Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models

Zhijie Wang*†, Zijie Zhou‡†, Da Song*†, Yuheng Huang§, Shengmai Chen‖, Lei Ma§*, Tianyi Zhang‖
* University of Alberta, Edmonton, AB, Canada ‡ University of Illinois Urbana-Champaign, Champaign, IL, USA
§ The University of Tokyo, Tokyo, Japan ‖ Purdue University, West Lafayette, IN, USA
zhijie.wang@ualberta.ca, zijiez4@illinois.edu, dsong4@ualberta.ca, yuhenghuang42@g.ecc.u-tokyo.ac.jp,
chen3301@purdue.edu, ma.lei@acm.org, tianyi@purdue.edu

***Abstract*—Large Language Models (LLMs) have demonstrated unprecedented capabilities in code generation. However, there remains a limited understanding of code generation errors that LLMs can produce. To bridge the gap, we conducted an in-depth analysis of code generation errors across six representative LLMs on the HumanEval dataset. Specifically, we first employed open coding and thematic analysis to distill a comprehensive taxonomy of code generation errors. We analyzed two dimensions of error characteristics—*semantic characteristics* and *syntactic characteristics*. Our analysis revealed that LLMs often made non-trivial, multi-line code generation errors in various locations and with various root causes. We further analyzed the correlation between these errors and task complexity as well as test pass rate. Our findings highlighted several challenges in locating and fixing code generation errors made by LLMs. In the end, we discussed several future directions to address these challenges.**

***Index Terms*—Empirical Study, Code Generation, Large Language Models**

I. Introduction

Automatically generating code from natural language has been a long-term pursuit across multiple research communities. Recent advances in Large Language Models (LLMs) have led to rapid, unprecedented improvements on this task [1]–[5]. Despite this great progress, LLMs still cannot reliably generate correct code for many tasks. Currently, there is a lack of deep understanding of the cases where LLMs fail. Specifically, it remains unclear *what types of code generation errors an LLM typically produces* and *whether different LLMs make similar errors*. Answering these questions would help researchers gain insights into the limitations of existing models and identify opportunities for model improvement.

To bridge this knowledge gap, we conducted an in-depth analysis of code generation errors made by LLMs. We focused on six popular LLMs: CodeGen-16B [1], InCoder-1.3B [2], GPT-3.5 [5], GPT-4 [4], SantaCoder [6], and StarCoder [3]. These models produced 557 incorrect code solutions on the 164 tasks from the HumanEval dataset [7]. Four of the authors worked together to locate the erroneous parts of these incorrect solutions and manually fix them. Specifically, for some tasks, LLMs may propose an alternative solution that differs from the ground truth solution in HumanEval, e.g., using a lambda expression to process a sequence of data instead of using a loop. To avoid overfitting the ground truth, the authors manually located and fixed errors following the problem-solving direction of the LLM instead of simply comparing the LLM-generated code with the ground truth.

We performed multiple rounds of open coding and iterative refinement to analyze the characteristics of the located errors. Specifically, we analyzed these errors alongside two dimensions—the *semantic characteristics* and *syntactic characteristics* of these errors:

- **Semantic characteristics** can help identify the high-level root causes of these code generation errors. Representative semantic characteristics include *missing condition*, *wrong (logical) direction*, *incorrect condition*, etc. Analyzing these semantic characteristics can help understand the limitations of current LLMs in interpreting task requirements and generating semantically correct programs.
- **Syntactic characteristics** can help localize where the error occurs in an incorrect code solution. Representative syntactic characteristics include *incorrect code blocks*, *incorrect function arguments*, etc. Understanding these characteristics allows for a better assessment of current LLMs' abilities to generate different kinds of code constructs. It can also help inform the design of new techniques for localizing and repairing code generation errors made by LLMs.

Our analysis shows that the majority of code generation errors involve multiple lines of code, rather than simple errors. These errors often require substantial code restructuring and repair rather than simple fixes. Furthermore, while the overall distribution of the syntactic characteristics of these errors (i.e., error locations) is similar across different LLMs, the semantic characteristics of the errors (i.e., root causes) vary significantly for different LLMs even for the same task. Interestingly, most of the incorrect code solutions are compilable and runnable without any compilation errors. Thus, we cannot easily capture these errors via compiler check. Careful code review and high-quality test cases are necessary to capture these errors. This also implies that modern LLMs have adequately learned the syntax rules of programming languages, but struggle with understanding intricacies in natural language task descriptions and generating delicate code with sophisticated logic.

In summary, this paper makes the following contributions:

- We established a taxonomy of both syntactic and semantic

† The first three authors contributed equally to this work. Zijie Zhou was a remote research intern at Purdue University.

characteristics of code generation errors through open coding and thematic analysis. Our labeling results are available at a GitHub repository [8].

- We analyzed the similarities and differences in the errors made by different LLMs, as well as the bug-fixing effort, the impact of task complexity, and the correlation between test pass rates and different kinds of errors.
- We discussed the implications and future opportunities for improving LLMs for code generation.
- We developed an interactive data analysis website to help researchers and developers examine and explore code generation errors in different categories. The website is available at https://llm-code-errors.cs.purdue.edu.

II. Methodology

A. Research Questions

This study investigates the following research questions.

- *RQ1: What kinds of code generation errors do different LLMs make?* This question aims to uncover the common characteristics and distinctions of code generation errors made by different LLMs. This can help us understand whether it is feasible to develop generic methods to improve LLMs or whether these models need specialized treatment.
- *RQ2: How much effort is needed to fix code generation errors?* In practice, it is unrealistic to expect LLMs to generate fully correct code for every possible scenario. Existing studies show that some incorrect code can still serve as a useful starting point for developers [9], [10]. Thus, it is important to understand what efforts are needed to fix the incorrect solutions and whether it is possible to automate the repair. This question aims to fill this knowledge gap.
- *RQ3: How does the task complexity affect an LLM's code generation?* Intuitively, complex tasks are more challenging to solve than simple tasks. Yet it is unclear whether different LLMs exhibit different code generation capabilities when solving tasks of different complexity levels. Specifically, it would be useful to find out whether there is an upper bound on the complexity of tasks that LLMs can eloquently solve, which can then be used to guide or estimate the effort required for code review, testing, and repair.
- *RQ4: Does partially failed code exhibit different characteristics compared with fully failed code?* This question explores the distinctions between code that fails a subset of test cases and code that fails all test cases. It can offer insights into the specific challenges faced in achieving full correctness.

B. Code Generation LLMs

In this study, we focus on six representative code generation LLMs: CodeGen-16B [1], InCoder-1.3B [2], GPT-3.5 [5], GPT-4 [4], SantaCoder [6], and StarCoder [3]. As shown in Table I, these models cover a wide range of model sizes and model performance. CodeGen-16B was trained on 217GB Python code from BigPython [1]. InCoder-1.3B was trained on 159GB of open-source repositories from GitHub, GitLab, and StackOverflow. SantaCoder and StarCoder were trained on The Stack dataset [11]. The training data of GPT-3.5 and GPT-4 are currently unknown. As GPT-3.5 and GPT-4 are constantly evolving, we used GPT-3.5-Turbo-0301 and GPT-4-0314, the two most recent model checkpoints at the time of our analysis.

