An Empirical Study of Code Generation Errors made by Large Language Models

Da Song*
University of Alberta
Edmonton, AB, Canada
dsong4@ualberta.ca

Zijie Zhou*
University of Illinois
Urbana-Champaign
Urbana, IL, USA
zijiez4@illinois.edu

Zhijie Wang
University of Alberta
Edmonton, AB, Canada
zhijie.wang@ualberta.ca

Yuheng Huang
University of Alberta
Edmonton, AB, Canada
yuheng18@ualberta.ca

Shengmai Chen
Purdue University
West Lafayette, IN, USA
chen3301@purdue.edu

Bonan Kou
Purdue University
West Lafayette, IN, USA
koub@purdue.edu

Lei Ma
University of Alberta, Canada
The University of Tokyo, Japan
ma.lei@acm.org

Tianyi Zhang
Purdue University
West Lafayette, IN, USA
tianyi@purdue.edu

ABSTRACT

The emergence of Large Language Models (LLMs) has revolutionized automatic code generation from natural language input. Despite the promising performance, there remains a limited understanding of the code generation errors that LLMs can produce. To bridge the gap, this study provides an in-depth analysis of code generation errors across three representative LLMs within the HumanEval dataset. Specifically, we employ open-coding and iterative refinement to distill a comprehensive taxonomy of code generation errors intrinsic to LLMs. Based on this taxonomy, we identified two predominant categories of errors: semantic errors, indicating logical misunderstandings of the natural language input, and syntactic errors, uncovering structural misconceptions within the code. Additionally, we observed a consistent distribution of different error types across three models despite the differing successful rates. Our findings reveal the challenges that current code generation LLMs encounter, shedding light on future research about error-handling and repair techniques for LLMs' code generation.

CCS CONCEPTS

• **Software and its engineering**; • **Computing methodologies** → **Natural language processing**;

KEYWORDS

Empirical Study, Code Generation, Large Language Models

*Both authors contributed equally to this work.

MAPS '23, December 3, 2023, San Francisco, CA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Da Song, Zijie Zhou, Zhijie Wang, Yuheng Huang, Shengmai Chen, Bonan Kou, Lei Ma, and Tianyi Zhang. 2023. An Empirical Study of Code Generation Errors made by Large Language Models. In *Proceedings of the 7th Annual Symposium on Machine Programming (MAPS '23).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Automatic code generation from natural language input has consistently been popular across multiple communities, e.g., artificial intelligence (AI), software engineering (SE), and programming languages (PL). Recent advancements in Large Language Models (LLMs) have propelled this field forward significantly [2, 13, 33, 42, 43]. According to a benchmark built upon the HumanEval dataset, OpenAI's CodeX (released in 2021) can successfully solve 28.8% of a given set of 164 hand-written programming problems [12]. In a more recent study, it was demonstrated that GPT-4, one of the latest state-of-the-art (SOTA) LLMs released by OpenAI in 2023, achieved an even higher success rate, solving 88.4% of the programming problems in the HumanEval dataset [25].

Despite the significant progress and promising performance of State-of-the-Art (SOTA) LLMs, there remains a lack of deep understanding regarding cases where code generation fails. Particularly, developers intending to integrate LLMs into their daily development process may harbor the following concerns: *What types of errors does an LLM typically produce? Do various LLMs tend to make similar errors, or do they exhibit widely divergent patterns of mistakes?* Answering these questions would help researchers and developers gain insights from SOTA code generation LLMs, revealing future opportunities for developing error-handling and repair mechanisms for LLMs.

To bridge these gaps, in this paper, we conducted a comprehensive analysis of code generation errors made by the SOTA LLMs. Our study focused on three representative LLMs: CodeGen-16B [31], InCoder-1.3B [20], and ChatGPT [2]. We first collect a subset of failure code generation cases of different LLMs on the

HumanEval dataset. Subsequently, four authors were involved in multiple rounds of open-coding and iterative refinement to derive a taxonomy of code generation errors made by LLMs. Building upon this taxonomy, we label the errors made by all three LLMs to investigate the following three research questions:

- **RQ1:** To what extent the code generated by LLMs is similar to the ground truth code?
- **RQ2:** What specific types of errors manifest in code generation across different LLMs?
- **RQ3:** What commonalities and distinctions emerge from the errors produced by different LLMs?

Our study reveals server insights into the code generation errors made by LLMs. First, we identified two main categories of errors that consistently exist across various models: semantic and syntactic errors.

- **Semantic errors** encompass various issues such as *Missing Condition*, *Wrong (Logical) Direction*, *Incorrect Condition*, and *Constant Value Error*, among others. These errors often stem from the model's misinterpretation of the natural language description, shedding light on the challenges LLMs face in comprehending logical constructs and conditional logic.
- **Syntactic errors** refer to mistakes in the structure or grammar of code committed by LLMs. Representative syntactic errors include *Incorrect Code Blocks*, *Incorrect Function Arguments*, *Incorrect Return Values*, etc. The presence of syntactic errors underscores the difficulties LLMs encounter in correctly structuring code.

Furthermore, we find that **despite variations in successful rates among the models, the distribution of these error types remained consistent**. Specifically, we observe that a large proportion of the errors stem from missing requirements from the natural language input (semantic error: "`Missing multiple statements`", and syntactic error: "`Incorrect code block`"). This indicates that it is still challenging for LLMs to generate correct code given complex task requirements.

