with them.

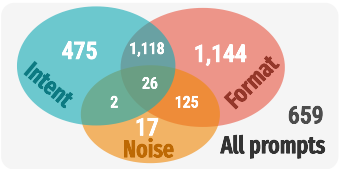

Fig. 1. Distribution of quality issues types

–***Intent-related issues***: This category refers to quality issues that can affect the LLM's ability to understand the intent (*i.e.*, purpose or goal) behind the prompt. We noticed that ***all*** benchmarks had at least one prompt with ***spelling and grammatical errors*** in them (**Q.I.2**). However, most of the prompts in these benchmarks were grammatically correct; only between **2.7%** to **20.7%** of them had spelling/grammatical errors. We also found that these benchmarks had prompts whose ***function/method's name does not match the intention described in the prompt*** (**Q.I.1**). It means the benchmark's developers used names that do not make the intended functionality clear. For example, all prompts in ODEX have this issue as the prompts' function name is in this format `f_Prompt_ID`. Similarly, all prompts' functions in MXEVAL$_{\text{MATHQA}}$ are named `problem`. We also found that two benchmarks had

prompts with ***partial/incomplete sentences*** (**Q.I.4**). Moreover, MXEVAL and HUMANEVAL have ***incorrect sample input-output pairs*** (**Q.I.5**); our analysis showed that ≈1% of their prompts are wrong. Prompts with incorrect examples of input/output pairs give inaccurate contextual information to the model. For instance, the HumanEval's prompt in Listing 1 should have `None` instead of an empty line (line 8).

Dataset: HumanEval [10]. **Prompt ID**: 12

```
1  from typing import List, Optional
2  def longest(strings: List[str]) -> Optional[str]:
3    """
4    Out of list of strings, return the longest one. Return the first one in case of
5    multiple strings of the same length.
6    Return None in case the input list is empty.
7    >>> longest([])
8
9    >>> longest(['a', 'b', 'c'])
10   'a'
11   >>> longest(['a', 'bb', 'ccc'])
12   'ccc'
13   """
```

Listing 1: Example of an incorrect input-output pair.

–***Format-related issues***: This category refers to problems related to how prompts are formatted. We found that **7** benchmarks ***did not properly use Javadocs/docstrings to express the function/method's intent*** (**Q.I.6**). This was especially pervasive on the MXEVAL benchmark; over **81%** of its prompts did not use proper Javadocs/docstrings. Moreover, **439** prompts were ***using single/multi-line comments to describe the intended behavior instead of using docstrings or Javadocs*** (**Q.I.7**). We also found ***inconsistent formatting in the benchmarks***, *i.e.*, style inconsistencies in them (**Q.I.8**). For example, we observed Python benchmarks in which some prompts included type annotations, but others did not.

–***Noise-related issues***: This category refer to cases where prompts contain unnecessary tokens (*noise*). We found **152** (**4%**) prompts with ***confusion questions***, *e.g.*, *"Is there a nice Pythonic way to do this?"* (**Q.I.9**). Another noise-related issue found was ***URLs in the prompt*** (**Q.I.10**), which do not carry meaningful information for the model.

RQ2 Summary of Findings:
- **2907** (**82%**) of studied prompts had at least one quality issue in them.
- ***Javadoc/docstring formatting issues***, ***function/method's name mismatching its intent***, and ***spelling/grammatical errors***, were the three most common quality issues.

C. RQ3: Impact on Performance

We ran five LLMs with the *original* prompt and *fixed* prompts. To better understand how each quality issue may affect an LLM's performance, we created prompts that fixed *one* issue at a time and prompts that fixed *all* issues. The **green** cells in Tables IV & V highlight the case in which the `pass@k` of the *fixed* prompt was higher than the *original* prompt.

We found that after fixing spelling and grammatical issues (**Q.I.2**), the CodeGen, and WizardCoder models, on average, performed better than the original Java prompts. Prompts with a correct JavaDoc and Docstring style (**Q.I.6**) tended to perform better than compared to the original prompts. Creating a consistent prompting style across the dataset is better for Python prompts from the GPT-3.5-Turbo model.