TABLE I: Code generation LLMs used in this study

Model	**Release**	**Size**	**Performance**	
			Pass@1	*Incorrect Solutions*
CodeGen-16B [1]	Mar. 2022	16B	32.9%	110
InCoder-1.3B [2]	Apr. 2022	1.3B	12.2%	144
GPT-3.5 [5]	Nov. 2022	175B	73.2%	42
GPT-4 [4]	Mar. 2023	1.7T	89.0%	18
SantaCoder [6]	Apr. 2023	1.1B	14.6%	139
StarCoder [3]	May. 2023	15.5B	34.1%	104

C. Collection of Incorrect Code Solutions

In this study, we utilize the widely used HumanEval benchmark [7] to collect code generation errors made by LLMs. HumanEval includes 164 hand-written Python programming tasks, each accompanied by an average of 7.7 unit tests. These tasks involve language comprehension, reasoning, algorithms, and simple mathematics. For each task, we followed the common practice in benchmarking the performance of code LLMs [1]–[3] to prompt each LLM with the original prompt from HumanEval, which includes a task description and several exemplary test cases (2.7 on average). While there are more advanced prompting strategies to augment LLMs for code generation, we are more interested in the innate capability of LLMs as the first step to understanding their limitations. Nevertheless, we discuss this as a threat to validity in Sec. VI. We also used greedy decoding with the temperature set to 0 to ensure the reproducibility of our results. Then, we executed the test cases to identify incorrect solutions. We also performed a round of manual checks to find solutions that pass test cases but are not fully correct since some tasks may not have sufficient test cases. We found 19 such cases. Example 1 shows an incorrect solution generated by GPT-3.5, which fails to handle the case where `x` is 0. In this scenario, the output should be `"0"` instead of an empty string. However, such test cases are absent in the HumanEval benchmark. In the end, we identified a total of 557 incorrect code solutions generated by the six models. Table I shows the distribution.

```python
# [Task 44] Change numerical base of input x to base.
def change_base(x, base):
    result = ""
    while x > 0: result, x = str(x%base) + result, x//base
    return result
```

Example 1: Incorrect solution that passed all test cases

D. Manual Analysis of Incorrect Code Solutions

We performed open coding [12]–[15] to analyze the characteristics of the 557 incorrect code solutions and developed a taxonomy of code generation errors made by LLMs.

Open coding. From the 557 incorrect solutions, we first randomly sampled 160 of them as a starting point for analysis. The sample size is statistically significant, with a 90% confidence level and a 5.5% margin of error. Two authors independently identified the erroneous parts of each incorrect solution and documented the root causes of the errors. For incorrect code solutions with multiple errors, the authors labeled the characteristics of each individual error. Since LLMs may

TABLE II: Taxonomy of semantic characteristics of code generation errors made by LLMs.

Error Characteristic		Example of Incorrect Solutions	Correct Solution

Condition Error	Missing condition	# [Task 151] CodeGen-16b
		def double_the_difference(lst):
		    sum = 0
		    for i in lst:
		        if i > 0 and i % 2 != 0:
		            sum += i ** 2
		    return sum
			# [Task 151] Ground Truth
			def double_the_difference(lst):
			    ans = 0
			    for num in lst:
			        if num % 2 == 1 and num >
			                0 and "." not in str(num): ans
			                += num ** 2
			    return ans

	Incorrect condition	# [Task 44] GPT-3.5
		def change_base(x, base):
		    r = ""
		    while x > 0:
		        rem, r = x % base, str(rem)+r
		        x = x // base
		    return result
			# [Task 44] Ground Truth
			def change_base(x, base):
			    if x == 0: return "0"
			    ret = ""
			    while x != 0:
			        ret, x = str(x%base) + ret, x // base
			    return ret

Constant Value Error	Constant value error	# [Task 51] SantaCoder
		def remove_vowels(text):
		    vowels = 'aeiou'
		    return ''.join([char for char in text if
		            char not in vowels])
			# [Task 51] Ground Truth
			def remove_vowels(text):
			    return "".join(list(filter(lambda ch: ch
			            not in "aeiouAEIOU", text)))

Reference Error	Wrong method/variable	# [Task 79] InCoder-1B
		def decimal_to_binary(decimal):
		    return str(decimal) + "db"
			# [Task 79] Ground Truth
			def decimal_to_binary(decimal):
			    return bin(decimal) + "db"

	Undefined name	# [Task 83] CodeGen-16B
		def starts_one_ends(n):
		    if n == 1: return 1
		    return starts_one_ends(n-1)+
		            ends_one_starts(n-1)
			# [Task 83] Ground Truth
			def starts_one_ends(n):
			    if n == 1: return 1
			    return 18 * 10 ** (n - 2)

Operation/
Calculation Error
	Incorrect arithmetic operation	# [Task 44] InCoder-1B
		def change_base(x: int, base: int):
		    digits = []
		    while x:
		        digits.append(str(x % base))
		        x /= base
		    return ''.join(reversed(digits))
			# [Task 44] Ground Truth
			def change_base(x: int, base: int):
			    if x == 0: return "0"
			    ret = ""
			    while x != 0:
			        ret = str(x % base) + ret
			        x //= base
			    return ret

	Incorrect comparison operation	# [Task 138] CodeGen-16B
		def is_equal_to_sum_even(n):
		    return n <= 8 and n % 2 == 0
			# [Task 138] Ground Truth
			def is_equal_to_sum_even(n):
			    return n >= 8 and n % 2 == 0

Garbage Code	Only comments	# [Task 152] InCoder-1B
		def compare(game,guess):
		    # Regenerate the task description.
			# [Task 152] Ground Truth
			def compare(game,guess):
			    return [abs(game[i] - guess[i]) for i in
			            range(len(game))]

	Meaningless code snippet	# [Task 138] StarCoder
		def is_equal_to_sum_even(n):
		    pass
			# [Task 138] Ground Truth
			def is_equal_to_sum_even(n):
			    return n >= 8 and n % 2 == 0

	Wrong (logical) direction	# [Task 20] InCoder-1.3B
		def find_closest_elements(numbers):
		    closest_to_one, closest_to_two = numbers
		            [0], numbers[1]
		    for number in numbers:
		        if number < closest_to_one:
		            closest_to_one = number
		        if number > closest_to_two:
		            cloeset_to_two = number
		    return closest_to_one, closest_to_two
			# [Task 20] InCoder-1.3B
			def find_closest_elements(numbers):
			    min_diff, min_pair = float("inf"), None
			    for l, r in zip(numbers[:-1], numbers
			            [1:]):
			        diff = r - l
			        if diff < min_diff:
			            min_diff = diff
			            min_pair = (l, r)
			    return min_pair

Incomplete Code/
Missing Steps
	Missing one step	# [Task 16] InCoder-1B
		def count_distinct_chars(string):
		    return len(set(string))
			# [Task 16] Ground Truth
			def count_distinct_chars(string):
			    return len(set(string.lower()))

	Missing multiple steps		

Memory Error	Infinite loop	# [Task 100] CodeGen-16b
		def make_a_pile(n):
		    if n % 2 == 0:
		        return [n] + make_a_pile(n+2)
		    else:
		        return [n] + make_a_pile(n+1)
			# [Task 100] Ground Truth
			def make_a_pile(n):
			    ans, num = [], n
			    for _ in range(n):
			        ans.append(num)
			        num += 2
			    return ans

generate alternative solutions compared with the ground-truth solution from HumanEval, we chose to manually debug the incorrect solution rather than simply comparing it with ground truth. Specifically, the two authors executed the failed test cases and performed step-by-step debugging to locate the errors and identify their root causes.

The authors documented all error locations and root causes and discussed them with other authors after the initial coding. They refined code labels and came up with an initial codebook. At this stage, we found that code generation errors made by LLMs can be categorized along two dimensions based on their *semantic* and *syntactic* characteristics. Semantic characteristics help identify the high-level root causes of code generation errors, such as a wrong logical direction to solve the task. In contrast, syntactic characteristics assist in error localization, such as determining whether the error is in the method name or the arguments. The initial codebook includes seven semantic characteristics and eight syntactic characteristics.

Iterative refinement of the codebook. After obtaining the initial codebook, we invited another two authors to iteratively improve the codebook. The four authors first independently analyzed 10 incorrect code snippets following the same procedure described above and labeled the error characteristics based on the initial codebook. If a new characteristic was identified, an author created a new label to describe the characteristic.

After the first round of labeling, we computed Fleiss' Kappa [16] to measure the inter-rater agreement. We used Fleiss' Kappa instead of Cohen's Kappa, since we had more than two labelers and more than two labels. The initial scores were 0.37 and 0.32 for semantic characteristics and syntactic characteristics, respectively [17]. To figure out where the disagreements were, the four authors met to discuss the disagreements and exchanged opinions about updating the codebook. They found that the low agreement was due to missing error characteristics in the initial codebook.