In summary, this paper makes the following contributions:

- We established a taxonomy of error types for three prominent and state-of-the-art code generation LLMs through iterative and axial coding procedures.
- We analyzed the similarities and differences in errors made by different code generation models, highlighting the challenges faced by LLMs.
- We discussed implications and future opportunities of developing error-handling and repair techniques for code-generation LLMs.

2 RELATED WORK

2.1 LLM-based Code Generation

Transformer has revolutionized AI-enabled text processing tasks ever since its introduction [45]. Its immense potential across a spectrum of downstream NLP tasks [15–17] has drawn the attention of researchers to explore the possibility of utilizing it for code-related tasks. Early attempts such as CodeBERT [18] and CodeT5 [47] exploit encoder-decoder architecture as the core backbone, aiming to generate both syntactically and semantically correct code. These models, comprised of millions of parameters, are pre-trained on a combination of programming and natural language data. Despite promising, these models have yet to match the proficiency of human programmers.

Marking a significant milestone along the direction, OpenAI released Codex, a decoder-only architecture with up to 12 billion parameters [12] in 2021. This model now also plays a vital role in a well-known commercial tool, GitHub Copilot. The HumanEval dataset is also proposed in the same paper, and it has become the default benchmark for the follow-up work. Later in the year, Google Research presents an empirical study with another famous benchmark dataset MBPP [8]. This study further shows the underscored the efficacy of the decoder-only architecture. Since then, the majority of subsequent work has gravitated towards the decoder-only LLM architectures [4, 20, 24, 31, 55]. Codex's success has drawn both Tech Giants and research groups into the area. For example, Meta proposes Incoder [20], Amazon provides commercial service CodeWhisperer [1], BigCode project launches SantaCoder [4] and StarCoder [24], and Salesforce proposes CodeGen [31]. Notably, all these models are tailored for tasks related to code processing.

Recently, pre-trained large language models with billions of parameters on massive text data have given rise to what is termed as emergent abilities [48]. These models exhibit impressive performance on various downstream tasks and have become the game changer in related fields. Examples of such are Palm series [5, 13], LLAMA series [42, 43], ChatGPT [2] and GPT series [33]. Among them, ChatGPT and GPT-4, which have been fine-tuned using Reinforcement Learning from Human Feedback (RLHF) [14, 34], are perceived as early indicators of Artificial General Intelligence (AGI) [10]. They can generate code with functional accuracy comparable to human developers, significantly surpassing prior methods. To illustrate, when evaluated on the HumanEval and EvalPlus benchmarks for the pass@1 score, GPT-4 achieves a successful rate of 88.4% and 76.2%, respectively, while ChatGPT records 73.2% and 63.4% [25]. Such performance markedly exceed those of the current SOTA LLMs specified for code processing, StarCoder [24], which achieves a successful rate of 34.1% and 29.3%, respectively.

2.2 Quality of AI-generated Code

Evaluating the quality of AI-generated code can reveal the current approach's shortcomings and guide future improvement. While prior research has delved into various facets such as robustness [6, 27, 38, 56], security [7, 36, 39], and usability [3, 9, 22, 44], studies specifically examining the correctness of AI-generated code [12, 21, 23, 26, 30, 37, 51] are of particular relevance to our work.

Objective metrics offer a relatively direct approach to assessing the quality and correctness of generated code. For instance, when the corrected reference code is predetermined, the CodeBLEU metric [37] quantifies the similarity between the generated and reference code, indicating the generated code's quality. However, since one functionality can yield multiple valid implementations, this metric might only partially capture correctness. In contrast, the *Pass@k* metric [12, 23] evaluates how many generated samples successfully pass a specified test suite, offering a potentially more precise measure of code correctness. Several following attempts have been made to evaluate generated code quality more holistically. These include efforts to transform coding problems into

multilingual versions [11, 55] and initiatives that expand upon existing test cases [25]. Building on these efforts, a large number of studies has statistically analyzed the code quality of various LLMs, providing a high-level overview of their programming capabilities [21, 30, 41, 50–52]. However, since these studies often neglect direct source code analysis, they may fall short of offering a detailed insight into the errors committed by LLMs. Compared with these studies, we conduct systematic categorization of fine-grained error types with manual investigation.

Three studies are particularly pertinent to our work [26, 28, 35]. Liu et al. [26] comprehensively evaluates the quality of code generated by ChatGPT, considering various aspects such as compilation and run-time errors, output correctness, coding style, maintainability, and performance. Notably, their analysis predominantly relies on error messages and static analysis tools, such as Pylint and Flake8. In contrast, our methodology emphasizes systematic annotations to pinpoint the underlying causes of each error for the generated code. Pan et al. [35] introduce a taxonomy centered on code translation bugs, organizing 14 categories into four distinct groups. While their emphasis is on code translation, our taxonomy pertains to code generation, positioning our work as a parallel endeavor. Liu et al. [28] provide a thorough examination of the quality of code generated by ChatGPT in a multi-round setting. While their taxonomy primarily addresses compilation and run-time errors, our study extends to encompass functional correctness, considering both syntactic and semantic faults.

3 METHODOLOGY

In this section, we introduce the dataset and LLMs used in our study and the procedure of our manual analysis.

