The Fault in Our Stars: Quality Assessment of Datasets for Code Generation Evaluation

Mohammed Latif Siddiq
msiddiq3@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Simantika Dristi and Joy Saha
{simantika.dristi,joy.saha}@bracu.ac.bd
BRAC University

Joanna C. S. Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, IN, USA

ABSTRACT

Large Language Models (e.g., GitHub Copilot, ChatGPT, *etc.*) are gaining popularity among software engineers' daily practices. A crucial aspect of developing effective code-generation LLMs is to evaluate these models using a robust dataset. Evaluation datasets 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 evaluation datasets used to benchmark different code generation models. To conduct this study, we manually analyzed 3,566 prompts from 9 evaluation datasets to identify quality issues in them. We also empirically studied the effect of fixing the identified quality issues in the dataset and their correlation with the model's performance. Finally, we investigated the memorization issues of the evaluation dataset, which can question the 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. They also suffered from quality issues like spelling and grammatical errors, unclear intentions, and not using proper documentation style. Fixing all these issues can lead to a better performance for Python code generation but not a significant improvement for Java code generation. We also found that GPT-3.5-Turbo and CodeGen-2.5 models possibly have data contamination issues.

CCS CONCEPTS

• **Software and its engineering → Software performance**; **Software usability**; *Empirical software validation.*

KEYWORDS

datasets, code generation, quality, data contamination

ACM Reference Format:

Mohammed Latif Siddiq, Simantika Dristi and Joy Saha, and Joanna C. S. Santos. 2024. The Fault in Our Stars: Quality Assessment of Datasets for Code Generation Evaluation. In *Proceedings of 21st International Conference on Mining Software Repositories (MSR 2024).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

With the recent release of GitHub Copilot [31] and ChatGPT [1], developers and students are increasingly relying on this code generation model assistants [50]. 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 [55]. Part of this fast widespread adoption is due to the increased productivity perceived by developers; AI helps them to automate repetitive tasks so that they can focus on higher-level challenging tasks [75]. As AI-driven Development Environments (AIDEs) become the mainstream approach to developing software [22], the need for reliable evaluation benchmarks is vital. Code generation benchmarks are crucial for evaluating and comparing the performance and effectiveness of various code generation techniques, frameworks, and tools. These benchmarks are designed to assess the generated codes from various perspectives, such as their precision and readability.

Code generation models generate code by taking prompts as inputs. These prompts capture the developers' intent and may contain natural language text or code elements, such as function declarations [42]. Code generation tools like GitHub Copilot [31] can improve productivity according to their internal research [37], and it also shows the developers can be 55% faster than the developers who do not use GitHub Copilot. The usage of and trust in these code-generation model-based tools depends on the backend models' performance [19], and they are evaluated with various benchmarks.

While there are more than 30 evaluation benchmarks for code generation [70], their *quality* and *reliability* are currently unclear. First, these datasets are often collected in an ad-hoc fashion, which may not be representative of real software scenarios [67]. Second, as these datasets are curated from publicly available data, there is the risk that existing models include data from these benchmarks in the training set (*i.e., testset contamination*). In this case, the reliability of `pass@k` [16] and other performance metrics is put into question, as the models might have memorized the solutions to the prompts in the dataset. Therefore, issues on these benchmark datasets significantly impact the trustworthiness of the evaluation results, making it crucial to investigate and evaluate the benchmarks themselves thoroughly.

In light of this research gap, in this paper, we present a systematic empirical investigation of code generation benchmarks. We evaluate the quality of prompts in these datasets from multiple dimensions (intent, format, and noise). We also investigate to what extent these quality issues affect a model's performance evaluation. Moreover, we also studied the possibility of test-set contamination issues. In our study, we reviewed the current benchmarks from the target programming languages, covered domain, and runnable level. We analyzed 3,500 prompts to check the quality issues in the prompts. We created 840 new Python and Java prompts by fixing quality issues to re-evaluate the code generation models. Finally, we leveraged code clone techniques to check for test-set contamination issues.

The contributions of this paper are:

- A thorough investigation of code generation benchmarks (RQ1 and RQ2) so researchers and developers can make informed decisions about choosing code generation techniques and tools based on their performance.
- A study of how quality issues in a prompt can affect a model's evaluation (RQ3).
- An investigation of possible test-set contamination issues (RQ4).

The **replication package** is publicly available in [6].

2 BACKGROUND

This section introduces core concepts to understand this paper.

2.1 Large Language Models

Large Language Models (LLMs) [63] are neural networks with tens of millions to billions of parameters that were trained on large amounts of unlabeled text using self-supervised learning or semi-supervised learning [10]. LLMs are intended to be general-purpose models for several natural language processing tasks, such as text generation, translation, summarization, *etc.* Examples of well-known LLMs are BERT (*Bidirectional Encoder Representations from Transformers*) [21], T5 (*Text-to-Text Transformer*) [52] and GPT-3 (*Generative Pre-trained Transformer*) [10].

While LLMs are trained to understand *natural* language, they can be fine-tuned with source code datasets to understand *programming* languages. This makes the LLMs to be used for a myriad of software engineering tasks, such as code completion [33, 38, 59], search [23], and summarization [28]. CodeBERT [23], CodeT5 [61], and Codex [17] are examples of "code LLMs", *i.e.*, LLMs trained on source code. Given a *prompt* as input, a code LLM generates new code tokens, one by one, until it reaches a stop sequence (*i.e.*, a pre-configured sequence of tokens) 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 declarations, expressions, comments, *etc.*

Transformer-based code generation methods employ masked language modeling objectives or left-to-right (causal) autoregressive language modeling objectives [10, 17]. That is, to generate source 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***. Code infilling aims to help the scenario in which developers edit an existing code, which means the edit action may depend on both sides of the cursor.

2.2 Benchmark datasets

Benchmark datasets are used to evaluate and compare models based on different metrics. From the perspective of code generation, current benchmark datasets [8, 17] are used to compare models based on their effectiveness in generating functionally correct code. Before that, generated codes are compared with a ground truth code based on their syntactical and data flow matching [29].

Code generation benchmarks usually contain a developer's intention, possibly captured in a natural language, comment, or combination of comment and code, referred to as a **prompt** [42]. After using the prompt for code generation, different metrics can be used to evaluate the performance. For example, `CodeBleu` [53] can be used for syntactical correctness, and `pass@k` [17] can be used for functional correctness. There can be particular purpose benchmarks; for example, SALLM [57] can evaluate code generation models from the perspective of secure code generation and use `vulnerable@k` to compare the performance of the models.