The four authors then refined the codebook with 11 semantic characteristics and 13 syntactic characteristics and labeled another batch of 10 incorrect solutions. The Fleiss' Kappa scores of this round of labeling were 0.68 and 0.69 for semantic characteristics and syntactic characteristics, respectively, indicating substantial agreement [17]. The authors further discussed the disagreements and refined the codebook with 13 semantic characteristics and 14 syntactic characteristics. Then, they conducted the third round of labeling with a new batch of

TABLE III: Taxonomy of syntactic characteristics of code generation errors made by LLMs.

Error Characteristic		Example of Incorrect Solutions	Correct Solution

Conditional Error	If error	`# [Task 151] CodeGen-16b`
		`def double_the_difference(lst):`
		`    sum = 0`
		`    for i in lst:`
		`        if i > 0 and i % 2 != 0:`
		`            sum += i ** 2`
		`    return sum`
			`# [Task 151] Ground Truth`
			`def double_the_difference(lst):`
			`    ans = 0`
			`    for num in lst:`
			`        if num%2==1 and num>0 and "." not`
			`            in str(num): ans += num ** 2`
			`    return ans`

Loop Error	For error	`# [Task 121] GPT-3.5`
		`def solution(lst):`
		`    sum = 0`
		`    for i in range(1, len(lst), 2):`
		`        if lst[i] % 2 != 0: sum +=lst[i]`
		`    return sum`
			`# [Task 121] Ground Truth`
			`def solution(lst):`
			`    return sum([x for idx, x in`
			`        enumerate(lst) if idx%2==0 and x`
			`            %2==1])`

	While error		

Return Error	Incorrect return value	`# [Task 103] GPT-3.5`
		`def rounded_avg(n, m)`
		`   if n > m: return -1`
		`   avg=round(sum(range(n,m+1))/(m-n+1))`
		`   return bin(avg)[2:]`
			`# [Task 103] Ground Truth`
			`def rounded_avg(n, m)`
			`    if n > m: return -1`
			`    avg = round((n + m) / 2)`
			`    return bin(avg)`

Method Call Error	Incorrect function name	`# [Task 54] StarCoder`
		`def same_chars(s0, s1):`
		`    return sorted(s0) == sorted(s1)`
			`# [Task 54] Ground Truth`
			`def same_chars(s0, s1):`
			`    return set(s0) == set(s1)`

	Incorrect function arguments		
	Incorrect method call target		

Assignment Error	Incorrect constant	`# [Task 138] InCoder-1.3B`
		`def is_equal_to_sum_even(n):`
		`    return n >= 4 and n % 2 == 0`
			`# [Task 138] Ground Truth`
			`def is_equal_to_sum_even(n):`
			`    return n >= 8 and n % 2 == 0`

	Incorrect arithmetic		
	Incorrect variable name		
	Incorrect comparison		

Import Error	Import error	`# [Task 133] StarCoder`
		`def sum_squares(lst):`
		`    return sum[int(math.ceil(i)**2 for i in`
		`        lst]`
			`# [Task 133] Ground Truth`
			`def sum_squares(lst):`
			`    import math`
			`    return sum(map(lambda x: math.ceil(x)**2,`
			`        lst))`

Code Block Error	Incorrect code block	`# [Task 83] InCoder-1.3B`
		`def starts_one_ends(n):`
		`    count = 0`
		`    while n > 0:`
		`        count, n = count + 1, n / 10`
		`    return count`
			`# [Task 83] Ground Truth`
			`def starts_one_ends(n):`
			`    if n == 1:`
			`        return 1`
			`    return 18 * 10 ** (n - 2)`

	Missing code block	`# [Task 60] CodeGen-16B`
		`def next_smallest(lst):`
		`    if len(lst)<2: return None`
		`    lst.sort()`
		`    return lst[1]`
			`# [Task 60] Ground Truth`
			`def is_prime(n):`
			`    if len(lst)<=1: return None`
			`    sorted_list=sorted(lst)`
			`    for x in sorted_list:`
			`        if x!=sorted_list[0]: return x`

10 errors. The authors did not find any new error characteristics in this round, and the Fleiss' Kappa scores increased to 0.84 and 0.71. At this point, the authors believed that the codebook was comprehensive enough. The final codebook includes 13 semantic characteristics and 14 syntactic characteristics.

Analyzing the remaining dataset. The two authors used the final codebook to label the remaining incorrect solutions. The final Fleiss' Kappa scores were 0.91 and 0.91 for semantic and syntactic characteristics, indicating perfect agreement [17]. They had disagreements on 29 errors' semantic characteristics and 28 errors' syntactic characteristics. These disagreements were resolved after discussing them with all the authors. No new error characteristics were found. The final coding results were documented in a spreadsheet and shared on GitHub [8]. The whole labeling process took about 328 person-hours.

E. Analysis of Repair Effort

To investigate the repair effort (RQ2), we employ three different metrics to measure the similarity between incorrect model-generated code and the corresponding correct solution. To ensure a fair comparison, we first removed all LLM-generated comments before calculation. We used Levenshtein distance [18] to compute the minimum number of edits (i.e., insertions, deletions, or substitutions) required to change an incorrect solution to the correct solution. We also used Jaccard similarity [19] as another textual similarity metric. Both of them are widely used for fault localization [20], [21]. We further used CodeBERTScore [22] to measure the semantic similarity between the incorrectly generated code and the ground truth.

Note that for some tasks, an LLM may propose an alternative solution with errors compared with the ground-truth solution in HumanEval. We identified 17 incorrect solutions where the LLM proposed an alternative way to solve the task but did not correctly solve it. In such cases, it is unfair to directly compare the incorrect code with the ground truth. To address this issue, one author manually solved the task following the LLM's solution and computed the metrics by comparing the incorrect solution with the alternative, correct solution.

III. Results

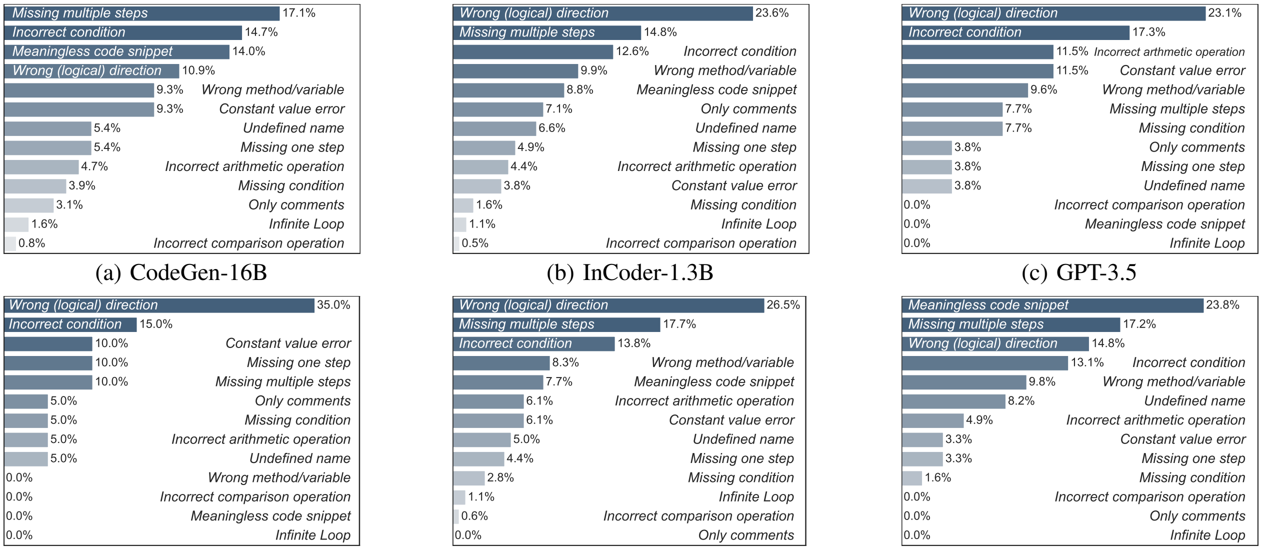
In this section, we denote the 164 programming tasks in HumanEval [7] as `Task 0-163`. Due to the page limit, some code examples are simplified. We refer the readers to our Github repository for more details [8].

A. RQ1: Characteristics of Code Generation Errors

Table II and Table III present the finalized taxonomy of code generation errors made by LLMs. The taxonomy categorizes code generation errors based on their semantic characteristics (i.e., root causes) and syntactic characteristics (i.e., error locations). In total, there are 13 semantic characteristics in 7 categories and 14 syntactic characteristics in 7 categories. We elaborate on each of them below.