TABLE IV
PASS@K COMPARISON (ORIGINAL *vs.* FIXED PROMPTS – JAVA)

Model	Fixed Issue	Total	Modified prompt with fixes			Original prompt		
			pass@1	pass@3	pass@10	pass@1	pass@3	pass@10
CodeGen	Q.I.1	6	0.000	0.000	0.000	0.100	0.302	0.639
CodeGen	Q.I.2	22	0.155	**0.488**	**0.795**	0.164	0.483	0.789
CodeGen	Q.I.3	1	0.000	0.000	0.000	0.000	0.000	0.000
CodeGen	Q.I.5	1	**0.600**	0.509	0.957	0.500	0.681	0.995
CodeGen	Q.I.6	161	0.157	0.444	0.768	0.219	0.494	0.807
CodeGen	Q.I.8	67	0.153	0.305	0.494	0.184	0.448	0.751
CodeGen	All	161	0.150	0.375	0.620	0.219	0.494	0.807
SantaCoder	Q.I.1	6	0.000	0.000	0.000	0.758	0.819	0.917
SantaCoder	Q.I.2	22	0.675	0.823	0.948	0.709	0.850	0.975
SantaCoder	Q.I.3	1	0.050	0.150	0.500	0.050	0.150	0.500
SantaCoder	Q.I.5	1	**0.200**	0.681	0.995	0.150	0.855	1.000
SantaCoder	Q.I.6	161	0.609	0.781	0.938	0.690	0.848	0.961
SantaCoder	Q.I.8	67	0.410	0.494	0.555	0.596	0.800	0.962
SantaCoder	All	161	0.532	0.646	0.738	0.690	0.848	0.961
StarCoder	Q.I.1	6	0.000	0.000	0.000	0.600	0.725	0.831
StarCoder	Q.I.2	22	**0.880**	0.853	0.941	0.866	0.901	0.972
StarCoder	Q.I.3	1	0.000	0.150	0.500	0.000	0.150	0.500
StarCoder	Q.I.5	1	0.800	0.926	1.000	0.900	0.982	1.000
StarCoder	Q.I.6	161	0.717	0.845	**0.947**	0.755	0.869	0.939
StarCoder	Q.I.8	67	0.432	0.508	0.562	0.742	0.847	0.958
StarCoder	All	161	0.574	0.684	0.744	0.755	0.869	0.939
WizardCoder	Q.I.1	6	0.000	0.000	0.000	0.758	0.741	0.826
WizardCoder	Q.I.2	22	**0.680**	**0.803**	**0.903**	0.659	0.787	0.893
WizardCoder	Q.I.3	1	0.000	0.000	0.000	0.000	0.000	0.000
WizardCoder	Q.I.5	1	0.950	**0.926**	**1.000**	0.950	0.601	0.984
WizardCoder	Q.I.6	161	0.641	0.798	0.898	0.683	0.827	0.919
WizardCoder	Q.I.8	67	0.403	0.494	0.530	0.617	0.761	0.905
WizardCoder	All	161	0.534	0.665	0.724	0.683	0.827	0.919
GPT-3.5	Q.I.1	6	0.000	0.000	0.000	0.875	0.976	1.000
GPT-3.5	Q.I.2	22	0.883	0.950	0.952	0.900	0.952	0.952
GPT-3.5	Q.I.3	1	0.000	0.000	0.000	0.000	0.000	0.000
GPT-3.5	Q.I.5	1	0.550	0.926	1.000	0.750	0.991	1.000
GPT-3.5	Q.I.6	161	0.860	0.941	0.956	0.876	0.945	0.961
GPT-3.5	Q.I.8	67	0.515	0.547	0.552	0.813	0.913	0.944
GPT-3.5	All	161	0.713	0.761	0.770	0.876	0.945	0.961

When fixing *incorrect input/output pair examples* (Q.I.5) we noticed that the `pass@1` improved for CodeGen and StarCoder. While we observed cases where fixing one (or all) issues in a prompt increased a model's `pass@k`, there was not a consistent trend across models and languages. Fixing all issues in a Python prompt increased the `pass@k` of SantaCoder, StarCoder, and GPT-3.5-Turbo models.

RQ3 Summary of Findings:
- Fixing spelling and grammatical issues and having the standard JavaDoc and Docstring style can perform similarly to original prompts.
- Fixing different quality issues in a single prompt can provide better performance for Python code generation.