2.3 Memorization

Memorization refers to a model's ability to preserve and generate an identical string from its training data [12, 13]. In this work, we used a similar definition as the one presented by Carlini *et al.* [13]. 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 memorized by the code generation tool, which prior work has referred to as verbatim memorization [73]. In light of this definition, to identify testset contamination, we verify whether the generated code is a clone of the solution available in the dataset. In our work, we search for *type-1*, *type-2*, and *type-3* code clones [54].

A **type 1** clone occurs when two code snippets have identical code fragments except for layout, comments, and whitespace differences. A **type 2** clone happens 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.

3 METHODOLOGY

In this study, we answer the following research questions:

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

Code generation models need rigorous evaluation and verification. However, existing benchmarks may not represent real-world scenarios and cover all programming languages. In this question, we compare the code generation benchmarks' domain, paradigm, and contextual complexity.

RQ2 ***What are the quality issues in evaluation datasets?***

We study quality issues in the prompts of code generation benchmarks across three different dimensions: the prompt's *intent*, *formatting*, and *noise*. We manually analyzed a total of **3,566** from **9** datasets.

RQ3 ***Does improving the quality of the evaluation dataset affect the evaluation result?***

Quality issues in the datasets' prompts can negatively impact the benchmarking result. In this question, we investigate whether improving the dataset's quality affects the benchmarking results of code generation models.

RQ4 ***Are there contamination issues in existing evaluation datasets?***

Since code generation models are fine-tuned on source code from open-source repositories, there is a possibility that evaluation datasets can be in the training set. If an evaluation dataset is in the training set, the code generation model can perform better for the set because they have memorized the answer. Hence, this contamination issue will affect the benchmarking process of the code generation model. In this research question, we explore the possibility of contamination issues in the existing evaluation dataset.

The next sections describe how we answer each of these RQs.

3.1 RQ1: Datasets Comparison

To verify the benchmarks' potential of representing real-world code generation scenarios, we analyzed each dataset in order to identify their **(i)** ***programming language(s)***, **(ii)** ***application domains***, **(iii)** ***number of prompts***, and **(iv)** ***contextual dependency complexity***. We classify the *contextual dependency complexity* of each benchmark dataset based on the categorization scheme described by Yu *et al.* [67]. The complexity can be one of the following:

- **self-contained** datasets are those whose solution to the prompt can be implemented using only built-in classes/modules that do not need to be imported (*e.g.*, the Python's and Java's String/str classes do not need to be imported to be used).
- **slib-runnable** refers to cases 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** are cases in which the prompts' solutions only use libraries that are publicly available on PyPi or Maven central.
- **class-runnable** are cases 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** occurs when the solution uses code *outside* in its class but that is still declared on the same file as the prompt.
- **project-runnable** The solution uses code elements declared in other source files in the dataset.

For each benchmark dataset, we created a dataset profile identifying the information above.

3.2 RQ2: Dataset Quality Evaluation

To study the quality issues in these datasets, we collected popular benchmark datasets (§ 3.2.1) and performed a systematic analysis of their prompts in order to identify quality issues (§ 3.2.2).

3.2.1 Benchmark Collection. We collected 28 evaluation datasets from a recent survey [70]. 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** benchmark datasets. Next, we randomly sample each dataset with a 99% confidence interval and a 5% margin of error. The total number of prompts that we analyzed per dataset is shown in Table 1. We analyzed a total of **3,566** prompts from **9** datasets. As the results in RQ1 will show (§ 4.1), datasets mostly supported Python and Java. Thus, we only analyzed Java and Python prompts.

Table 1: Number of analyzed prompts per dataset

Dataset	#Prompts	#Sampled Prompts	Dataset	#Prompts	#Sampled Prompts
MXEVAL [8]	6,031	2,037	**MBPP** [9]	426	261
MBPP	1,940	791	**TorchDataEval** [68]	302	270
HumanEval	325	262	**HumanEval** [16]	164	132
MathQA	3,766	984	**PandasEval** [69]	101	88
CoderEval [66]	460	342	**NumpyEval** [69]	101	88
ODEX [62]	439	265	**JigsawDataset** [35]	88	83

3.2.2 Quality Issues. For each sampled prompt, we focused on identifying quality issues related to ***intent***, ***formatting***, and ***noise***. To identify these issues, two of the authors manually inspected each prompt independently, as described below. After each author finished the analysis, a third author, who has over 2 years of professional programming experience, resolved the discrepancies through discussion and mediation. We calculated the Cohen's Kappa coefficient [44] to measure the inter-rater reliability, and it is **0.76**, which indicates substantial agreement [44]. This analysis took us over 600 person-hours.

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.
–**I1: Spelling and grammatical errors**: Prompts with proper grammar help make the developer's intent clear. Hence, we checked each prompt for spelling/grammar mistakes using the LanguageTool spellchecker [4]. This automated inspection is prone to false positives because source code can include variable identifiers (*e.g.*, `"isManager"`) which would be incorrectly flagged as a mistake by the spellchecker. Thus, we manually inspected each prompt to check if the issues that arose were spelling and grammatical errors and adjusted accordingly.
–**I2: Very short sentences:** Comments in a prompt should be elaborate enough to give as many details as needed to the model. We checked each prompt in the dataset to verify whether its comment was too short to be clearly understood by the model. We implemented a rule that automatically flagged prompts if their source code comment had less than 3 words after splitting the comment by blank spaces. To eliminate the chances of false positives, we read each comment to see if it has a meaning on its own, albeit short.

–I3: Partial/Incomplete sentence: As incomplete comments can lead to confusion, we read the prompt's comments to check if they were partially written or incomplete. An example where the comment does not have a complete meaning is shown in Listing 1. It does not make it clear what the function is supposed to return, nor does it explain the meaning of the variables `n0`, `n1`, and `n2`.