1) Semantic Characteristics:

- **Condition Error** includes *missing condition* and *incorrect condition*. Missing condition is when a necessary condition is omitted, while incorrect condition is when an condition is incorrectly formulated in an if statements or a loop, leading to errors.
- **Constant Value Error** is an error that occurs when an incorrect constant value is set, which can occur in function arguments, assignments, or other parts of the code.
- **Reference Error** involves incorrect references to variables or functions, which includes the usage of an incorrect function



(a) CodeGen-16B (b) InCoder-1.3B (c) GPT-3.5

(d) GPT-4 (e) SantaCoder (f) StarCoder

Fig. 1: Distribution of *semantic characteristics* of code generation errors made by six LLMs.

or variable that does not match the requirement (*wrong method/variable*) and reference to a variable or method name that has not been defined (*undefined name*).

- **Operation/Calculation Error** indicates the mistakes in mathematical or logical operations, e.g., an incorrect comparison operation in a return statement "`return n <= 8`."
- **Garbage Code** is defined as unnecessary or irrelevant code that does not contribute to the intended functionality. It can occur in several forms: a *meaningless code snippet*, where the code, though syntactically correct, is irrelevant to the assigned task; *only comments*, where the code consists exclusively of comments without any executable statements; or *wrong (logical) direction*, where the code significantly deviates from the intended task logic or expected outcomes.
- **Incomplete Code/Missing Steps** indicates the absence of crucial steps to achieve the task.
- **Memory Error** includes *infinite loop*, which is a loop or recursion that never terminates.

Comparison between LLMs. Fig. 1 shows the distribution of the 13 semantic characteristics for each LLM. We find that several characteristics are frequently shared among all LLMs, such as *incorrect condition* and *wrong (logical) direction*. This implies that all LLMs struggle with certain kinds of task requirements, such as handling complex logic conditions, regardless of their model size and capability.

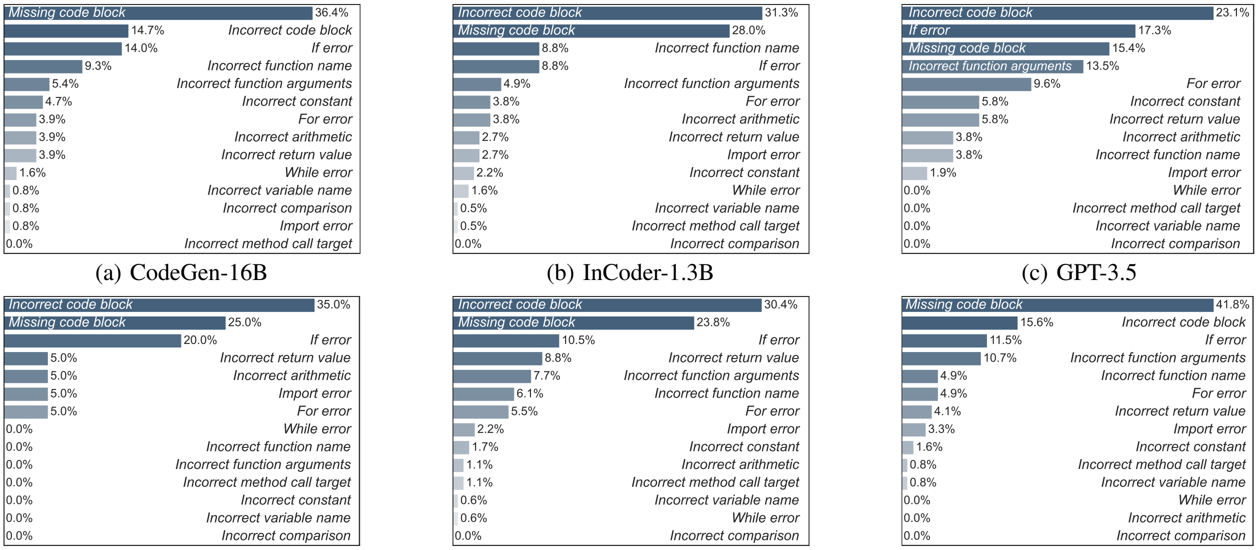
However, small models such as InCoder and CodeGen are more likely to generate *meaningless code* and code that *miss multiple steps*, while larger models such as GPT-3.5 and GPT-4 tend to make more *constant value errors* and *arithmetic operation errors*. Notably, incorrect code generated by GPT-4 only exhibited 9 of the 13 semantic characteristics, while incorrect code generated by smaller models exhibited all sorts of errors. One plausible reason is that GPT-3.5 and GPT-4 are much larger and are thus better at interpreting task descriptions. For instance, neither GPT-3.5 nor GPT-4 generated any meaningless code snippets. In contrast, 7% to 25% of the incorrect code solutions produced by the other four LLMs consist of meaningless code snippets.

> **Finding 1**: The most common semantic characteristics among six LLMs are *wrong (logical) direction* and *incorrect condition*, indicating that all LLMs struggle with interpreting complex task requirements and generating correct logic conditions. Compared with ultra-large models such as GPT-3.5, small models generate more meaningless code and code that misses multiple steps.

2) *Syntactic Characteristics:*

- **Conditional Error** indicates there is an error within the '`if`' statement, causing the code to behave incorrectly.
- **Loop Error** indicates there is an iteration mistake in the '`for`' or '`while`' loop, either through incorrect loop boundaries or mismanagement of loop variables.
- **Return Error** indicates the error is in a return statement that returns a wrong value or a value in the unexpected format.
- **Method Call Error** indicates the error is in a function call. It can be *incorrect function name*, wrong arguments (*incorrect function arguments*), or *incorrect method call target*.
- **Assignment Error** indicates the error is in an assignment statement. It can be an incorrect constant/variable name/comparison operator used in an assignment, leading to errors or unexpected behaviors in the code's execution.
- **Import Error** indicates the error is in an import statement.
- **Code Block Error** indicates multiple statements are incorrectly generated or omitted, leading to the task failure.

Comparison between LLMs. Fig. 2 shows the distribution of the 14 syntactic characteristics across the six LLMs. We observed similar distribution patterns as semantic characteristics. For all models, the top 3 error locations are either in entire code blocks (i.e., multiple statements in a sequence) or in an if statement. The fact that all LLMs struggle with generating entire code blocks correctly implies that many code generation errors are not small errors and require substantial efforts to fix, as investigated further in RQ2.



(a) CodeGen-16B (b) InCoder-1.3B (c) GPT-3.5

(d) GPT-4 (e) SantaCoder (f) StarCoder

Fig. 2: Distribution of *syntactic characteristics* of code generation errors made by six LLMs.

Compared with other models, the code generation errors from GPT-4 are more well-contained in a few types of code constructs. GPT-4 did not introduce any errors in method call expressions, variable references, or constant values used in an assignment statement. By contrast, GPT-3.5 still hallucinates when generating method calls. Other models exhibited a more diverse set of error locations compared with GPT-4 and GPT-3.5. Interestingly, CodeGen-16B and InCoder-1.3B have more cases of *incorrect function name*, while GPT-3.5, SantaCoder, and StarCoder encounter *incorrect function arguments* more frequently. This implies that during pre-training, CodeGen and InCoder are less effective in learning the mappings between task descriptions and which functions to use to achieve the tasks. One interesting direction to improve these models is to design pre-training tasks that predict function names and arguments to strengthen the model's memory of function usage.

Finding 2: More than 40% of the syntactic characteristics made by six LLMs are *missing/incorrect code block*. The studied LLMs also encountered a significant number of *if error* and *incorrect function name/argument*.

3) Mappings Between Semantic and Syntactic Characteristics: To further investigate the relationship between the semantic and syntactic characteristics of these code generation errors, we visualize their mappings as Sankey diagrams in Fig. 3. We observed that certain semantic characteristics are often paired with specific syntactic characteristics. For instance, *wrong (logical) direction* typically corresponds to *incorrect code block*. In other cases, a single syntactic characteristic can be associated with multiple semantic characteristics. For example, an *incorrect function argument* might arise from *constant value error*, *incorrect arithmetic operation*, or *wrong method/variable*. These results indicate that errors in similar locations may have various semantic root causes and thus require different kinds of fixes.