TABLE V
PASS@K COMPARISON (ORIGINAL *vs.* FIXED PROMPTS – PYTHON)

Model	Fixed Issue	Total	Modified prompt with fixes			Original prompt		
			pass@1	pass@3	pass@10	pass@1	pass@3	pass@10
CodeGen	Q.I.2	27	0.093	0.172	0.341	0.104	0.190	0.363
CodeGen	Q.I.1	7	0.000	0.000	0.000	0.014	0.138	0.251
CodeGen	Q.I.3	1	0.000	0.000	0.000	0.000	0.000	0.000
CodeGen	Q.I.6	164	0.045	0.187	0.352	0.066	0.225	0.423
CodeGen	Q.I.8	147	0.057	0.121	0.228	0.058	0.121	0.237
CodeGen	All	164	0.039	0.155	0.297	0.066	0.209	0.394
SantaCoder	Q.I.2	27	0.000	0.000	0.000	0.000	0.006	0.019
SantaCoder	Q.I.1	7	0.000	0.000	0.000	0.000	0.000	0.000
SantaCoder	Q.I.3	1	0.000	0.000	0.000	0.000	0.000	0.000
SantaCoder	Q.I.6	164	0.002	**0.022**	**0.063**	0.002	0.016	0.045
SantaCoder	Q.I.8	147	0.002	**0.019**	**0.054**	0.002	0.011	0.030
SantaCoder	All	164	**0.005**	**0.033**	**0.081**	0.002	0.016	0.045
StarCoder	Q.I.2	27	0.002	0.054	**0.168**	0.013	0.058	0.164
StarCoder	Q.I.1	7	0.000	0.000	0.000	0.057	0.079	0.199
StarCoder	Q.I.3	1	0.000	0.000	0.000	0.000	0.000	0.000
StarCoder	Q.I.6	164	0.023	**0.100**	**0.232**	0.035	0.097	0.225
StarCoder	Q.I.8	147	**0.035**	**0.086**	0.199	0.029	0.085	0.204
StarCoder	All	164	0.025	**0.105**	**0.246**	0.035	0.097	0.225
WizardCoder	Q.I.2	27	**0.013**	0.011	0.037	0.006	0.038	0.121
WizardCoder	Q.I.1	7	0.000	0.000	0.000	0.007	0.043	0.143
WizardCoder	Q.I.3	1	0.000	0.000	0.000	0.000	0.000	0.000
WizardCoder	Q.I.6	164	0.057	0.095	0.187	0.085	0.124	0.236
WizardCoder	Q.I.8	147	0.063	0.095	0.172	0.085	0.124	0.236
WizardCoder	All	164	0.005	0.033	0.081	0.002	0.016	0.045
GPT-3.5	Q.I.2	27	**0.235**	**0.450**	0.638	0.219	0.422	0.639
GPT-3.5	Q.I.1	7	0.000	0.000	0.000	0.393	0.500	0.565
GPT-3.5	Q.I.3	1	0.000	0.000	0.000	0.000	0.000	0.000
GPT-3.5	Q.I.6	164	**0.274**	**0.458**	**0.639**	0.249	0.402	0.549
GPT-3.5	Q.I.8	147	**0.295**	**0.459**	**0.630**	0.275	0.441	0.590
GPT-3.5	All	164	**0.275**	**0.454**	**0.640**	0.249	0.402	0.549

D. RQ4: Test set Contamination

We ran the NiCad [53] tool to detect generated codes that are clones of the canonical solutions in the HumanEval dataset. We can not see any Type-1 and hardly see Type-2 clones in the SantaCoder, StarCoder, and WizardCoder after using this tool. These models use the Stack dataset [46], and we checked if the original HumanEval dataset from OpenAI in this dataset using a tool provided by them[1]. Our result and the tool confirm that ***there is no test set contamination issue for the HumanEval dataset for these models***. The CodeGen-2.5 models are also trained with the Stack dataset [46] and three other training sets. There are comparatively more Type-1 and Type-2 clones from the output of this model. This indicates that test set contamination issues are present in the CodeGen-2.5 model.

The result for GPT-3.5 has more Type-1 and Type-2 clones than other models. As it is a closed source model, we cannot directly verify its training set to check whether it includes the HumanEval benchmark. However, given the higher code clone incidence, this model may have HumanEval's solution in its training set, which can justify the high performance of this model. We can also see that modified prompts generate fewer or equal numbers of clones than the original prompts.

[1]https://huggingface.co/spaces/bigcode/in-the-stack

TABLE VI
RQ5 RESULTS FOR EACH LLMS AND TEMPERATURE (**T**).