Dataset: MathQA [8]. **Prompt ID:** 94.

```python
def problem():
    """
    a train is 410 meter long is running at a speed of 45 km / hour . in what time will it
    pass a bridge of 140 meter length n0 = 410.0 n1 = 45.0 n2 = 140.0
    """
```

Listing 1: Example of a prompt with an incomplete sentence

–I4: Self-Admitted Technical Debt (SATD): SATD are comments (*e.g.*, `TODO`, `FIXME`, *etc.*) made by developers indicating the existence of unfinished or suboptimal code that needs further attention [26]. SATD in a prompt degrades the quality of a dataset since it only makes the input lengthier and more vague for a code generation model. To identify prompts with this issue, we used the full list of SATD patterns from a prior study [51] and flagged a prompt as problematic if the source code contains one of those hack patterns. Subsequently, we read the comments flagged as problematic to check whether the prompt contains SATD.

–I5: Function/method's name mismatches with its intent: The name of a function/method should be meaningful, precise, and readable. A function name that is either gibberish or does not reflect the programmer's true intent can make it harder for the model to understand the purpose of the code to be generated. The prompt from the HumanEval dataset [16] shown in Listing 2 is an example of a function whose name ("skjkasdkd") carries no semantic meaning and does not clarify the function's purpose.

Dataset: HumanEval [16]. **Prompt ID**: 94

```python
def skjkasdkd(lst):
    """You are given a list of integers.
    You need to find the largest prime value and return the sum of its digits.
...
    """
```

Listing 2: Example of gibberish function name

–I6: The prompt description is unclear: Unclear prompt descriptions can lead to generating a very different code than intended. If a description is unclear to a human, the machine may have even more difficulty understanding it correctly. Moreover, some prompts need more critical small details for the machine to understand, and the solution may depend on that. One can find the following example in Listing 3 hard to understand.

Dataset: TorchDataEval [68]. **Prompt ID**: 17

```python
import monkey as mk
import numpy as np
kf = mk.KnowledgeFrame({'a': [4, 1, 7, 3], 'b': [5, 2, 9, 6], 'c': [6, 3, 2, 8]})
# I would like to create new knowledgeframe out of the old one in a way that
# there will only be values that exceed the average value of the column.
# We can compare values and then add NaNs by indexing or `where`
# We want remove NaNs also in first rows add custom function with `sipna`
kf =
```

Listing 3: Example of a unclear intention.

–I7: Incorrect input/output pair example: Input-output pair example helps the code generation model to better understand what it will receive and what it will have to generate from it. Wrong inputs/output pairs may confuse the model and throw it into an ambivalent situation. Furthermore, wrong output examples can force the model to generate an incorrect code in order to match the incorrect output. For instance, the first example in Listing 4 should return `None` instead of empty.

Dataset: HumanEval [17]. **Prompt ID**: 12

```python
from typing import List, Optional
def longest(strings: List[str]) -> Optional[str]:
    """
    Out of list of strings, return the longest one. Return the first one in case of multiple
    strings of the same length. Return None in case the input list is empty.
    >>> longest([])

    >>> longest(['a', 'b', 'c'])
    'a'
    >>> longest(['a', 'bb', 'ccc'])
    'ccc'
    """
```

Listing 4: Example of an incorrect input-output pair.

Format-related issues. This category has three issues related to formatting the prompts in a dataset.

–I8: Not using JavaDoc (Java) or docstrings (Python) on the prompt: When writing code in Java or Python, the standard way to document functions/methods is to use Javadocs and docstrings, respectively. Thus, we inspected the datasets to check whether the prompts ***are not*** using javadocs/docstring to indicate the function/method's purpose.

–I9: JavaDoc/docstring has formatting issues: Python's docstrings and Java's JavaDocs are expected to follow a specific format. For example, the Javadoc for a method should be placed before its declaration and must include a description followed by `@param`, `@return`, `@throws`, and `@see` tags [3]. Similarly, Python's docstrings [5] must use triple double quotes (*i.e.*, `"""`) to delimit the start and beginning of the docstring, and these must be placed right *after* the function/method's declaration. We implemented a rule checker to verify whether the docstrings/JavaDocs are compliant with the language's guidelines [3, 5]. Specifically, we consider a JavaDoc/docstring to have a proper format if it (i) has a *non-empty* function/method description or return documentation (using the `@return` tag), ***and*** (ii) has all the method's parameters documented using the `@param` tag. For Python, it is important to highlight that although the PEP 287[1] recommends the reStructuredText (reST) format, it is common to see Python functions documented using different styles. In this work, we accept docstrings using one of the following styles: reST, Google's format[2], Numpydoc format[3], and Epytext[4]. To ensure our results were accurate, we manually checked each prompt automatically flagged as problematic.

–I10: Inconsistent prompt style: We inspected each prompt in a dataset to identify style inconsistencies in them. For example, datasets for Python may include type annotations for some prompts but not for all of them. To do so, we first created a dataset profile for each dataset, which documents the typical format of each prompt, *e.g.*, whether it includes Python's type annotations, which docstring formatting style is used (*e.g.*, reST, Google, *etc.*), whether it has input/output examples on the description, *etc.* Subsequently, we inspected each prompt in the dataset to verify whether it deviated from the expected format.

[1]https://www.python.org/dev/peps/pep-0287
[2]https://github.com/google/styleguide/blob/gh-pages/pyguide.md#38-comments-and-docstrings
[3]https://numpydoc.readthedocs.io/en/latest/
[4]http://epydoc.sourceforge.net/

Noise-related issues. Finally, we investigated quality issues related to prompts containing unnecessary tokens (noise).
–**I11: Automatically generated comments:** Modern IDEs can generate placeholder code to help programmers. Placeholder codes in a prompt have no value to the models, instead, they just unnecessarily increase the input length and can divert focus from the important information. Therefore, we searched for comments in the prompt containing the strings `"auto-generated"`, `"method stub"`, `"automatically generated"`, and `"autogenerated"`. We then manually inspected the search results to ensure accuracy.
–**I12: URL Link or Reference**: Prompts can contain URLs from a website, for example, StackOverflow, or a reference to a file that is part of the full code base, but the context is not part of the prompt. This type of external information does not add additional context to the prompts. For example, Listing 5 includes a link from the GeeksforGeeks website.

Dataset: MBPP [9]. **Prompt ID**: 420

```
def is_perfect_square(n):
    """
    Write a function to check whether the given number is a perfect square or not.
    https://www.geeksforgeeks.org/check-if-given-number-is-perfect-square-in-cpp/
    """
```

Listing 5: Example of having URL in the prompt.

–**I13: Interrogation- Questions in the comment**: Developers may put in a comment questions about the function's/variable's intention. These questions express the developer's confusion, but not the actual purpose of the function/variable. Listing 6 is an example of a question in a prompt.