For example, to fix SantaCoder's solution for Task 50 (Example 2), one only needs to replace the constant `97` with `26`. By contrast, fixing the incorrect function argument in CodeGen-16B's solution for Task 149 is more challenging, as it requires generating an additional comparator function as the argument of the lambda function (Example 2).

```
# [Task 50] Decode the string that is encoded by shifting
# 5 characters in the alphabet
def decode_shift(s):
   return "".join([chr(((ord(ch) - 5 + 97) % 26) + ord("a"
        )) for ch in s])
```

```
# [Task 149] Delete the strings that have odd lengths.
# Return a sorted list ascending by length of each word.
# Sort alphabetically for words of the same length.
def sorted_list_sum(lst):
   return sorted(list(filter(lambda x: len(x)%2==0, lst)),
         key=len)
```

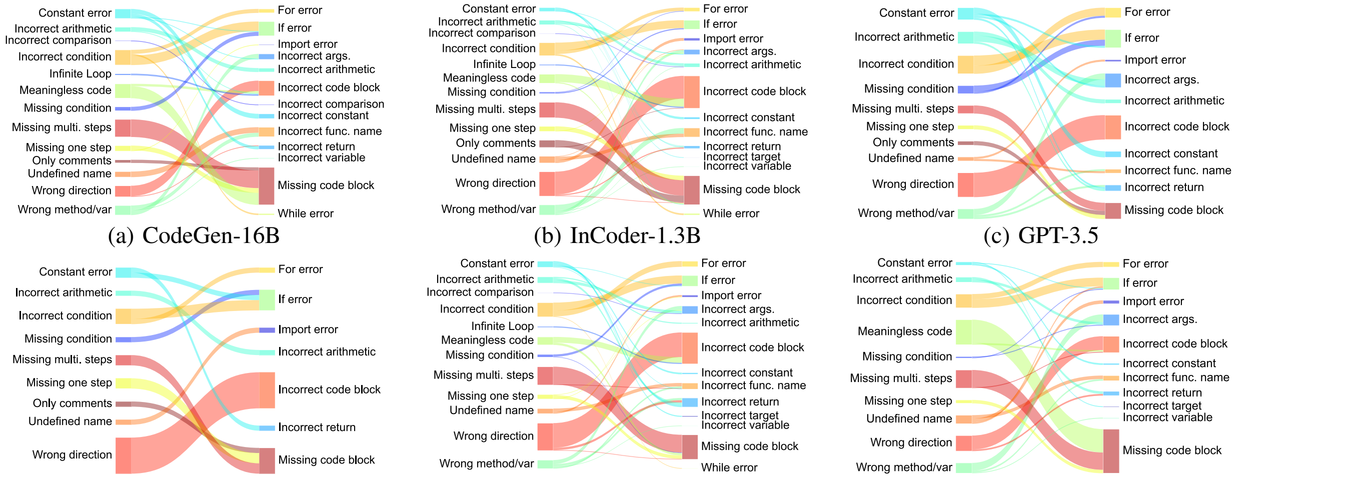
Example 2: *Incorrect function argument*

We also found that one kind of semantic characteristic, such as *constant value error*, may occur in different kinds of locations, such as if statements and function calls. Precisely locating such errors can, therefore, be challenging. For instance, in Task 91 (Example 3), GPT-4 should execute `startswith('I ')` instead of `startswith('I')` since a boredom is a sentence that starts with the word "I" rather than the character "I." While the fix itself is small, it requires a deep understanding of the task to derive the fix.

```
# [Task 91] 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.strip() for s in re.split('[.!?]', S)]
   return sum(1 for s in sentences if s.startswith("I"))
```

Example 3: *Constant value error* by GPT-4

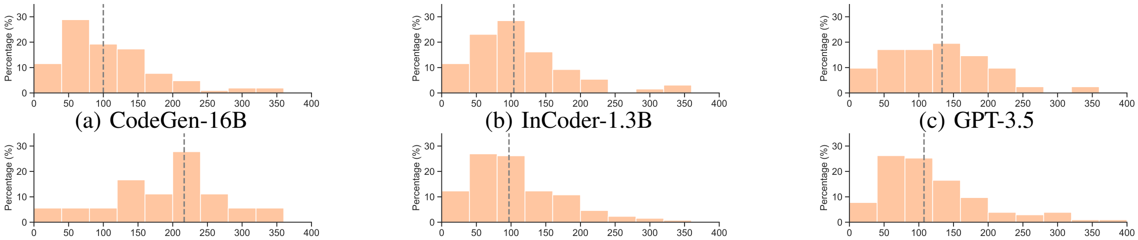
4) The Impact of Training Data: Regarding semantic characteristics, we found that GPT-3.5 and GPT-4 did not generate any *meaningless code snippets*, unlike the other four LLMs. This might be due to their training on significantly larger datasets, enhancing their ability to avoid meaningless outputs. In terms of syntactic characteristics, we observed that CodeGen-16B produced a smaller proportion of *incorrect code blocks* compared to the other models. This could be attributed to its specialized training with Python code data alone, which



(a) CodeGen-16B (b) InCoder-1.3B (c) GPT-3.5

(d) GPT-4 (e) SantaCoder (f) StarCoder

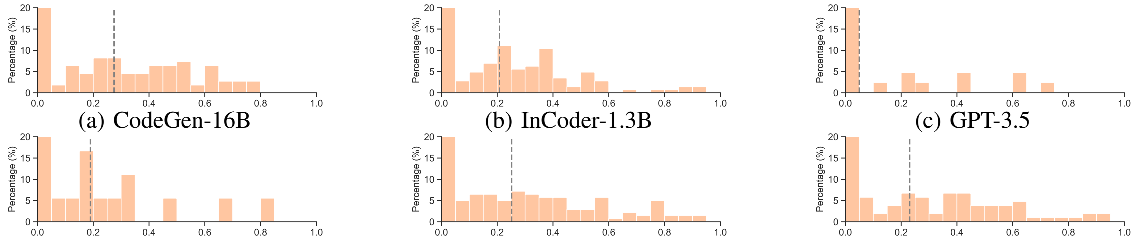
Fig. 3: Mappings between semantic and syntactic error characteristics of code generation errors made by six LLMs.



(a) CodeGen-16B (b) InCoder-1.3B (c) GPT-3.5

(d) GPT-4 (e) SantaCoder (f) StarCoder

Fig. 4: Levenshtein distance between the incorrect code and correct code. The vertical dashed lines indicate the medians.



(a) CodeGen-16B (b) InCoder-1.3B (c) GPT-3.5

(d) GPT-4 (e) SantaCoder (f) StarCoder

Fig. 5: CodeBERTScore between the incorrect code and correct code. The vertical dashed lines indicate the medians.

possibly contributed to reducing the generation of large chunks of syntactically incorrect code. In contrast, the other LLMs were trained on code corpora of multiple programming languages.

B. RQ2: Repair Effort for Code Generation Errors

Fig. 4 shows the distribution of Levenshtein distances. All models exhibit a wide range of Levenshtein distances for incorrect code, with median distances around or greater than 100. Notably, 84.21% of the incorrectly generated code has Levenshtein distance scores above 50 edits, with 52.63% of them requiring more than 200 edits. The results of Jaccard similarity show similar trends as the Levenshtein distances. Due to the page limit, we refer readers to our GitHub repository [8] for the detailed results of Jaccard similarity.

As shown in Fig. 5, the median CodeBERTScore values are approximately 0.2 for the six LLMs, except for GPT-3.5, which has a median similarity of 0.05. Additionally, a large portion of CodeBERTScore values across all six LLMs falls below 0.1. According to [22], a CodeBERTScore value of 0 indicates that two code snippets are unrelated, while a value of 1 indicates that the snippets are exactly the same. These results indicate that the majority of the incorrect solutions deviate significantly in semantics from the correct solutions. Overall, both low textual and semantic similarities suggest that the incorrect code generated by LLMs often exhibits big differences from the correct solutions, not just minor errors. Such observations are also aligned with our findings in RQ1—LLMs tend to make non-trivial errors such as *missing multiple steps* and *wrong logical direction*. Addressing these issues would require substantial edits. For instance, the median number of edits to fix errors of *missing multiple steps* is 108 edits.