		Original			Modified		
Model	**T**	**Type-1**	**Type-2**	**Type-3**	**Type-1**	**Type-2**	**Type-3**
CodeGen	0.2	1 (0.6%)	2 (1.2%)	10 (6.1%)	0 (0.0%)	2 (1.2%)	3 (1.8%)
CodeGen	0.6	1 (0.6%)	5 (3.0%)	21 (12.8%)	0 (0.0%)	4 (2.4%)	20 (12.20%)
SantaCoder	0.2	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
SantaCoder	0.8	0 (0.0%)	0 (0.0%)	1 (0.6%)	0 (0.0%)	0 (0.0%)	1 (0.6%)
StarCoder	0.2	0 (0.0%)	0 (0.0%)	1 (0.6%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
StarCoder	0.8	0 (0.0%)	1 (0.6%)	8 (4.9%)	2 (1.2%)	4 (2.4%)	10 (6.1%)
WizardCoder	0.2	0 (0.0%)	0 (0.0%)	1 (0.6%)	0 (0.0%)	0 (0.0%)	1 (0.6%)
WizarCoder	0.8	0 (0.0%)	1 (0.6%)	5 (3.0%)	0 (0.0%)	0 (0.0%)	4 (2.4%)
GPT-3.5	1.0	4 (2.4%)	11 (6.7%)	32 (19.5%)	3 (1.8%)	11 (6.7%)	42 (25.6%)

RQ4 Summary of Findings:
The dataset used for training the CODEGEN-2.5 and GPT-3.5 model has a data contamination issue with the HumanEval dataset.

E. RQ5: Quality Assessment of Developers' Prompts

We identified **11** quality issues from **198** conversations between developers and ChatGPT (*i.e.*, 54% of the conversations had *at least one* quality issue). Similar to RQ2, these issues are classified into three categories:

–***Intent-related issues****:* This category refers to the developers' intention to describe the task for the ChatGPT.

- **Unclear prompt description:** When the description is ambiguous, confusing, and not precise, it may lead to a very different output than the intended one. It can be caused by a lack of detailed information or user mistakes. For **57** conversations, we observed that ChatGPT did not understand the prompt description and provided an output that did not fulfill what the developer asked.
- **Spelling or grammatical error:** There were **53** developers' prompts which had a spelling or grammatical error.
- **Lack of enough context:** Many parameters and values, as well as the description of related classes and objects, might be necessary to generate complex code. Problematic prompts ask to generate complex code without specifying proper context. We found **51** conversations that missed important context to solve the task described in the prompt.
- **Very Short Sentence:** We found **3** conversations with very short sentences (less than 3 words in length).
- **Self Admitted Technical Debt (SATD)** We found **6** prompts that contained Self Admitted Technical Debt (SATD). These are comments written by developers to indicate buggy, incomplete, or suboptimal codes (*e.g.*, `TODO`) [54].

–***Formatting-related issues****:* This category pertains to the improper formatting done by the developers while describing a coding task to ChatGPT.

- **Not using standard JavaDoc or Docstring:** We checked if the prompts follow standard JavaDoc or Docstring for

Java or Python, respectively. Providing input in a standard format should help the code generation tool better utilize it. We found that developers did not use proper JavaDoc or Docstring for **43** conversations.

- **Messy code snippet:** Codes in the prompts can be messy, with no or incorrect indentation and spacing. There were **14** prompts where the included code snippets were messy.

–***Noise-related issues****:* The prompt from the developers can contain unnecessary portions that may not be helpful in expressing the context.

- **URL Link or reference:** The prompt may contain a URL Link or reference to an external source. As some versions of ChatGPT may not be able to browse the Webpage using the link, the link will not add any additional information to the prompt. There were **15** prompts from the developers, which included URLs.

Comparison with benchmarks' prompts*:* In RQ2, we analyzed quality issues on prompts within benchmarks, and in this RQ we perform the same analysis over developers' prompts to ChatGPT. In our analyses, we found that intent-related and formatting-related quality issues are not the same in both prompt types. The developers' prompts included SATD and very short sentences. Moreover, RQ2 results showed cases that the benchmark prompts had *incorrect* input/output pair examples, and the function name did not match the intent.

RQ5 Findings:
- 54% of the conversations from the developers with ChatGPT had at least one issue.
- Prompts from developers mostly lack enough context.