Dataset: NumpyEval[69]. **Prompt ID**: 95

```
import numpy as np

a = np.array([1,2,3,4,5,6])
b = np.array([1,4,5])

# Is there a way to compare what elements in a exist in b?
# Return a array of booleans, True if elements in a exist in b, False otherwise
c =
```

Listing 6: Example of interrogation in prompt

–**I14: Comment includes PII**: Since datasets are curated from public examples, they may include Personal Identifiable Information (PII), *e.g.*, author name, email addresses, username *etc.* PIIs in a prompt do not give any contextual information to the model. Thus, we manually checked each prompt to verify whether it contained PIIs.

3.3 RQ3: Impact on Performance

In the previous research question, we identified the quality issues in the evaluation datasets. In this RQ, we fixed the issues to verify whether fixing them affects the performance of the models. As it would be time-consuming to fix over 3 thousand prompts, for this question, we fixed the issues identified for the HumanEval dataset Python and Java version [8, 17]. We chose this dataset because most of the code generation models are evaluated with this dataset. In fact, a popular leaderboard published on PapersWithCode.com website [2] shows at least 50 models are evaluated with this dataset.

To fix the issues in this dataset, first, two of the authors, who have done the quality analysis in RQ2, went through 164 prompts from HumanEval Python [17] and 161 prompts from HumanEval Java version [8]. They fixed the issues one by one and put in a single prompt for each problem. Finally, they fixed all the issues in a single prompt for each problem. This way, we have 663 Python and 501 prompts for the Java version of HumanEval, including the original prompts (*i.e.*, 164 Python and 161 Java prompts). There are 27 prompts fixed for issue I1, 1 prompt for I4, 7 prompts for ID I5, 1 prompt for I6, 164 prompts for I9, 147 prompts for I10, and 164 prompts after fixing all problems for the HumanEval Python. For Java, there are 22 prompts fixed for issue I1, 6 prompts for I5,1 prompt for each of ID P6 and P7, 161 prompts for I9, 67 prompts for I10, and 161 prompts after fixing all problems.

We used three models to generate code from these prompts:

- **CodeGen** [47] is an LLM for code generation trained on three large code datasets and has three variants: CodeGen-nl, CodeGen-multi, and CodeGen-mono. CodeGen-nl is trained with the *Pile* dataset [27] 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) [30]. The CodeGen-mono is built from CodeGen-multi and further trained with a dataset [47] of only Python code snippets. They also released another version called CodeGen-2.5 [46], which is trained on the StarCoder data from BigCode [39]. 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.
- **StarCoder** [43] 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 later one is further trained with Python codes, we used that for Python and the former one for Java code generation.
- The **Generative Pre-trained Model (GPT)** [10] is a family of transformer-based [60] 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 [1].

We chose these models because they are representative code generation models. GPT-3 is used in GitHub Copilot [31], a popular code generation user tool [55]. StarCoder and CodeGen-2.5 models are open-source top-performing models for code generation.

We generated 20 outputs with a maximum of 512 new tokens for each prompt. We used `tiktoken`[5] package to calculate the token size of the canonical solution of the HumanEval Python set. We found that the maximum token size is 240; on average, there are 54 tokens. Hence, we asked the model to generate 512 new tokens (*i.e.*, 10x of the average). We used the HuggingFace interface for the CodeGen and StarCoder models. For GPT-3.5-Turbo, we used the OpenAI API to generate the code. Then, we calculated the `pass@k` metric by running the test cases for each output.

Computing and contrasting the pass@k. Code generation models are commonly evaluated using the `pass@k` metric [17, 40]. This metric estimates the probability that *at least one* out of *k* generated samples are functionally correct. This metric is computed by generating *n*

[5]https://pypistats.org/packages/tiktoken

samples per prompt ($n \geq k$), counting the number of samples c that are functionally correct ($c \leq n$), and calculating the unbiased estimator from Kulal *et al.* [40]:

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

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 [10] for all pass@k, **0.2** for pass@1 and **0.8** for calculating the rest of the metric for StarCoder [43], and **0.2** for pass@1 and **0.6** for the other two values of k for CodeGen model [46]. We chose these temperatures as these were the ones that were reported in their papers. Once we obtained the pass@k, we made a pair-wise comparison between the *original* prompt and its *fixed* version. This way, for example, if there are 27 fixed prompts for issue I1, we compare the pass@k with the original 27 and the fixed 27 prompts.

3.4 RQ4: Contamination Issues

In this question, we checked the contamination issue of the widely used HumanEval dataset's Python version [17]. We used this dataset because it is the only version that has the canonical solution and is at risk of the contamination issue. To answer this question, we ran NiCad (Automated Detection of Near-Miss Intentional Clones), a state-of-the-art code clone detection tool [20], 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 canonical solutions and the generated code to answer this question. Then, we used the 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 fixes *all* issues in RQ3. As the prompts are modified, there is no way they can be a part of the training set though they are syntactically close to the original prompts. However, comparing the results with the original and the modified can provide us with more insight into the data contamination issue.

We configured NiCad [20] 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 keep the default maximum difference threshold to 30%. For the minimum line number of clones, we configured it 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 important to highlight that since NiCad can detect clones with at least 5 lines, in case the canonical solution had less than five lines, we kept the threshold to 5. The *maximum* line number of a clone is set to the default value, *i.e.*, 2,500.

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 include Type-1 and Type-2 and keep the result as it is. Similar to a prior work [65], 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, that means the model likely has memorized the solution.

4 RESULTS

This section describes our findings and answers our RQs.

4.1 RQ1: Datasets Comparison

Table 2 shows the characteristics of each studied dataset. In terms of supported languages, we found that **all** datasets included prompts to generate Python code, and only **2** out of **9** datasets included other languages besides Python. The MXEVAL [8] is a dataset that extends three other datasets (MathQA [7], MBPP [9], and HumanEval [17]) to offer support to other 12 languages besides Python. CoderEval is a dataset created by mining Python and Java projects on GitHub and contains 230 prompts for each language. The studied datasets had an average of **516** prompts per programming language.

Although programming languages can have multiple paradigms (*e.g.*, Python supports structured and object-oriented programming), the studied datasets mainly cover *structured*, *procedural*, and *declarative* paradigms. These datasets also did not have a variety of usage scenarios; the prompts in these datasets were mostly crafted from coding exercises. Among these datasets, CoderEval [67] and ODEX [62] 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 scenarios.