Interestingly, GPT-3.5 and GPT-4, despite having high performance (Table I), exhibit larger deviations when generating incorrect code, with greater median Levenshtein distances compared to other models. This suggests that though GPT-3.5 and GPT-4 are more accurate in general, when they make mistakes, the mistakes are likely to cause a larger deviation (e.g., a large *incorrect code block*) from the correct solution and thus require more edits to fix.

Following the automated program repair literature [23], we also classify all incorrect code snippets into three categories

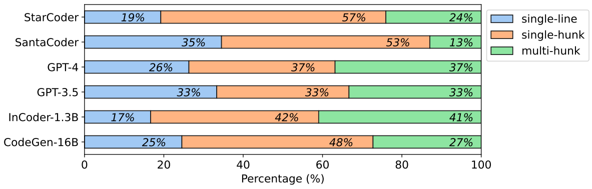


Fig. 6: Distribution of *single-line*, *single-hunk*, and *multi-hunk* faults in different LLMs

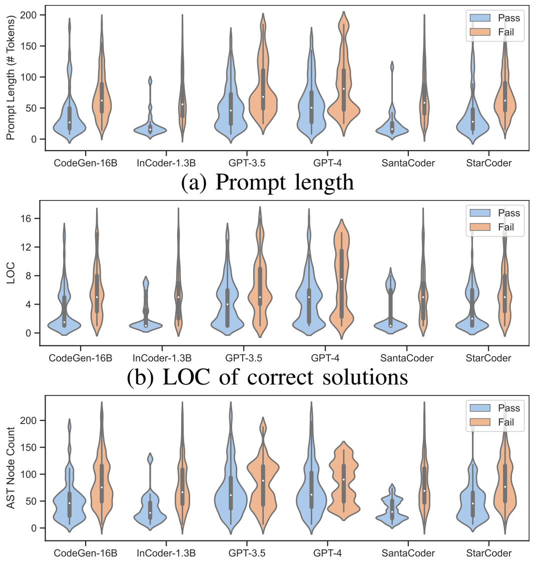
based on the effort required to fix them: (1) single-line errors, (2) single-hunk errors, and (3) multi-hunk errors. Specifically, a "hunk" refers to several contiguous lines in a program. Fig. 6 shows the distribution. Overall, the majority of errors are single-hunk or multi-hunk errors, which require substantial effort to repair compared with single-line errors. Compared with other LLMs, GPT-3.5 and GPT-4 exhibit a more balanced distribution among the three categories. SantaCoder made the most single-line errors (35%). StarCoder made the most single-hunk errors (57%). Notably, the least accurate model, InCoder-1.3B, made the most multi-hunk errors (41%).

Finding 3: The majority of incorrect code solutions generated by the six LLMs deviate significantly from the correct code. This implies that fixing LLM-generated code requires non-trivial efforts.

C. RQ3: Impact of Task Complexity

To the best of our knowledge, there is no established metric to measure task complexity. In this study, we used the length of the task description (i.e., the number of words in the prompt) and the length of the correct solution in terms of lines of code (LOC) as proxy metrics for task complexity. It is a pragmatic choice to enable objective measurements, but we acknowledge its limitation and discuss the potential threats to validity in Sec. VI. To avoid inflating the prompt length, we removed all test cases from the prompt. We also noticed that 20 ground-truth solutions only contain one LOC but with complex constructs such as lambda expressions. Therefore, we used the number of abstract syntax tree (AST) nodes as an alternative metric to LOC to verify its validity.

We investigated whether there was a significant difference between the complexity of successfully solved tasks and the complexity of failed tasks. We ran the Mann-Whitney U-test to examine the statistical difference in task complexity for each LLM. Fig. 7 shows the results. We observed a statistically significant difference in both prompt length and LOC of correct solutions among all six LLMs. Specifically, the p-values for prompt length are 4e-9, 5e-9, 2e-5, 2e-10, 2e-9, and 5e-4; and the effect sizes are 0.82, 1.38, 0.71, 0.77, 1.32, and 0.75 for the six models in the order listed in Fig. 7a. The p-values for LOC of correct solutions are 3e-6, 4e-5, 6e-3, 1e-2, 1e-5, and 1e-4; and the effect sizes are 0.72, 0.77, 0.63, 0.65, 0.69, and 0.73, respectively. The number of AST nodes shows a similar distribution as LOC (Fig. 7c), where the correct solutions of successfully solved tasks have significantly more AST nodes. The p-values are 2e-7, 4e-5, 6e-3, 1e-2, 1e-5, and 1e-4; the effect sizes are 0.65, 0.88, 0.30, 0.43, 0.64, and 0.74. We further calculated the Pearson correlation coefficient between the number of AST nodes and LOC, which shows a statistically significant correlation (r=0.864, p=4e-50). This shows the consistency of our findings based on different metrics.



(a) Prompt length

(b) LOC of correct solutions

(c) Number of AST nodes of correct solutions

Fig. 7: Distribution of passed/failed tasks' complexity

We further dig into each LLM's failed tasks with a prompt longer than 150 words. Based on the semantic taxonomy described in Sec. III-A, a large portion (64.0%) of these failed tasks are *Garbage Code* (i.e., unnecessary or irrelevant code that does not contribute to the intended functionality). For instance, Task 129's prompt includes 249 words, which requires returning a path with a length of k in a given grid that has the minimum sum of values alongside this path. InCoder-1.3B failed to understand the complex task requirement and only generated a sequence of meaningless appending operations (Example 4).

```
# [Task 129] Given a grid with N rows and N columns (N >=
# 2) and a positive k, each cell of ... (230 words left)
def minPath(grid, k):
   ans. visited = [], {}
   visited[grid[0][0]] = True
   ans.append(grid[0][0])
   ans.append(grid[0][1])
   ...
```

Example 4: *Meaningless code snippet* by InCoder-1.3B

We also observed similar patterns in the LOC metric. Specifically, we looked into programming tasks whose ground-truth solution has more than 12 lines of code. We found that 55.9% of these failed tasks are *Garbage Code*. For example, Task 105 requires three steps: 1) sorting an array of integers between 1 and 9 inclusive, 2) reversing the sorted result, and 3) replacing each digit with its corresponding name as a string. The correct solution has 24 lines of code. However, CodeGen-16B failed to understand the task requirements and returned only an empty array (Example 5).

```
# [Task 105] Given an array of integers, sort the integers
# that are between 1 and 9 inclusive, reverse the
# resulting array, and then replace each digit by its
# corresponding name "One", "Two" ... (144 words left)
def by_length(arr): return []
```

Example 5: *Meaningless code snippet* by CodeGen-16B

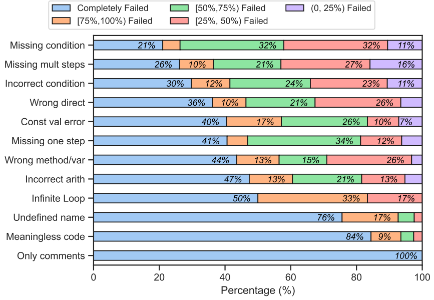


Fig. 8: Semantic characteristics vs. test pass rates

Finding 4: We observed a significant performance gap between short, simple problems and long, complex ones. When the task prompt exceeded 150 words or the correct solution required more than 12 lines of code (LOC), about 60% outputs are *Garbage Code*. This highlights the research opportunity of estimating the upper bound of the task complexity that code generation LLMs can handle.

D. RQ4: Impact of Test Pass Rate

Since HumanEval [7] provides test cases for each task, we are interested in whether completely failed code (i.e., failing all test cases) has different error characteristic patterns compared with partially failed code (i.e., failing a subset of test cases). Note that we excluded the 19 cases identified in Sec. II-C that passed all tests but are not equivalent to the ground truths.