V. Discussion

Benchmarks lack diversity*:* Our RQ1 findings indicate that the benchmarks mainly focus on Python and have small code exercises. According to a recent survey with about 67,000 professional developers from Stack Overflow [39], the most popular language is JavaScript. At the same time, Typescript is close to Python, and Java, C#, and C++ are not that behind. This indicates that we need to expand the benchmark dataset beyond Python. It is also noticeable from our findings that the benchmarks are not that complex. At the same time, real-world software can have thousands of lines of code intra and inter-dependencies with local and public libraries [55]. Hence, the current benchmarks do not mimic real-world scenarios. That is also indicated by a recent study [56] on unit test generation using Code-LLMs. They perform well on the small samples from HumanEval [10] but have substandard performance on real-world open-source projects.

Guidelines for Quality Prompts*:* Shi *et al.* [57] showed that cleaning noisy data from code summarization benchmarks can improve the performance of code summarization models. In RQ3, we fixed the quality issues identified in the studied benchmarks' prompts. The model's performance increases a little according to the result of this research question. However, it can viewed as an updated version of the benchmark, which includes less noise, consistent, and more understandable by human prompts. In addition, the fourth result indicates that an updated version of the existing dataset can help solve code contamination issues [11]. From our result, we can suggest following: We can suggest the following guidelines from our results: (1) Diversify the benchmark's programming languages, domains, and complexity (RQ1); (2) Use proper format and naming convention while defining the code prompts (RQ2); (3) Provide sufficient context in the prompt (RQ2); (4) Before using a sample extracted from public repositories in an evaluation set, check if whether it has been used as part of training sets or not (RQ4). One way to decrease contamination likelihood is to extract code made available after the knowledge cut-off date of models; (5) Make developers aware of following a specific format while working with conversation style code generation model, as other developers can benefit from reading them (RQ5).

Data contamination and Possible Solution*:* The model can perform well if the evaluation dataset leaks in the training set. Large language models need a large amount of data to train to be generalized for different tasks [16]. It can take a lot of work to deduplicate and remove test samples from the training set. In addition to that, there can be indirect leakage. For example, the HumanEval dataset was initially released by OpenAI, but it can be re-uploaded in other projects. These projects can be included in the training set, and that can be hard to capture, and may lead to data contamination. One of the possible solutions is uploading the evaluation set in a binary format [58]. Benchmarks can be updated regularly and benchmark the state-of-the-art models. It can also be better not to release canonical solutions to the problems, though it can create an issue to extend and verify the result.

Implication for the Developers and Researchers*:* A code generation model's performance can affect the model's usage. Developers may prefer the model which has better performance than other ones. However, the data contamination issue indicates that the benchmark result can be rigged. Our work in RQ4 can be a way to check if the model has contamination issues. Another thing is that real-world software is complex, and from RQ1, we can see that the existing benchmarks are not robust. Hence, a model can perform better on the existing evaluation dataset but may not perform well in real-world software due to hallucinations [56]. Thus, we need more robust benchmarks to evaluate the code generation model. While creating the dataset, researchers should also consider prompts that are less noisy, human-understandable, and follow standard coding practices.

VI. Threats to validity

We manually analyzed around 3,900 prompts from developers and the benchmarks, which can introduce *internal* threats to validity. However, we performed a peer review of our analyses, and Cohen's kappa score indicates a substantial agreement between the raters. Moreover, an experienced author has resolved the disagreements.

An *external* threat to the validity is that we considered benchmarks from left to right code generation and used two versions of HumanEval datasets [10], [23] to answer RQ3 and RQ4. However, HumanEval datasets are the *most* used benchmarks for the code generation model, and the majority of the benchmarks for code generation are left-to-right [7]. As we used only HumanEval datasets in RQ3 and it contained only a subset of the found issues in RQ2, the results may not be generalized for other benchmarks and with different issues.

To detect testset contamination, we detect code clones using NiCad, which introduces *construct* validity threats. However, NiCad is one of the most used tools for clone detection for different languages. Another thing is that some problems in the benchmarks may have solutions that are inherently similar (*e.g.*, calculating the sum of numbers in an array) that can be solved in one way, leading to cloned solutions.