3.1 Dataset

In this study, we utilize the HumanEval dataset to analyze code generation errors made by LLMs. The HumanEval dataset encompasses 164 hand-written Python programming tasks, each accompanied by an average of 7.7 unit tests [12]. These tasks involve language comprehension, reasoning, algorithms, and simple mathematics. Notably, the inclusion of hand-writing programming tasks serves to mitigate certain potential biases that might arise when employing crowd-sourced datasets such as MBPP [8], considering that a significant fraction of code generation LLMs is trained on publicly available GitHub repositories.

A common practice to evaluate an LLM's performance on code generation is adopting the *Pass@k* metric [12]. *Pass@k* measures the successful rates over a code generation dataset for an LLM, where k refers to the number of code samples generated by the LLMs. A task is deemed successful if, among the k samples, any manages to successfully pass all given test cases. In this paper, our primary focus lies on code generation errors obtained with the *Pass@1* metric.

3.2 Code Generation LLMs

We focus on three representative code generation LLMs in this study: CodeGen-16B [31], InCoder-1.3B [20], and ChatGPT [2]. Table 1 presents each model's performance on the HumanEval dataset. We introduce each model in the following.

Table 1: Code generation LLMs included in this study

Model	Release	Performance		
		Pass@1	*Pass@10*	*Pass@100*
CodeGen-16B [31]	Mar. 2022	32.9%	56.0%	81.5%
InCoder-1.3B [20]	Apr. 2022	12.2%	15.9%	25.2%
ChatGPT [2]	Nov. 2022	73.2%	88.6%	94.0%

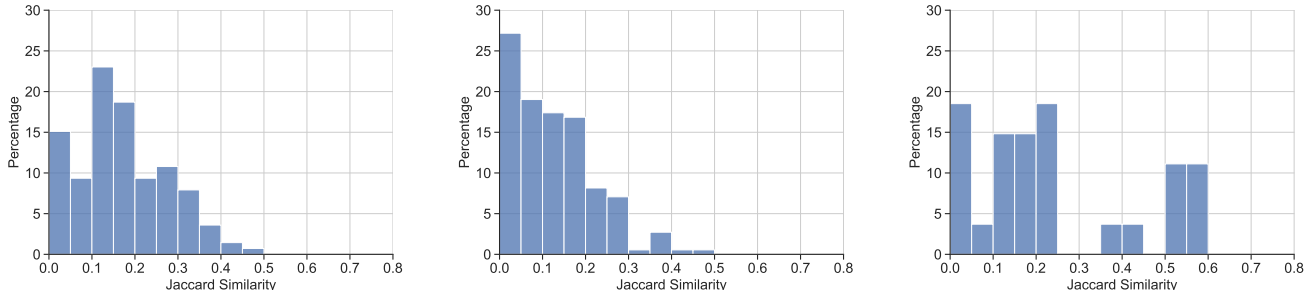
- **CodeGen-16B [31]** is an open-source LLM released by Salesforce. It employs a decoder-only architecture with rotary position embedding. The series of CodeGen models are trained on 217GB Python code. We utilize a version of CodeGen with 16B parameters (CodeGen-16B).
- **InCoder-1.3B [20]** is another open-source LLM released by Meta AI. InCoder utilizes a new causal masking objective, which allows filling code blocks as well as standard left-to-right code generation. InCoder is trained on 159GB open-source repositories with a permissive license licensed from GitHub, GitLab, and StackOverflow. We adopt an InCoder variant with 1.3b parameters (InCoder-1.3B).
- **ChatGPT [2]** is a fine-tuned version of GPT 3.5, released by OpenAI. It is further optimized for dialogue by using Reinforcement Learning with Human Feedback (RLHF). ChatGPT is trained with a massive number of crowd-sourced web text up to September 2021.

3.3 Manual Analysis Procedure

We followed the well-established open coding process to develop a taxonomy of code generation errors made by LLMs [40]. We detail this process as the following steps:

3.3.1 Open coding. To begin, we randomly collected 31 errors made by ChatGPT [2], 22 errors made by CodeGen-16B[31], and 88 (about 60 %) errors made by InCoder-1.3B [20] within the HumanEval dataset. To establish a comprehensive taxonomy of code generation errors, we further collected 10 errors each from two other code-oriented LLMs (StarCoder [24] and SantaCoder [4]) to enrich our labelling. A total of 161 code generation errors were collected for coding at this stage. Then, four authors with rich Python programming experience performed open coding on these errors. Given ground truth (i.e., correct code snippets) provided by the original HumanEval dataset, they were instructed to answer the following three questions during the process: (1) What are the syntactic differences between incorrect code and ground truth? (2) Where was the error made? (3) What is the root cause of the error? Upon the completion of the initial coding iteration, all coded samples were recorded in a document. All authors then convened to collectively discuss the codes and achieved consensus within this preliminary version of the codebook.