Fig. 8 shows the test pass rate distribution of incorrect solutions with different semantic characteristics. *Only comments* and *meaningless code* are the top two characteristics that lead to complete failures. Though *undefined name* sounds like a small error, the majority of code with *undefined names* also leads to complete test failures due to runtime crashes. Surprisingly, while some error characteristics, such as *missing multiple steps*, sound severe by definition, they do not often lead to complete failures. After digging into some instances, we noticed that this was because LLMs did not completely misunderstand the task description, and the generated code could still pass some weak test cases.

For instance, Task 125 requires the program to split on commas if there is no space in the string. If there is also no comma in the string, the program should perform other operations. However, the code generated by CodeGen-16B only split the input string into spaces while missing the remaining steps. As a result, it can only pass the test cases that include spaces, leading to a partially failed task (Example 6).

```
# [Task 125] Given a string, return words split on space;
# if no space, split on ’,’; if no ’,’, return the number
# of lower-case letters with odd order in the alphabet.
def split_words(txt): return txt.split()
```

Example 6: *Missing multiple steps* by CodeGen-16B

Another surprising finding is that code with the wrong direction can pass some test cases accidentally. For example, in Task 75, the code generated by InCoder-1.3B does not follow the task instructions (Example 7). However, it can pass a few test cases, such as 5, 10, and 30. One plausible reason is that since the prompt includes test cases, LLMs may have memorized a superficial correlation between test cases and some other irrelevant solutions that pass those test cases.

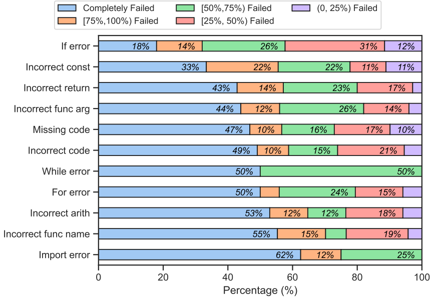


Fig. 9: Syntactic characteristics vs. test pass rates

```
# [Task 75] Return true if the given number is the
# multiplication of 3 prime numbers and false otherwise.
def is_multiply_prime(a): return a < 100 and a % 3 == 0
```

Example 7: *Wrong (logical) direction* by InCoder-1.3B

Finding 5: When LLM-generated code only passes a subset of the given test cases, it is more likely to contain errors with *missing multiple steps* and *incorrect condition*. By contrast, completely failed code (no test cases passed) have *meaningless code snippet* more frequently.

In terms of syntactic characteristics (Fig. 9), we observed an obvious difference on *if error*, where the majority of code with this error led to partial failure (82% v.s. 18%). This is because when encountering *if error*, LLMs were more likely to only misinterpret a specific condition requirement while generating correct code for the other parts. For instance, in Task 0, SantaCoder only considered the adjunct elements in the given list (Example 8). As a result, it failed to pass the test case where the two closet elements were not adjunct.

```
# [Task 0] Check if in the given list of numbers, are any
# two numbers closer to each other than given threshold.
def has_close_elements(numbers, threshold):
   if len(numbers) < 2: return False
   for i in range(len(numbers) - 1):
      if abs(numbers[i] - numbers[i+1]) > threshold:
         return True
   return False
```

Example 8: An *if error* example by SantaCoder

Finding 6: For syntactic characteristics, LLMs made more *if error* in partially failed tasks (82%) than in fully failed ones (18%). These subtle error characteristics in partially failed tasks, likely due to missed conditions, may be fixable with traditional automated program repair techniques.

IV. Discussion and Future Work

Our findings shed light on several future directions to improve the quality and reliability of LLM-generated code.

Repairing Errors in LLM-generated Code. Our Finding 1 and Finding 2 reveal that errors made by different LLMs exhibit a wide range of different semantic and syntactic characteristics. Single-line logic errors such as *if errors* and *incorrect function arguments* may be relatively easy to fix, since existing automated program repair (APR) techniques are specialized to fix such errors [24]–[30]. However, our Finding 2 indicates that many errors made by code LLMs involve multiple lines of code, which require substantial edits to fix.

To overcome the limitations of traditional APR techniques, recent work has proposed to leverage LLMs to build program repair agents that are capable of synthesizing more complex patches [31]–[33]. These approaches typically rely on *coarse-grained* information such as error messages to guide the repair process. By contrast, our taxonomy provides a more detailed analysis of code generation errors based on their semantic and syntactic characteristics. We believe that leveraging this fine-grained information to repair errors in LLM-generated code is a promising direction. Specifically, one can first train a machine learning model to predict the types of errors. Then, the predicted errors can then be encoded in the prompt to guide the repair agent. For instance, instead of prompting the repair agent with "*fix the bug at Line 6*," one can prompt the agent with "*fix the incorrect function arguments at Line 6*," which may lead to a more accurate patch.

To demonstrate the effectiveness of repairing code generation errors with our labeled characteristics, we compared the performance of prompting GPT-4 to repair its incorrect code solutions with and without labeled semantic and syntactic characteristics (detailed prompts are available in our repository [8]). Since GPT-4 generated only 18 incorrect solutions on the HumanEval dataset, we included incorrect solutions generated by GPT-4 from an additional dataset, BigCodeBench-Hard [34], to increase the sample size. We selected BigCodeBench-Hard since it provides more challenging coding tasks compared to HumanEval and can better demonstrate the potential of fixing code generation errors using our taxonomy. In total, our sample set includes 18 incorrect solutions from HumanEval and 84 incorrect solutions from BigCodeBench-Hard. All these solutions were generated using the original prompt provided by each dataset and a decoding temperature of 0, which is consistent with the settings in Sec. II-C. Two authors independently labeled the error characteristics of these solutions using our taxonomy. The Fleiss' Kappa scores for semantic and syntactic characteristics were 0.91 and 0.93, indicating almost perfect agreement. All disagreements were resolved through discussion. Finally, we prompted GPT-4 to repair these incorrect solutions using the labeled error characteristics.

On HumanEval, GPT-4 was able to repair 7 out of 18 incorrect solutions by prompting with our labeled error characteristics, while it repaired only 4 out of 18 incorrect solutions without these characteristics. On the more challenging dataset, BigCodeBench-Hard, our method shows a bigger improvement over the baseline. Specifically, GPT-4 was able to repair 12 incorrect solutions with our labeled error characteristics while only 3 incorrect solutions without these characteristics. These results demonstrated that our taxonomy and labeled error characteristics are even more helpful when repairing incorrect solutions for challenging tasks.

Fault Localization for LLM-generated Code. Precisely locating the error location is an important first step for fixing code generation errors. As shown by Finding 2, code generation errors can occur in a variety of code constructs, which poses challenges to locating them precisely. Although many fault localization approaches have been proposed [21], [35]–[37], these methods may still fall short in locating errors in LLM-generated code. Traditional approaches, such as spectrum-based fault localization, rely heavily on high-quality test cases that comprehensively examine different execution paths in a program. However, it is effortful and time-consuming to design test cases, which limits the utility of these approaches. Recently, deep learning has also been applied to fault localization [36], [37], typically using features extracted from source code and error messages. For code generated by LLMs, there is richer information from different sources, such as logits, token probability distribution, and self-attention scores at each decoding step. It would be interesting to investigate whether such information, together with error messages and other code characteristics, can be leveraged to predict at which step the model may start generating incorrect code.

Estimating the Correctness of LLM-generated Code. Our Finding 4 shows that current LLMs continue to struggle with understanding and solving complex task requirements. This observation raises several interesting research questions. First, for a given code generation LLM, can we develop a method to estimate the upper bound of task complexity that this LLM can handle? In this study, we have only experimented with simple metrics such as prompt length and LOC of the correct solution. Further investigation on this question could help assess the capabilities and limitations of a code generation LLM more accurately, leading to more reliable and trustworthy code generation. Second, for a given programming task, can we use prompt characteristics, such as length, to predict the task's difficulty and the correctness of the generated code? Some recent work leverages LLMs as a judge for code correctness without the need of test cases [38], [39]. While they achieve promising accuracy on simple benchmarks (e.g., 73.13% by CodeJudge [39] on HumanEval), the accuracy on more challenging ones needs more improvement (e.g., 54.56% on BigCodeBench). Such techniques could help developers better allocate their code review and testing effort based on code correctness estimation.