VII. Related Work

A. Empirical Study of Benchmarks

Shi *et al.* described a study on *code summarization* benchmarks [57] that characterized the data noises into 12 categories and ranked benchmarks based on the percentage of noisy data. They also developed a code-comment cleaning tool and showed that cleaning improves performance drastically up to 67.3%. Prior works have also focused on automatically translating existing datasets to other languages [59]. Cassano *et al.* [59] translated the HumanEval and MBPP datasets into 18 different languages and compared the effectiveness of these two datasets which showed that HumanEval is more useful than MBPP. Moreover, question-answering (QA), reasoning, and reading comprehension datasets were also evaluated based on their effectiveness [60]. Unlike these prior works, we studied the quality of prompts in ***code generation*** benchmarks.

B. Empirical Study of LLMs

Recently, the study of LLMs has gained substantial attention owing to their good performance in various applications. Chang *et al.* [61] presented a comprehensive survey on evaluating LLMs from three aspects: what to evaluate, where to evaluate, and how to evaluate. A total of 45 widely used benchmarks are selected for this study, each focusing on distinct aspects and evaluation criteria. After summarizing existing works on LLM evaluation, the authors conclude that no concrete evidence exists that one particular evaluation protocol or benchmark—albeit with distinct features and focuses—is the most beneficial and successful. They also summarized LLMs' success and failure cases in different tasks to reveal their intrinsic strengths and weaknesses.

Chen *et al.* [62] analyzed the effectiveness of ChatGPT to assess the quality of the generated text. After comparing three reference-free evaluation techniques, they deduced that the Explicit Score—which uses ChatGPT to produce a numerical score indicating text quality—is the most efficient technique out of the three exploited techniques. On the other hand, Wu *et al.* [63] evaluated the potential and constraints of different GPT-4 approaches for addressing increasingly difficult and demanding math problems. Similarly, an assessment of ChatGPT's performance on various benchmarks has been conducted by Laskar *et al.* [64]. They tested ChatGPT on 140 tasks and examined 255K responses produced in these datasets. Valerio *et al.* [65] discussed the future of AI-driven software development, specifically the requirement engineering for LLMs to understand the task. In our work, we focused on prompt quality for the code generation model and checked their influence on performance while fixing quality issues.

C. Memorization in LLMs

Carlini *et al.* [9] quantitatively measured the risk of memorization in generative sequence natural language models. A follow-up study showed that sensitive personal data can be easily extracted by simple attacks on a language model like GPT-2 [26]. Moreover, larger models are much more vulnerable to such attacks. The model's capacity, the number of duplications of an example, and the number of tokens of context used to prompt the model demonstrate a log-linear relationship with the degree of memorization of the model [12]. Though it may show unique memorization behaviors, memorization during fine-tuning was not explored much. Shorter tasks, *e.g.*, sentiment analysis and extractive QA are less likely to be memorized; on the other hand, longer tasks, *e.g.*, summarization, increase the possibility of memorization [66].

Oren *et al.* [67] propose a *exchangeability* property-based statistical method of proving test set contamination in the form of benchmark memorization in language models. They conduct a series of tests comparing the log probability of the language model on the "canonical" ordering to the log probability on a dataset containing shuffled examples and flag contamination when statistically significant differences exist between the two log probabilities. Yang *et al.* [28] studied memorization issues in open-sourced pre-trained GPT2 models. They compared the model's output to the training set, while in our study, we focused on test set contamination (*i.e.*, models memorizing the answers to existing evaluation prompts) by including not only open-source but also closed-source models (*i.e.*, GPT-3.5). Ippolito *et al.* [68] contend that definitions of verbatim memorization are overly restrictive and overlook more nuanced types of memorization. They create MEMFREE, an effective defense against all verbatim memorization. They demonstrate how this seemingly perfect filter is insufficient to protect against training data leaks.

Unlike these prior works, we studied quality issues in ***code generation benchmarks*** and how they affect a model's performance.

VIII. Conclusion

Code generation models, used to aid developers in writing code faster, are evaluated using benchmarks. In our paper, we

studied these benchmarks and found that they are limited and have quality issues. Improving the quality of the benchmark can provide a better description of the prompt and may lead to a better performance of the model. In addition to that, data contamination issues can hinder the usefulness of the popular benchmark. In the future, we will explore the solution to automatically fix the prompts with quality issues, and solve data contamination issues.

REFERENCES

[1] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Comput. Surv.*, vol. 53, no. 3, jun 2020.

[2] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" *arXiv preprint arXiv:2211.03622*, 2022.