3.3.2 Iterative refinement of the codebook. After obtaining the preliminary codebook, four annotators iteratively improved the established codebook. Each iteration included the following steps. First, four annotators was assigned with a batch of 30 CodeGen-16B errors (including 8 out of 30 unlabeled errors). Subsequently, four annotators performed three rounds of coding independently. Within each round, they were tasked with justifying their respective



(a) CodeGen-16B **(b) InCoder-1.3B** **(c) ChatGPT**

Figure 1: Jaccard similarity between the generated code and ground truth across three LLMs.

codes for every error and recording their reasoning in a document. If a new error occurs that the codebook does not cover, the annotator needs to write a description of the error. Second, we computed Fleiss' Kappa [19] to determine the internal consistency among the annotators at the end of each round of labeling. After each Fleiss' Kappa calculation, the annotators were asked to conduct an internal meeting. The theme of the meeting is to discuss the differences between annotators' justification of specific coding and exchange opinions about updating the codebook. Finally, if the inter-rater reliability scores between the annotators are not appreciable after going through the above procedures, a new round of labeling will be held until the scores were substantial. At the end of the final refinement iteration, the annotators had completed a total of three iterations. The final codebook was established, and the Fleiss' Kappa of the syntactic errors and semantic errors were both stable above 0.68.

3.3.3 Annotating the rest of the code. Annotators use the final codebook to label the remaining dataset. In this process, the unlabled errors collected from each model was evenly assigned to four annotators. Four annotators then independently labeled their assigned errors. The final coding results were documented as a spreadsheet.

4 RESULTS

4.1 RQ1: Code Similarity

Figure 1 shows the distribution of Jaccard similarity scores between the incorrect generated code and the ground truth for three different LLMs. All three LLMs have low Jaccard similarity scores, where the highest similarity score across three models is lower than 0.6. The results indicate that non-trivial efforts might be needed to fix these errors. Compared with the results of ChatGPT (Figure 1c), we also find that two open-sourced LLMs, CodeGen-16B and InCoder-1.3B, have much lower median similarity scores (0.15 and 0.11 v.s 0.20). We also observe that InCoder-1.3B has much more code snippets with a low similarity score (i.e., Jaccard similarity score < 0.1). A plausible explanation is the generation of meaningless code snippets by InCoder-1.3B. For example, InCoder-1.3B kept generating similar but meaningless statements (Example 1) for a sorting task, resulting a extremely low Jaccard similarity score.

```
# Given an array of non-negative integers, sort it in ascending
# order if the sum(first index value, last index value) is odd,
# in descending order otherwise.

def sort_array(array):
    sorted_array = []
    sorted_array.append(array[0])
    sorted_array.append(array[1])
    sorted_array.append(array[2])
    ...
```

Example 1: An example of meaningless code snippets generated by InCoder-1.3B

Notably, there is a large performance gap between the open-sourced LLMs and ChatGPT, which potentially lead to a relatively lower Jaccard similarity. We further analyze the distribution of different code types with different LLMs in Section 4.3.

Finding 1

Most of the code generated by three LLMs is not similar to the ground truth, while InCoder-1.3B generated more code snippets with extreme low similarity.

4.2 RQ2: Error Taxonomy

Table 2 presents the finalized taxonomy of code generation errors made by LLMs. The taxonomy contains two main categories of errors alongside two different dimensions: *Semantic Errors* and *Syntactic Errors*. The *semantic errors* predominantly exposes misconceptions in comprehending the natural language descriptions accompanying programming tasks. By contrast, the *syntactic errors* could indicate potential grammatical or structural imprecision within the generated code. We adopt a three-level categorization for each error type. For instance, "`Semantic Errors`"(*category*)—"`Condition Error`" (*type*)—"`Missing condition`" (*sub-type*). Below we show a few example of errors committed by studied LLMs.

```
# Given an integer. return a tuple that has the number of even and
# odd digits respectively.

def even_odd_count(num):
    even_count = 0
    odd_count = 0
    for digit in str(abs(num)):
        if int(digit) % 2 == 0:
            even_count += 1
        else:
            odd_count += 1
    if num < 0:
```

Table 2: Taxonomy of code generation errors made by LLMs.

Error Categories	Error Types	
Semantic Errors	Condition Error	Missing condition Incorrect condition
	Constant Value Error	Constant value error
	Reference Error	Wrong method/variable Undefined name
	Operation/Calculation Error	Incorrect arithmetic operation Incorrect comparison operation
	Garbage Code	Only comments Meaningless code snippet Wrong (logical) direction
	Incomplete Code/Missing Statements	Missing one statement Missing multiple statements
	Memory Error	Infinite loop Integer overflow
Syntactic Errors	Conditional Error	If error
	Loop Error	For error While error
	Return Error	Incorrect return value
	Method Call Error	Incorrect function name Incorrect function arguments Incorrect method call target
	Assignment Error	Incorrect arithmetic Incorrect constant Incorrect variable name Incorrect comparison
	Import Error	Import error
	Code Block Error	Incorrect code block Missing code block

```
        even_count *= -1
        odd_count *= -1
    return (even_count, odd_count)
```

Example 2: An example of ChatGPT error

In this example (Example 2), ChatGPT adds an extra code block (highlighted in red) to deal with the cases when the input is a negative number, which is unnecessary. Therefore, from a semantic viewpoint, this example is categorized as "`Wrong (logical) direction`", while from a syntactic angle, it falls under "`Incorrect code block`".