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

RQ1 Findings:

- **Python** is the most supported language. Only 2 datasets supported other languages besides Python.
- The studied datasets mainly cover the ***structured***, ***procedural***, and ***declarative*** paradigms.
- The studied datasets had prompts whose solution would mostly require **built-in classes**. Only **3** datasets had prompts that would require using public libraries to solve the problem.

4.2 RQ2: Quality Evaluation of Datasets

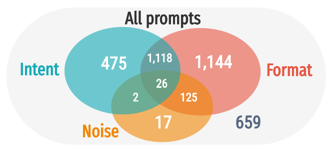

Figure 1: Distribution of quality issues types

Figure 1 shows the quality issue types we found and their counts, while Table 3 enumerates these issues. Most issues were related

Table 2: Evaluation Dataset Summary

Name	# Prompts	Target Code Language	Paradigm	Scenario	Contextual Dependency Complexity
CoderEval [66]	460	Python, Java	Structured	Pragmatic Code Generation	self-contained
HumanEval [16]	164	Python	Structured	Code Exercise	slib-runnable
JigsawDataset [35]	87	Python	Declarative	Public Library	
MBPP [9]	974	Python	Structured	Code Exercise	self-contained
MXEVAL [8]	16,171	C#, C++, Go, Java, JavaScript, Kotlin, Perl, PHP, Python, Ruby, Scala, Swift, TypeScript	Structured	Code/Math Exercises	slib-runnable, self-contained
MBPP	8,588	*All of the 13 languages listed above*	Structured	Code Exercises	slib-runnable, self-contained
HumanEval	1,934	*All except C++*	Structured	Code Exercises	slib-runnable
MathQA	5,649	*Only Python, Java, and JavaScript*	Structured	Math Exercises	slib-runnable, self-contained
NumpyEval [69]	101	Python	Procedural	Single Public Library	plib-runnable
ODEX [62]	945	Python	Structured	Open Domain	self-contained
PandasEval [69]	101	Python	Procedural	Single Public Library	plib-runnable
TorchDataEval [68]	50	Python	Structured	Private Library	plib-runnable

Table 3: Quality issues in each dataset (the percentages are the percent prompts with the issue in the dataset).

Type	Quality Issue	# Prompts	Dataset	%	Dataset	%	Dataset	%
Intent	**I5: 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%
	I1: 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%		
	I6: 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%
	I3: Partial or incomplete sentence	132	MXEVAL$_{\text{MathQA}}$	13.3%	ODEX	0.4%		
	I7: Incorrect input/output pair example	15	HumanEval	1.5%	MXEVAL$_{\text{HumanEval}}$	1.5%	MXEVAL$_{\text{MBPP}}$	1.1%
Formatting	**I9: 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%				
	I8: 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%		
	I10: 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	**I13: Interrogation questions in the prompt**	152	MXEVAL$_{\text{MathQA}}$	0.1%	NumpyEval	44.3%	ODEX	0.4%
			PandasEval	36.4%	TorchDataEval	29.3%		
	I12: URL or reference in the comment	18	MBPP	6.1%	MXEVAL$_{\text{MBPP}}$	0.3%		

to the prompt's *format* and *intent*; there were **2,585** prompts improperly formatted and **1,895** 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. We also did not find prompts with *very short sentences*, *self-admitted technical debt*, *automatically generated comments*, and *comments that include PIIs*.

Format-related issues. We found that **7** datasets did not properly use Javadocs/docstrings to express the function/method's intent (I9). This was especially pervasive on the MXEVAL dataset; 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 (I8). We also found inconsistency issues in the datasets (I10).

Intent-related issues. We noticed that ***all*** datasets had *spelling and grammatical errors* in them (I1). However, most of the prompts in these datasets did not have these issues; only between **2.7%** to **20.7%** of them had spelling/grammatical errors. We also found that these datasets had several prompts whose *function/method's name does not match the intention described in the prompt* (I5). It means the dataset developers used names that do not make the functionality, which is an example of the *"fallacious method name"* code smell [36]. MBXP and HumanEval have sample input-output pairs, and our analysis showed that ≈1% of them are wrong, which can give inaccurate contextual information to the model (I7).

Noise-related issues. We found **152** prompts with *confusion questions*, *e.g.*, "Is there a nice Pythonic way to do this?" (I13). Another noise-related issue found was *URLs in the prompt* (I12), which do not carry meaningful information for the model.

RQ2 Findings:
- Most of the observed quality issues were related to the prompts' *format* and *intent*.
- Most of the prompts do not have meaningful function/method names and follow standard JavaDoc or Docstring style to describe the intent.
- All datasets had prompts with spelling and grammatical errors.

4.3 RQ3: Performance Impact

We ran three models with the *original* prompt and *fixed* prompts. To better understand how each quality issue may affect a model's performance, we generated a prompt that fixed *one* issue at a time and another prompt that fixed *all* issues. The cells with a **green** color highlight the case in which the `pass@k` of the *fixed* prompt was higher than the *original* prompt.

Table 4 shows that after fixing spelling and grammatical issues (`I1`), the models performed similarly to the original prompts for both Java and Python. Prompts with a fixed JavaDoc and Docstring style (I9) 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. Fixing all issues in a single prompt does not decrease the performance for Java, though this approach performs better for the Python prompt of StarCoder and GPT-3.5-Turbo model.

RQ3 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.

4.4 RQ4: Testset Contamination

We ran the NiCad [20] tool to detect generated code 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 StarCoder after using this tool. This model uses the Stack dataset [39], and we checked if the original HumanEval dataset from OpenAI in this dataset using a tool provided by them[6]. Our result and the tool confirm that there is no data contamination issue for the HumanEval dataset for this model. The CodeGen-2.5 models are also trained with the Stack dataset [39] and three other datasets. There are comparatively more Type-1 and Type-2 clones from the output of this model. It indicates that there may be data contamination issues from these three datasets.

The result for GPT-3.5 has more Type-1 and Type-2 clones, and as it is a black box (closed source) model, it can not be verifiable if it includes the HumanEval dataset in its training set. However, our result indicates that it can have this evaluation 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.