[3] I. Shani, "Survey reveals AI's impact on the developer experience | The GitHub Blog," *GitHub Blog*, Jun. 2023. [Online]. Available: https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/#methodology

[4] E. Kalliamvakou, "Research: quantifying github copilot's impact on developer productivity and happiness," 2023, [Online; accessed 10. Nov. 2023]. [Online]. Available: https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

[5] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proc. of the 6th ACM SIGPLAN Int'l Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: ACM, 2022, p. 21–29.

[6] N. A. Ernst and G. Bavota, "Ai-driven development is here: Should you worry?" *IEEE Software*, vol. 39, no. 2, p. 106–110, Mar 2022.

[7] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, Y. Wang, and J.-G. Lou, "When neural model meets NL2Code: A survey," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, 2023.

[8] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, T. Xie, and Q. Wang, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *International Conference on Software Engineering (ICSE)*, 2023.

[9] N. Carlini, C. Liu, Ú. Erlingsson, J. Kos, and D. Song, "The secret sharer: Evaluating and testing unintended memorization in neural networks," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 267–284.

[10] 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.

[11] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using LLMs in software engineering," in *ACM/IEEE 46th International Conference on Software Engineering - New Ideas and Emerging Results*. ACM/IEEE, Jan. 2024.

[12] N. Carlini, D. Ippolito, M. Jagielski, K. Lee, F. Tramer, and C. Zhang, "Quantifying memorization across neural language models," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=TatRHT_1cK

[13] "Chat completions," Accessed Mar 25, 2023, 2023. [Online]. Available: https://platform.openai.com/docs/guides/chat

[14] (2023, Jun.) Repository status - Anonymous GitHub. [Online; accessed 16. Nov. 2023]. [Online]. Available: https://anonymous.4open.science/status/Datasets-Quality-30C7

[15] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent Abilities of Large Language Models," *arXiv*, Jun. 2022.

[16] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[17] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *44th Intl. Conf. on Software Engineering (ICSE)*, 2022.

[18] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd Intl. Conf. on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162.

[19] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," in *2021 IEEE/ACM 18th Intl. Conf. on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 329–340.

[20] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[21] Y. Gao and C. Lyu, "M2ts: Multi-scale multi-modal approach based on transformer for source code summarization," in *Proc. of the 30th IEEE/ACM Intl. Conf. on Program Comprehension*, ser. ICPC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 24–35.

[22] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708.

[23] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang, S. K. Gonugondla, H. Ding, V. Kumar, N. Fulton, A. Farahani, S. Jain, R. Giaquinto, H. Qian, M. K. Ramanathan, R. Nallapati, B. Ray, P. Bhatia, S. Sengupta, D. Roth, and B. Xiang, "Multi-lingual evaluation of code generation models," in *The Eleventh International Conference on Learning Representations (ICLR)*, 2023.

[24] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[25] M. L. Siddiq, J. C. S. Santos, S. Devareddy, and A. Muller, "Generate and pray: Using sallms to evaluate the security of llm generated code," 2024.

[26] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson *et al.*, "Extracting training data from large language models," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2633–2650.

[27] Z. Zhou, J. Xiang, C. Chen, and S. Su, "Quantifying and analyzing entity-level memorization in large language models," *arXiv preprint arXiv:2308.15727*, 2023.

[28] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, "Unveiling memorization in code models," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 856–856.

[29] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[30] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: a critical review and guidelines," in *Proceedings of the 38th International conference on software engineering*, 2016, pp. 120–131.

[31] T. Xiao, C. Treude, H. Hata, and K. Matsumoto, "Devgpt: Studying developer-chatgpt conversations," in *Proceedings of the International Conference on Mining Software Repositories (MSR 2024)*, 2024.

[32] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models," *arXiv e-prints*, p. arXiv:2302.00288, Feb. 2023.

[33] Z. Wang, S. Zhou, D. Fried, and G. Neubig, "Execution-based evaluation for open-domain code generation," 2023.

[34] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[35] D. Zan, B. Chen, Z. Lin, B. Guan, Y. Wang, and J.-G. Lou, "When language model meets private library," 2022.

[36] D. Zan, B. Chen, D. Yang, Z. Lin, M. Kim, B. Guan, Y. Wang, W. Chen, and J.-G. Lou, "Cert: Continual pre-training on sketches for library-oriented code generation," 2022.

[37] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," 2021.