Example 3 shows an example of errors made by CodeGen-16B. In this example, the model fails to understand the task requirements. Subsequently, the generated code is simply the copy&paste of the given examples. Therefore, we code this example as "`Meaningless code snippet`" and "`Incorrect code block`", respectively

```
# You are given a list of integers.
# You need to find the largest prime value and return the sum of
# its digits.

def skjkasdkd(lst):
    ...
    lst = [0,724,32,71,99,32,6,0,5,91,83,0,5,6]
    lst = [0,81,12,3,1,21]
    lst = [0,8,1,2,1,7]
    ...
```

Example 3: An example of CodeGen-16B error

Example 4 shows an example of InCoder-1.3B error. In this example, InCoder-1.3B miss the task requirements that 'y' is also a vowel, resulting in a semantic error of "`Missing one statement`" and a syntactic error of "`Missing code block`".

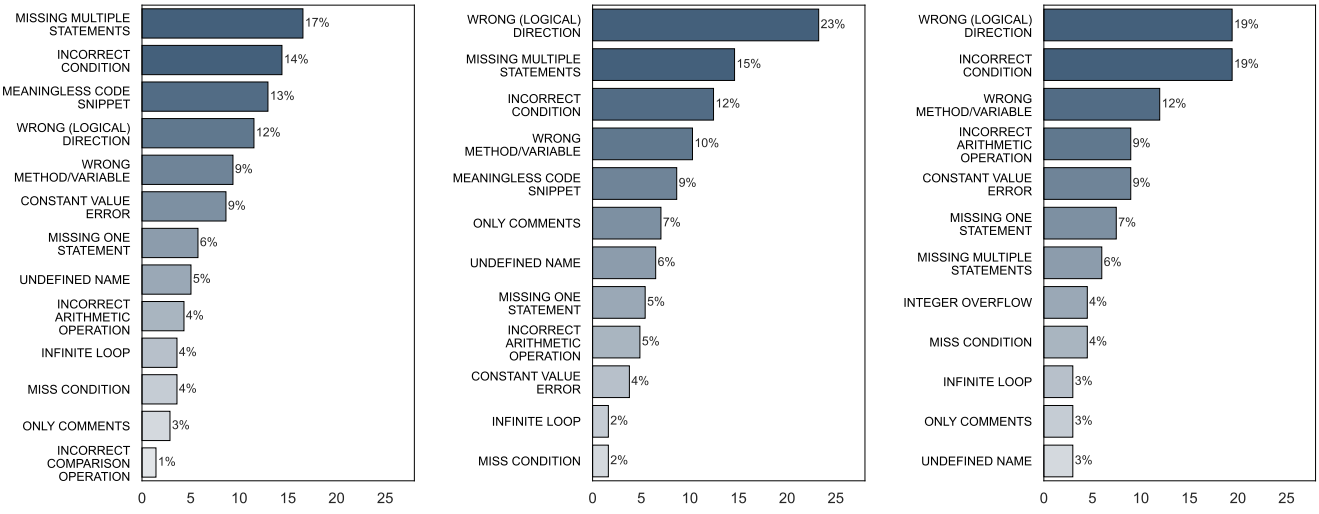
```
# Write a function vowels_count which takes a string representing
# a word as input and returns the number of vowels in the string.
# Vowels in this case are 'a', 'e', 'i', 'o', 'u'. Here, 'y' is
# also a vowel, but only when it is at the end of the given word.

def vowels_count(s):
    if sum(1 for c in s if c in 'AEIOUaeiou')
```

Example 4: An example of InCoder-1.3B error

Finding 2

We establish the first taxonomy of LLMs' code generation errors from two dimensions: *semantic* and *syntactic* errors. Each of them includes fourteen specific types of errors.



(a) CodeGen-16B **(b) InCoder-1.3B** **(c) ChatGPT**

Figure 2: Distribution of *Semantic Errors* made by three LLMs.

4.3 RQ3: Comparison between Different Models

Semantic Errors. Figure 2 shows the distribution of semantic errors of three studied LLMs: CodeGen-16B, InCoder-1.3B, and ChatGPT. Collectively, all three LLMs exhibit instances of all seven primary categories of semantic errors, encompassing a total of 13, 12, and 12 respective sub-type errors. Three studied LLMs also share a similar distribution regarding the error types. The most common semantic error type is "`Garbage Code`", consisting of "`Only comments`", "`Meaningless code snippet`", and "`Wrong (logical) direction`". 27.3%, 38.1%, and 22.4% of the errors made by three LLMs fall into this category, respectively. We also notice that InCoder-1.3B has a larger portion of "`Wrong (logical) direction`". We suspect this is due to the much smaller size of the model compared with the other two LLMs, resulting in insufficient capabilities of understanding complex task requirements. As shown in Example 5, InCoder-1.3B failed to understand a relatively complex task. As a result, InCoder-1.3B generated code in a wrong logical direction that only sorted the input.

```
# You are given a 2 dimensional data, as a nested lists, which is
# similar to matrix, however, unlike matrices, each row may
# contain a different number of columns. Given lst, and integer x,
# find integers x in the list, and return list of tuples,
# [(x1, y1), (x2, y2) ...] such that each tuple is a coordinate -
# (row, columns), starting with 0. Sort coordinates initially by
# rows in ascending order. Also, sort coordinates of the row by
# columns in descending order.

def get_row(lst, x):
    lst = sorted(lst, key=lambda x: x[0])
    lst = sorted(lst, key=lambda x: x[1])
    return [(i, j) for i, j in zip(lst, lst[1:])]
```