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

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

5 DISCUSSION

Datasets lack diversity. Our finding in RQ1 indicates that the benchmark datasets are limited from different perspectives. They mainly focus on Python and are from small code exercises. According to a recent survey with about 67,000 professional developers from the popular question-answering site Stack Overflow [49], 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 datasets 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 [45]. Hence, the current benchmarks do not mimic the real-world scenarios. That is also indicated by a recent study [58] on unit test generation using Code-LLMs. They perform well on the small samples from HumanEval [17] but have substandard performance on the Evosuite benchmark [24], which consists of real-world open-source software.

Implication of the fixing prompts. Shi *et al.* [56] showed that cleaning noisy data from code summarization datasets can improve the performance of code summarization models. In RQ3, we fixed the quality issues identified in the studied datasets' 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.

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 [10]. 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, 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 [34]. Benchmark datasets can be updated regularly and benchmark the state-of-the-art models. It can be also 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

Table 4: Pass@k Results Comparison (original prompts *vs.* fixed prompts)

Model	Language	Fixed Issue (s)	Total	Modified prompt (with fixes)			Original prompt		
				pass@1	pass@3	pass@10	pass@1	pass@3	pass@10
CodeGen	Java	I1	22	0.155	**0.488**	**0.795**	0.164	0.483	0.789
CodeGen	Java	I5	6	0.000	0.000	0.000	0.100	0.302	0.639
CodeGen	Java	I6	1	0.000	0.000	0.000	0.000	0.000	0.000
CodeGen	Java	I7	1	**0.600**	0.509	0.957	0.500	0.681	0.995
CodeGen	Java	I9	161	0.157	0.444	0.768	0.219	0.494	0.807
CodeGen	Java	I10	67	0.153	0.305	0.494	0.184	0.448	0.751
CodeGen	Java	All	161	0.150	0.375	0.620	0.219	0.494	0.807
StarCoder	Java	I1	22	**0.880**	0.853	0.941	0.866	0.901	0.972
StarCoder	Java	I5	6	0.000	0.000	0.000	0.600	0.725	0.831
StarCoder	Java	I6	1	0.000	0.150	0.500	0.000	0.150	0.500
StarCoder	Java	I7	1	0.800	0.926	1.000	0.900	0.982	1.000
StarCoder	Java	I9	161	0.717	0.845	**0.947**	0.755	0.869	0.939
StarCoder	Java	I10	67	0.432	0.508	0.562	0.742	0.847	0.958
StarCoder	Java	All	161	0.574	0.684	0.744	0.755	0.869	0.939
GPT-3.5	Java	I1	22	0.883	0.950	0.952	0.900	0.952	0.952
GPT-3.5	Java	I5	6	0.000	0.000	0.000	0.875	0.976	1.000
GPT-3.5	Java	I6	1	0.000	0.000	0.000	0.000	0.000	0.000
GPT-3.5	Java	I7	1	0.550	0.926	1.000	0.750	0.991	1.000
GPT-3.5	Java	I9	161	0.860	0.941	0.956	0.876	0.945	0.961
GPT-3.5	Java	I10	67	0.515	0.547	0.552	0.813	0.913	0.944
GPT-3.5	Java	All	161	0.713	0.761	0.770	0.876	0.945	0.961
GPT-3.5	Python	I1	27	**0.235**	**0.450**	0.638	0.219	0.422	0.639
GPT-3.5	Python	I4	1	0.050	0.150	0.500	0.750	0.991	1.000
GPT-3.5	Python	I5	7	0.000	0.000	0.000	0.393	0.500	0.565
GPT-3.5	Python	I6	1	0.000	0.000	0.000	0.000	0.000	0.000
GPT-3.5	Python	I9	164	**0.274**	**0.458**	**0.639**	0.249	0.402	0.549
GPT-3.5	Python	I10	147	**0.295**	**0.459**	**0.630**	0.275	0.441	0.590
GPT-3.5	Python	All	164	**0.275**	**0.454**	**0.640**	0.249	0.402	0.549
StarCoder	Python	I1	27	0.002	0.054	**0.168**	0.013	0.058	0.164
StarCoder	Python	I4	1	0.000	0.000	0.000	0.000	0.000	0.000
StarCoder	Python	I5	7	0.000	0.000	0.000	0.057	0.079	0.199
StarCoder	Python	I6	1	0.000	0.000	0.000	0.000	0.000	0.000
StarCoder	Python	I9	164	0.023	**0.100**	**0.232**	0.035	0.097	0.225
StarCoder	Python	I10	147	**0.035**	**0.086**	0.199	0.029	0.085	0.204
StarCoder	Python	All	164	0.025	**0.105**	**0.246**	0.035	0.097	0.225
CodeGen	Python	I1	27	0.093	0.172	0.341	0.104	0.190	0.363
CodeGen	Python	I4	1	0.000	0.000	0.000	0.000	0.000	0.000
CodeGen	Python	I5	7	0.000	0.000	0.000	0.014	0.138	0.251
CodeGen	Python	I6	1	0.000	0.000	0.000	0.000	0.000	0.000
CodeGen	Python	I9	164	0.045	0.187	0.352	0.066	0.225	0.423
CodeGen	Python	I10	147	0.057	0.121	0.228	0.058	0.121	0.237
CodeGen	Python	All	164	0.039	0.155	0.297	0.066	0.209	0.394

Table 5: RQ4 results for each LLMs and temperature (T).

		Original			Modified		
Model	**Temp.**	**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%)
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%)
GPT-3.5	1.0	4 (2.4%)	11 (6.7%)	32 (19.5%)	3 (1.8%)	11 (6.7%)	42 (25.6%)

to check if the model has contamination issues. Another thing is that real-world software can be 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 [58]. Thus, we need more robust benchmarks to evaluate the code generation model. While creating the dataset, researchers should also consider having prompts with less noise, being human-understandable, and following standard coding practices.

6 THREATS TO VALIDITY

We manually analyzed around 3,500 prompts, which can introduce internal threats to validity. However, we performed a peer review of our analyses and the Cohen's kappa score indicates a substantial agreement between the raters. Moreover, an experienced author has resolved the disagreements. We used two versions of HumanEval datasets [8, 17] to answer RQ3, which can be an external threat to validity, but as mentioned before, they are the most used benchmark for the code generation model. We have also used NiCad clone detector [20], which is one of the most used tools for clone detection for different languages. We used three Transformer-based code generation models [60] as they have the best-performing result in recent times [43].