[38] A. Amini, S. Gabriel, S. Lin, R. Koncel-Kedziorski, Y. Choi, and H. Hajishirzi, "MathQA: Towards interpretable math word problem solving with operation-based formalisms," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 2357–2367. [Online]. Available: https://aclanthology.org/N19-1245

[39] S. Overflow, "Stack overflow devlopers survey," 2023. [Online]. Available: https://survey.stackoverflow.co/2023/#most-popular-technologies-language-prof

[40] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[41] "Code Generation on HumanEval," Nov. 2023, [Online; accessed 14. Nov. 2023]. [Online]. Available: https://paperswithcode.com/sota/code-generation-on-humaneval

[42] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint*, 2022.

[43] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, "The pile: An 800gb dataset of diverse text for language modeling," 2020.

[44] G. Inc, "Bigquery public datasets," 2022. [Online]. Available: https://cloud.google.com/bigquery/public-data

[45] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," *ICLR*, 2023.

[46] D. Kocetkov, R. Li, L. Ben Allal, J. Li, C. Mou, C. Muñoz Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The stack: 3 tb of permissively licensed source code," *Preprint*, 2022.

[47] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey *et al.*, "Santacoder: don't reach for the stars!" *arXiv preprint arXiv:2301.03988*, 2023.

[48] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[49] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," 2023.

[50] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[51] G. Inc., "Github copilot : Your ai pair programmer," 2022, [Online; accessed 10. Oct. 2022]. [Online]. Available: https://copilot.github.com

[52] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang, "Spoc: Search-based pseudocode to code," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[53] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 219–220.

[54] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 91–100.

[55] S. M. Meidani, "Towards an enhanced dependency graph," Master's thesis, University of Waterloo, 2022.

[56] M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, "Using large language models to generate junit tests: An empirical study," in *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*, 2024.

[57] L. Shi, F. Mu, X. Chen, S. Wang, J. Wang, Y. Yang, G. Li, X. Xia, and Q. Wang, "Are we building on the rock? on the importance of data preprocessing for code summarization," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 107–119.

[58] A. Jacovi, A. Caciularu, O. Goldman, and Y. Goldberg, "Stop uploading test data in plain text: Practical strategies for mitigating data contamination by evaluation benchmarks," *arXiv preprint arXiv:2305.10160*, 2023.

[59] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda, "Multipl-e: A scalable and polyglot approach to benchmarking neural code generation," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3675–3691, 2023.

[60] K. Zhou, Y. Zhu, Z. Chen, W. Chen, W. X. Zhao, X. Chen, Y. Lin, J.-R. Wen, and J. Han, "Don't make your llm an evaluation benchmark cheater," 2023.

[61] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie, "A survey on evaluation of large language models," 2023.

[62] Y. Chen, R. Wang, H. Jiang, S. Shi, and R. Xu, "Exploring the use of large language models for reference-free text quality evaluation: An empirical study," 2023.

[63] Y. Wu, F. Jia, S. Zhang, H. Li, E. Zhu, Y. Wang, Y. T. Lee, R. Peng, Q. Wu, and C. Wang, "An empirical study on challenging math problem solving with gpt-4," 2023.

[64] M. T. R. Laskar, M. S. Bari, M. Rahman, M. A. H. Bhuiyan, S. Joty, and J. X. Huang, "A systematic study and comprehensive evaluation of chatgpt on benchmark datasets," 2023.

[65] V. Terragni, P. Roop, and K. Blincoe, "The Future of Software Engineering in an AI-Driven World," in *Workshop 2030 Software Engineering co-located with FSE 2024*, 2024. [Online]. Available: https://arxiv.org/abs/2406.07737

[66] S. Zeng, Y. Li, J. Ren, Y. Liu, H. Xu, P. He, Y. Xing, S. Wang, J. Tang, and D. Yin, "Exploring memorization in fine-tuned language models," *arXiv preprint arXiv:2310.06714*, 2023.

[67] Y. Oren, N. Meister, N. Chatterji, F. Ladhak, and T. B. Hashimoto, "Proving test set contamination in black box language models," *arXiv preprint arXiv:2310.17623*, 2023.

[68] D. Ippolito, F. Tramèr, M. Nasr, C. Zhang, M. Jagielski, K. Lee, C. A. Choquette-Choo, and N. Carlini, "Preventing verbatim memorization in language models gives a false sense of privacy," *arXiv preprint arXiv:2210.17546*, 2022.