Example 5: An example of Wrong (logical) direction made by InCoder-1.3B

Another common error type is "`Incorrect condition`", where 14%, 12%, and 19% of the errors made by CodeGen-16B, InCoder-1.3B, and ChatGPT can be grouped into this category, respectively. Example 6 shows an incorrect condition generated by ChatGPT.

```
# Create a function that returns True if the last character of a
# given string is an alphabetical character and is not a part of
# a word, and False otherwise. Note: "word" is a group of
# characters separated by space.

# Incorrect code by ChatGPT
def check_if_last_char_is_a_letter(txt):
    if len(txt) == 0:
        return False
    else:
        last_char = txt[-1]
        if last_char.isalpha() and (len(txt.split()[-1]) == 1) :
            return True
        else:
            return False

# Groundtruth
def check_if_last_char_is_a_letter(txt):
    if len(txt) == 0: return False
    if len(txt) == 1: return txt.isalpha()
    return txt[-1].isalpha() and txt[-2] == " "
```

Example 6: An example of Incorrect condtion made by ChatGPT

In this example, compared with the ground truth, ChatGPT does not take the case that *the last character might be a space* into account. For instance, if the given input is "`apple pi e `", ideally, the function should return `False`. However, the code generated by ChatGPT would return `True` because the `strip` method would delete the spaces at the end of the string. These results indicate that it is still challenging for LLMs to handle complex logical conditions.

Additionally, our analysis reveals that the two open-sourced LLMs, CodeGen-16B and InCoder-1.3B, grapple with a notable proportion (17% and 15%, respectively) of "`Missing multiple statements`" errors. However, this category is relatively less frequent in ChatGPT, with only 6% of the total errors falling into this

classification. A plausible explanation is that ChatGPT is better at understanding natural language input (task requirements). For instance (Example 7), both CodeGen-16B and InCoder-1.3B miss the requirements that "*the elements must be even numbers*", resulting in missing statements within the generated code.

```
# Given a non-empty list of integers lst. Add the even elements
# that are at odd indices.

# Incorrect code by CodeGen-16B
def add(lst):
    return sum(lst[1::2])

# Incorrect code by InCoder-1.3B
def add(lst):
    return sum([lst[i] for i in range(1, len(lst), 2)])

# Groundtruth
def add(lst):
    s = 0
    for i in range(1, len(lst), 2):
        if lst[i] % 2 == 0:
            s += lst[i]
    return s
```

Example 7: An example of missing statements by CodeGen-16B and InCoder-1.3B

Finding 3

The most common error semantic types among three LLMs are "`Garbage Code`" and "`Condition Error`". Compared with CodeGen-16B and InCoder-1.3B, ChatGPT deos not often generate code with missing statements.

Furthermore, we find that even for a same failure task, different LLMs could have different semantic error types. Out of 25 problems that all three LLMs fail to solve, only one problem triggers the exact same semantic error for these LLMs. This indicates that different LLMs could behave differently on a same task. Example 8 shows an example that three LLMs made different semantic errors.

```
# Complete the function that takes two integers and returns the
# product of their unit digits.

# Incorrect code by CodeGen-16B
def multiply(a, b):
    return a * b

# Incorrect code by InCoder-1.3B
def multiply(a, b):
    product = 1
    while a > 0:
        product *= a
        a -= 1
    while b > 0:
        product *= b
        b -= 1
    return product

# Incorrect code by ChatGPT
def multiply(a, b):
    return abs(a % 10 * b % 10)

# Groundtruth
def multiply(a, b):
    return int(str(a)[-1]) * int(str(b)[-1])
```

Example 8: An example of three LLMs made different errors on a same task

In this example, CodeGen-16B miss a few statements of handling the "*unit digits*" ("`Missing multiple statements`"), while InCoder-1.3B completely ignored the task ("`Wrong (logical) direction`"). ChatGPT only made a small mistake with parentheses ("`Incorrect arithmetic operation`"). One takeaway from such observation is that *ensemble* of different LLMs might improve the code generation successful rate.

Finding 4

Different LLMs could commit completely different semantic errors with a same programming task.

Syntactic Errors. Figure 3 shows the distribution of *Syntactic Errors* made by three LLMs. We can observe that all three LLMs again share a similar distribution of syntactic errors. The most common syntactic error type is "`Code Block Error`", where 53.2%, 60.0%, and 43.2% of the errors made by CodeGen-16B, InCoder-1.3B, and ChatGPT fall into this category, respectively. We believe this could be largely attributed to an LLM's mis-interpret the task requirements. In such cases, a code patch might be needed to fix "`Incorrect code block`" or "`Missing code block`". Example 9 shows an example that CodeGen-16B only generated comments.

```
# Returns a list l' such that l' is identical to l in the indicies
# that are not divisible by three, while its values at the
# indicies that are divisible by three are equal to the values of
# the corresponding indicies of l, but sorted.

def sort_third(l: list):
    # l' = [l[i] for i in range(len(l)) if i % 3 != 0]
    # l' = [l[i] for i in range(len(l)) if i % 3 != 0]
    ...
```