7 RELATED WORK

Empirical Study of Benchmarks. : Shi *et al.* described a study on *code summarization* benchmarks[56] 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 [14]. Cassano *et al.* [14] 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, similar to code generation datasets, question-answering (QA), reasoning, and reading comprehension datasets were also evaluated based on their effectiveness [72].

Empirical Study of LLMs. Recently, the study of LLMs has gained substantial attention owing to their good performance in a variety of applications, including both domain-specific and general natural language tasks. Chang *et al.* [15] presented a comprehensive survey on the evaluation of 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 with each benchmark focusing on distinct aspects and evaluation criteria. After summarizing existing works on LLM evaluation, the authors here conclude that there is no concrete evidence that one particular evaluation protocol or benchmark—albeit with distinct features and focuses—is the most beneficial and successful. They also summarized the success and failure cases of LLMs in different tasks to reveal the intrinsic strengths and weaknesses of LLMs.

There has been recent empirical research on using LLMs for unit test generation [58] where the authors investigated how useful three generative models (Codex, GPT-3.5-Turbo, and StarCoder) can be for unit test generation. They used three LLMs to generate unit tests for 194 classes from 47 open-source projects in the SF110 dataset [25] and 160 classes from the HumanEval dataset [17], and the performance of the LLMs were evaluated based on metrics like branch coverage, line coverage, correctness, and quality.

Chen *et al.* [18] analyzed the effectiveness of ChatGPT to assess the quality of the generated text. After comparing three different reference-free evaluation techniques, they deduced that the Explicit Score—which makes use of 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.* [64] evaluated the potential and constraints of different GPT-4 approaches for addressing increasingly difficult and demanding math problems.

An assessment of ChatGPT's performance on various benchmark datasets has been conducted [41]. They tested ChatGPT on 140 tasks and examined 255K responses that it produces in these datasets. Zhou *et al.* [74] presented an empirical investigation of the adversarial robustness of a large prompt-based language model of code, Codex. They further suggest techniques to overcome the challenges of adversarial attacks on natural-language inputs by enhancing the robustness of semantic parsers.

Memorization in LLMs. Carlini *et al.* [12] quantitatively measured the risk of memorization in generative sequence models. A follow-up study showed that sensitive personal data can be easily extracted by simple attacks on a language model like GPT-2 [13]. 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 [11]. 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 [71].

Oren *et al.* [48] 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.* [65] investigate memorization in large pre-trained code models empirically by using the concept of code clone to determine whether output from a code model contains memorization. Moreover, a positive correlation (a correlation coefficient of 0.804 in Spearman's rank correlation test) between the frequency of output in the training data and that in the generated outputs is shown in the study, suggesting that eliminating duplicates from the training data may be one strategy to lessen memorization. Ippolito *et al.* [32] 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 and conclude with the remarks that defining memorization needs further attention.

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

8 CONCLUSION

Code generation models, used to aid developers in writing code faster, are evaluated using benchmark datasets. In our paper, we studied these datasets 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] 2023. Chat completions. Accessed Mar 25, 2023. https://platform.openai.com/docs/guides/chat

[2] 2023. Code Generation on HumanEval. https://paperswithcode.com/sota/code-generation-on-humaneval [Online; accessed 14. Nov. 2023].

[3] 2023. How to Write Doc Comments for the Javadoc Tool. https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html [Online; accessed 15. Nov. 2023].

[4] 2023. *LanguageTool - Online Grammar, Style & Spell Checker.* https://languagetool.org [Online; accessed 16. Nov. 2023].

[5] 2023. PEP 257 – Docstring Conventions | peps.python.org. https://peps.python.org/pep-0257 [Online; accessed 15. Nov. 2023].

[6] 2023. *Repository status - Anonymous GitHub.* https://anonymous.4open.science/status/Datasets-Quality-30C7 [Online; accessed 16. Nov. 2023].

[7] Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. 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 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 2357–2367. https://doi.org/10.18653/v1/N19-1245

[8] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2023. Multi-lingual Evaluation of Code Generation Models. In *The Eleventh International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=Bo7eeXm6An8

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

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[11] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. 2023. Quantifying Memorization Across Neural Language Models. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=TatRHT_1cK

[12] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. 2019. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *28th USENIX Security Symposium (USENIX Security 19)*. 267–284.

[13] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. 2021. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*. 2633–2650.

[14] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3675–3691. https://doi.org/10.1109/TSE.2023.3267446

[15] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. 2023. A Survey on Evaluation of Large Language Models. arXiv:2307.03109 [cs.CL]

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

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

[18] Yi Chen, Rui Wang, Haiyun Jiang, Shuming Shi, and Ruifeng Xu. 2023. Exploring the Use of Large Language Models for Reference-Free Text Quality Evaluation: An Empirical Study. arXiv:2304.00723 [cs.CL]

[19] Ruijia Cheng, Ruotong Wang, Thomas Zimmermann, and Denae Ford. 2022. " It would work for me too": How Online Communities Shape Software Developers' Trust in AI-Powered Code Generation Tools. *arXiv preprint arXiv:2212.03491* (2022).

[20] James R. Cordy and Chanchal K. Roy. 2011. The NiCad Clone Detector. In *2011 IEEE 19th International Conference on Program Comprehension*. 219–220. https://doi.org/10.1109/ICPC.2011.26

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[22] Neil A. Ernst and Gabriele Bavota. 2022. AI-Driven Development Is Here: Should You Worry? *IEEE Software* 39, 2 (Mar 2022), 106–110. https://doi.org/10.1109/MS.2021.3133805

[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[24] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[25] Gordon Fraser and Andrea Arcuri. 2012. Sound Empirical Evidence in Software Testing. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE, 178–188.

[26] Gianmarco Fucci, Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2020. Who (Self) Admits Technical Debt?. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 672–676. https://doi.org/10.1109/ICSME46990.2020.00070

[27] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. arXiv:2101.00027 [cs.CL]

[28] Yuexiu Gao and Chen Lyu. 2022. 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* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/3524610.3527907

[29] Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. AixBench: A Code Generation Benchmark Dataset. arXiv:2206.13179 [cs.SE]

[30] Google Inc. 2022. BigQuery public datasets. https://cloud.google.com/bigquery/public-data

[31] GitHub Inc. 2022. GitHub Copilot : Your AI pair programmer. https://copilot.github.com [Online; accessed 10. Oct. 2022].