Example 9: An example of missing code blocks by CodeGen-16B

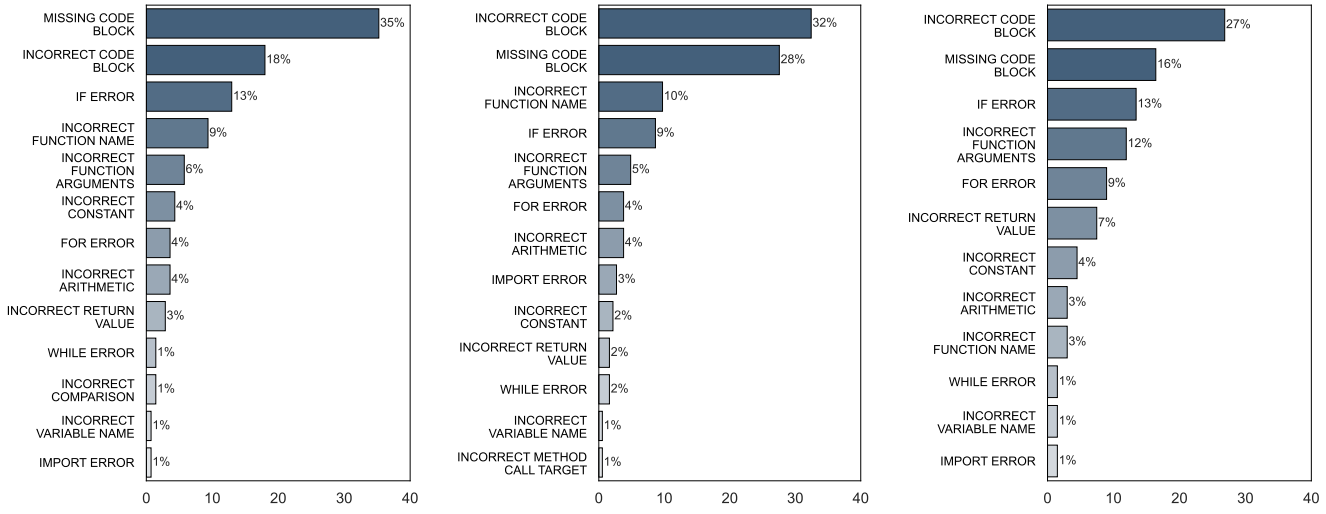
Another major type of syntactic errors is "`Method Call Error`". Specifically, the two open-sourced LLMs have more cases of "`Incorrect funtion name`", while ChatGPT encounters more "`Incorrect function arguments`". Example 10 shows an example that ChatGPT generated incorrect arguments for the "`spilt()`" method.

```
# You'll be given a string of words, and your task is to count the
# number of boredoms. A boredom is a sentence that starts with the
# word "I". Sentences are delimited by '.', '?' or '!'.

def is_bored(S):
    sentences = S.split('.')
    sentences += S.split('?')
    sentences += S.split('!')
    count = 0
    for sentence in sentences:
        if sentence.strip(). startswith('I') :
            count += 1
    return count
```

Example 10: An example of incorrect function arguments generated by ChatGPT

The task requires the generated code to *count the sentence that starts with the word "I"*. Therefore, the correct code should change the argument `'I'` to `'I '` (a space is added).



(a) CodeGen-16B **(b) InCoder-1.3B** **(c) ChatGPT**

Figure 3: Distribution of *Syntactic Errors* made by three LLMs.

Finding 5

A large number of syntactic errors made by three LLMs could be grouped into "`Code Block Error`". Besides, "`Method Call Error`" also poses threats to the correctness of the generated code.

5 IMPLICATIONS AND FUTURE OPPORTUNITIES

Our study reveals several significant implications for the code generation by LLMs and unveils promising research opportunities to improve the interpretability and reliability of code LLMs.

Firstly, we identify a significant proportion of semantic errors originating from LLMs categorized as "`Garbage Code`". This highlights instances where LLM behavior proves challenging to decipher for human developers, exemplified by the production of meaningless code snippets (Listing 1) or comments only (Listing 9). These findings underscore potential research opportunities in the realm of interpreting LLMs and the domain of explainable AI (XAI). For instance, in cases where LLMs recurrently generate similar statements, the employment of XAI techniques (such as attention-based [46, 53, 54] or perturbation-based [29, 49]) could be explored to ascertain if the model has compromised its capacity for capturing long-term dependencies, i.e., if it loses track of task requirements when generating new content. Additionally, when LLMs miss parts of task requirements, an examination of whether the model exhibits diminished attention to corresponding tokens could be undertaken. By seamlessly integrating appropriate XAI methodologies, developers might gain insight into the underlying causes of LLM's critical generation errors and potentially devise strategies for enhancing the model's performance.

Secondly, although a few minor syntactic errors can potentially be rectified through existing software repair techniques, yet, a larger number of errors require non-trivial repair efforts due to their roots in task requirement misinterpretation (Listing 7 and Listing 10). This reveals the distinction between repairing machine-generated and human-authored code, resulting in the need of novel error-handling mechanisms. For instance, when repairing an incorrect code snippet generated by LLMs, it might be imperative to first determine if the LLM has comprehensively grasped the task requirements. If this understanding is established, some of the errors might be directed towards existing software repair methodologies [32]; alternatively, if the comprehension is lacking, a repair approach that compels the LLM to accurately interpret the task prerequisites becomes pertinent. Proposing these new repairing techniques would help improve an LLM's code generation accuracy.