[32] Daphne Ippolito, Florian Tramèr, Milad Nasr, Chiyuan Zhang, Matthew Jagielski, Katherine Lee, Christopher A Choquette-Choo, and Nicholas Carlini. 2022. Preventing verbatim memorization in language models gives a false sense of privacy. *arXiv preprint arXiv:2210.17546* (2022).

[33] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-token Code Completion by Jointly Learning from Structure and Naming Sequences. In *44th Intl. Conf. on Software Engineering (ICSE)*.

[34] Alon Jacovi, Avi Caciularu, Omer Goldman, and Yoav Goldberg. 2023. Stop uploading test data in plain text: Practical strategies for mitigating data contamination by evaluation benchmarks. *arXiv preprint arXiv:2305.10160* (2023).

[35] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2021. Jigsaw: Large Language Models meet Program Synthesis. arXiv:2112.02969 [cs.SE]
[36] Marcel Jerzyk and Lech Madeyski. 2023. Code Smells: A Comprehensive Online Catalog and Taxonomy. In *Developments in Information and Knowledge Management Systems for Business Applications: Volume 7*. Springer, 543–576.
[37] Eirini Kalliamvakou. 2023. Research: quantifying GitHub Copilot's impact on developer productivity and happiness. https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/ [Online; accessed 10. Nov. 2023].
[38] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd Intl. Conf. on Software Engineering (ICSE)*. IEEE, 150–162.
[39] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *Preprint* (2022).
[40] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. 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.
[41] Md Tahmid Rahman Laskar, M Saiful Bari, Mizanur Rahman, Md Amran Hossen Bhuiyan, Shafiq Joty, and Jimmy Xiangji Huang. 2023. A Systematic Study and Comprehensive Evaluation of ChatGPT on Benchmark Datasets. arXiv:2305.18486 [cs.CL]
[42] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* 53, 3, Article 62 (jun 2020), 38 pages. https://doi.org/10.1145/3383458
[43] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
[44] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.
[45] Seyed Mehran Meidani. 2022. *Towards an Enhanced Dependency Graph*. Master's thesis. University of Waterloo.
[46] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *ICLR* (2023).
[47] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. *arXiv preprint* (2022).
[48] Yonatan Oren, Nicole Meister, Niladri Chatterji, Faisal Ladhak, and Tatsunori B Hashimoto. 2023. Proving Test Set Contamination in Black Box Language Models. *arXiv preprint arXiv:2310.17623* (2023).
[49] Stack Overflow. 2023. Stack Overflow Devlopers Survey. https://survey.stackoverflow.co/2023/#most-popular-technologies-language-prof
[50] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do Users Write More Insecure Code with AI Assistants? *arXiv preprint arXiv:2211.03622* (2022).
[51] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 91–100. https://doi.org/10.1109/ICSME.2014.31
[52] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html
[53] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).
[54] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495. https://doi.org/10.1016/j.scico.2009.02.007
[55] Inbal Shani. 2023. Survey reveals AI's impact on the developer experience | The GitHub Blog. *GitHub Blog* (June 2023). https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/#methodology
[56] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 107–119. https://doi.org/10.1145/3540250.3549145
[57] Mohammed Latif Siddiq and Joanna Santos. 2023. Generate and Pray: Using SALLMS to Evaluate the Security of LLM Generated Code. *arXiv preprint arXiv:2311.00889* (2023).
[58] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. An Empirical Study of Using Large Language Models for Unit Test Generation. arXiv:2305.00418 [cs.SE]
[59] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th Intl. Conf. on Mining Software Repositories (MSR)*. IEEE, 329–340.
[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.
[61] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. 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*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685
[62] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023. Execution-Based Evaluation for Open-Domain Code Generation. arXiv:2212.10481 [cs.SE]
[63] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. *arXiv* (June 2022). https://doi.org/10.48550/arXiv.2206.07682 arXiv:2206.07682
[64] Yiran Wu, Feiran Jia, Shaokun Zhang, Hangyu Li, Erkang Zhu, Yue Wang, Yin Tat Lee, Richard Peng, Qingyun Wu, and Chi Wang. 2023. An Empirical Study on Challenging Math Problem Solving with GPT-4. arXiv:2306.01337 [cs.CL]
[65] Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, DongGyun Han, and David Lo. 2023. What Do Code Models Memorize? An Empirical Study on Large Language Models of Code. arXiv:2308.09932 [cs.SE]
[66] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2023. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. *arXiv e-prints*, Article arXiv:2302.00288 (Feb. 2023), arXiv:2302.00288 pages. https://doi.org/10.48550/arXiv.2302.00288 arXiv:2302.00288 [cs.SE]
[67] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3597503.3623322
[68] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. When Language Model Meets Private Library. arXiv:2210.17236 [cs.PL]
[69] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual Pre-Training on Sketches for Library-Oriented Code Generation. arXiv:2206.06888 [cs.SE]
[70] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. When Neural Model Meets NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*.
[71] Shenglai Zeng, Yaxin Li, Jie Ren, Yiding Liu, Han Xu, Pengfei He, Yue Xing, Shuaiqiang Wang, Jiliang Tang, and Dawei Yin. 2023. Exploring Memorization in Fine-tuned Language Models. *arXiv preprint arXiv:2310.06714* (2023).
[72] Kun Zhou, Yutao Zhu, Zhipeng Chen, Wentong Chen, Wayne Xin Zhao, Xu Chen, Yankai Lin, Ji-Rong Wen, and Jiawei Han. 2023. Don't Make Your LLM an Evaluation Benchmark Cheater. arXiv:2311.01964 [cs.CL]
[73] Zhenhong Zhou, Jiuyang Xiang, Chaomeng Chen, and Sen Su. 2023. Quantifying and Analyzing Entity-level Memorization in Large Language Models. *arXiv preprint arXiv:2308.15727* (2023).
[74] Terry Yue Zhuo, Zhuang Li, Yujin Huang, Fatemeh Shiri, Weiqing Wang, Gholamreza Haffari, and Yuan-Fang Li. 2023. On Robustness of Prompt-based Semantic Parsing with Large Pre-trained Language Model: An Empirical Study on Codex. arXiv:2301.12868 [cs.CL]
[75] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity

Assessment of Neural Code Completion. In *Proc. of the 6th ACM SIGPLAN Int'l Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. ACM, New York, NY, USA, 21–29. https://doi.org/10.1145/3520312.3534864