Finally, our study has only studied the error patterns of Python code generation. Since SOTA LLMs [2, 33] are able to support multiple programming languages, it is also worthwhile to expand our study with programming tasks from other languages (e.g., C/C++ and Java [55]). Extensive studies would help improve our taxonomy of code generation errors, thereby empowering developers with a more profound and comprehensive understanding of LLMs' capabilities in code generation.

6 THREATS TO VALIDITY

Internal Validity. Potential threats come from our manual analysis process. Labeling a code snippet's error type is subjective, especially for the semantic errors. Different labelers might have different determinations of a same code snippet. To mitigate this, we first performed open-coding and iteratively refined our codebook until a substantial agreement is achieved. Our final Fleiss' Kappa regarding the semantic and syntactic errors are both above 0.68.

External Validity. One potential threat lies in the choice of dataset. We have only labeled the HumanEval dataset, which only includes Python programming tasks. Therefore, our established taxonomy and findings might not generalize to other programming languages

and datasets. Furthermore, we have only experimented three LLMs released in 2022. It is unclear whether our findings can generatlize to the most recent LLMs, e.g., GPT4 [33], StarCoder [24], and SantaCoder [4].

Construct Validity. Our taxonomy is built from an open-coding process on a sampled subset of errors. Therefore, there is a potential threat that specific types of errors were missed during the establishment of the codebook. To mitigate this, we sampled errors from five different LLMs to include a diverse set of errors.

7 CONCLUSION

In this paper, we present an empirical study on code generation errors made by large language models. We first derived a taxonomy of LLMs' code generation errors based on SOTA LLM's failure cases within the HumanEval dataset [12] through open-coding and iterative refinements. Furthermore, we labeled errors committed by three SOTA code generation LLMs based on the established taxonomy. Through the investigation of three research questions, we find that despite the difference of successful rates between LLMs, a similar distribution of semantic and syntactic errors exists across different models. At the end of the paper, we further discuss the implications from our study and propose a few future research opportunities for improving LLMs' interpretability and reliability in code generation.

REFERENCES

[1] 2023. Amazon CodeWhisperer. https://aws.amazon.com/codewhisperer/.
[2] 2023. ChatGPT. http://chat.openai.com.
[3] Naser Al Madi. 2022. How readable is model-generated code? examining readability and visual inspection of github copilot. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.
[4] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).
[5] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403* (2023).
[6] Shushan Arakelyan, Rocktim Jyoti Das, Yi Mao, and Xiang Ren. 2023. Exploring Distributional Shifts in Large Language Models for Code Analysis. *arXiv preprint arXiv:2303.09128* (2023).
[7] Owura Asare, Meiyappan Nagappan, and N Asokan. 2022. Is github's copilot as bad as humans at introducing vulnerabilities in code? *arXiv preprint arXiv:2204.04741* (2022).
[8] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
[9] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.
[10] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).
[11] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* (2023).
[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[13] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).
[14] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems* 30 (2017).
[15] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860* (2019).
[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[17] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified language model pre-training for natural language understanding and generation. *Advances in neural information processing systems* 32 (2019).
[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[19] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.
[20] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).
[21] Kevin Jesse, Toufique Ahmed, Premkumar T Devanbu, and Emily Morgan. 2023. Large Language Models and Simple, Stupid Bugs. *arXiv preprint arXiv:2303.11455* (2023).
[22] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–23.
[23] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).
[24] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
[25] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210 [cs.SE]
[26] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2023. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. *arXiv preprint arXiv:2307.12596* (2023).
[27] Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, and Li Li. 2023. On the Reliability and Explainability of Automated Code Generation Approaches. *arXiv preprint arXiv:2302.09587* (2023).
[28] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2023. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *arXiv preprint arXiv:2308.04838* (2023).
[29] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).
[30] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 1–5.
[31] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[32] Wonseok Oh and Hakjoo Oh. 2022. PyTER: effective program repair for Python type errors. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 922–934.
[33] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[34] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.
[35] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the Effectiveness of Large Language Models in Code Translation. *arXiv preprint arXiv:2308.03109* (2023).
[36] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.
[37] 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).

[38] Atsushi Shirafuji, Yutaka Watanobe, Takumi Ito, Makoto Morishita, Yuki Nakamura, Yusuke Oda, and Jun Suzuki. 2023. Exploring the Robustness of Large Language Models for Solving Programming Problems. *arXiv preprint arXiv:2306.14583* (2023).

[39] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 71–82.

[40] Anselm L Strauss and Juliet Corbin. 2004. Open coding. *Social research methods: A reader* (2004), 303–306.

[41] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[42] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[43] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[44] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[46] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*. 2377–2388.

[47] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[48] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[49] Zhiyong Wu, Yun Chen, Ben Kao, and Qun Liu. 2020. Perturbed Masking: Parameter-free Probing for Analyzing and Interpreting BERT. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4166–4176.

[50] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[51] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778* (2023).

[52] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. 62–71.

[53] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 39–51.

[54] Kechi Zhang, Ge Li, and Zhi Jin. 2022. What does Transformer learn about source code? *arXiv preprint arXiv:2207.08466* (2022).

[55] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568* (2023).

[56] Terry Yue Zhuo, Zhuang Li, Yujin Huang, Yuan-Fang Li, Weiqing Wang, Gholamreza Haffari, and Fatemeh Shiri. 2023. On robustness of prompt-based semantic parsing with large pre-trained language model: An empirical study on codex. *arXiv preprint arXiv:2301.12868* (2023).