V. Related Work

LLM-based Code Generation. LLMs have shown great potentials in code-related tasks, such as generation [1], [2], [40], [41], summarization [42]–[44], understanding [45], [46], search [47]–[49], and translation [50]. Recent work on code generation can then be roughly categorized into three groups: (1) developing high-quality training data [51]–[53], (2) developing better instruction fine-tuning techniques [54]–[56], and (3) developing better prompting strategies [57]–[65]. Compared to these efforts, our research focuses on analyzing the errors LLMs produce and deriving empirical insights to help develop new methods in the direction and contribute to better LLM-enabled intelligent software engineering.

Quality of LLM-generated Code. Evaluating the quality of LLM-generated code can reveal the current approach's shortcomings and guide future improvement. While prior research has delved into facets such as robustness [66], [67], security [68], [69], and attention alignment [70], studies focusing on the correctness of LLM-generated code [7], [71]–[74] are of particular relevance to our work.

Liu et al. [73] evaluated the quality of code generated by ChatGPT, addressing factors such as compilation/runtime errors and coding style. Different from their work, we investigated a broader range of LLMs and analyzed fine-grained error characteristics and their correlations with task complexity, test pass rates, etc. Pan et al. [75] introduced a taxonomy for LLM's code translation bugs. A key distinction is in the origin of errors, where Pan et al.'s work focuses on code-to-code translation, while our work focuses on NL-to-code generation. Our study also identifies distinct semantic error characteristics (e.g., *wrong (logical) direction*) and syntactic error characteristics (e.g., *missing/incorrect code block*).

Finally, Liu et al. [74] conducted a study of the quality of code generated by ChatGPT, assessing their correctness, understandability, and security. In their examination of code correctness, they primarily focused on compile errors and runtime crashes. Our research differs from theirs in two key respects: (1) Our study subjects are more diverse with both open-source and closed-source models; (2) our taxonomy also considers behavior deviations informed by test failures in addition to compiling errors and crashes. Based on the findings, we further provide actionable suggestions and implications for future enhancements in LLM-enabled code generation.

Taxonomy on Software Defects. Building a systematic taxonomy of software bugs can help stakeholders understand the pitfalls of target systems and provide guidance for better development practices. One of the early attempts was the orthogonal defect classification proposed by IBM Research [76], [77]. Since then, numerous endeavors have been made to construct defect taxonomies targeting different programming languages [78]–[86] or different applications [87], [88]. Unlike previous attempts that focused on bugs in real-world software projects, our work focuses on LLM-generated code. Our findings reveal that a large portion of code errors made by LLMs exhibit complex semantic characteristics rather than subtle errors usually introduced by human programmers.

VI. Threats to Validity

Threats to Internal Validity. One potential threat comes from our manual analysis process, where labelers may have different opinions and sometimes may make mistakes. To mitigate this, four of the authors first performed open coding and iteratively refined our codebook until a substantial agreement was achieved before two of the authors labeled the complete dataset. Our final Fleiss' Kappa regarding the semantic and syntactic characteristics are 0.91 and 0.91, respectively, indicating almost perfect agreement [17].

Threats to External Validity. One potential threat lies in the choice of dataset. Considering the extensive labeling effort (e.g., running the code and performing step-by-step debugging to locate the root cause), we have only labeled one dataset. In future work, one may consider labeling more datasets to confirm the generalizability of our findings. Nevertheless, given the size of our labeled dataset (557 code snippets), we believe our findings can be generalized to other similar datasets [34], [89], [90]. Additionally, our study has only explored errors from function-level code generation tasks. There are many other code generation tasks (e.g., class-level code generation [91]), as well as other code-related tasks such as code summarization and refactoring. In future work, it is interesting to investigate whether these tasks share similar error characteristics with function-level code generation errors.

Moreover, our study only covers Python programs, which might not generalize to programming languages that are very different from Python, such as PHP and Rust. We chose Python because it is one of the most popular object-oriented programming languages. In future work, we plan to expand our study with programming tasks from other languages.

Finally, we have only experimented with six LLMs and one prompting strategy in this study due to the intensive labeling effort (i.e., 328 person-hours for labeling 557 incorrect code snippets). Given that code generation with LLMs is a fast-developing research field, we plan to expand our study by labeling errors from more recent code LLMs, e.g., MagiCoder [56], CodeLlama [54], etc. Additionally, our findings may not generalize to code generation errors with different prompting strategies, e.g., chain-of-thought [92], self-debugging [63], etc. It is worthwhile to investigate code generation errors with more advanced prompting strategies in future work.

Threats to Construct Validity. In RQ3, we use the length of the task description and the length of the correct solution to estimate the task complexity. Although they may not be the best metrics for such estimation, we believe this is a pragmatic choice since there is no commonly used metric for measuring task complexity for LLM's code generation. In future work, researchers may consider designing new metrics to estimate the task complexity and investigate its correlation with LLM's code generation capabilities.

VII. Conclusion

This paper presents an empirical study on code generation errors made by LLMs. We first derived a taxonomy of LLMs' code generation errors based on six popular LLMs' failure cases on the HumanEval dataset [7] through open coding and thematic analysis. We labeled a total of 557 errors committed by these LLMs according to the taxonomy. We found that these LLMs exhibited different distributions of semantic and syntactic error characteristics. We further analyzed the bug-fixing effort, the impact of task complexity, and the correlation between test pass rates and different kinds of errors. In the end, we discussed the implications of our study and propose future research opportunities for improving the quality and reliability of code LLMs.

Acknowledgment

This work was supported in part by Canada CIFAR AI Chairs Program, Natural Sciences and Engineering Research Council of Canada, JST CRONOS Grant (No.JPMJCS24K8), JSPS KAKENHI Grant (No.JP21H04877, No.JP23H03372, and No.JP24K02920), and the Autoware Foundation. This work was also supported in part by NSF CCF-2340408.

References

[1] E. Nijkamp, B. Pang, H. Hayashi *et al.*, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2022.

[2] D. Fried, A. Aghajanyan *et al.*, "Incoder: A generative model for code infilling and synthesis," in *The Eleventh International Conference on Learning Representations*, 2022.

[3] R. Li, L. B. Allal *et al.*, "Starcoder: may the source be with you!" *Transactions on Machine Learning Research*, 2023.

[4] OpenAI, "Gpt-4 technical report," 2023.

[5] "Chatgpt," http://chat.openai.com, 2023.

[6] L. B. Allal, R. Li *et al.*, "Santacoder: don't reach for the stars!" *arXiv preprint arXiv:2301.03988*, 2023.

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

[8] Z. Wang, Z. Zhou, D. Song *et al.*, "GitHub Repository of The Paper: Towards Understanding Error Characteristics of Code Generated by LLMs." https://github.com/ma-labo/defects4codellm, 2025.

[9] P. Vaithilingam, T. Zhang, and E. L. Glassman, "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*, 2022, pp. 1–7.

[10] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.

[11] D. Kocetkov, R. Li, L. B. Allal *et al.*, "The stack: 3 tb of permissively licensed source code," *arXiv preprint arXiv:2211.15533*, 2022.

[12] A. L. Strauss and J. Corbin, "Open coding," *Social research methods: A reader*, pp. 303–306, 2004.

[13] A. Antoine, S. Malacria, N. Marquardt, and G. Casiez, "Interaction illustration taxonomy: Classification of styles and techniques for visually representing interaction scenarios," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–22.

[14] J. Lazar, J. H. Feng, and H. Hochheiser, *Research methods in human-computer interaction*. Morgan Kaufmann, 2017.

[15] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[16] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[17] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[18] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[19] P. Jaccard, "Étude comparative de la distribution florale dans une portion des alpes et des jura," *Bull Soc Vaudoise Sci Nat*, vol. 37, pp. 547–579, 1901.

[20] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[21] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S.-C. Cheung, "Historical spectrum based fault localization," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2348–2368, 2019.

[22] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "Codebertscore: Evaluating code generation with pretrained models of code," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 13 921–13 937.

[23] S. Saha *et al.*, "Harnessing evolution for multi-hunk program repair," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 13–24.

[24] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 416–426.

[25] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[26] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[27] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 298–309.

[28] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 31–42.

[29] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 511–523.

[30] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[31] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481.

[32] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," *arXiv preprint arXiv:2404.05427*, 2024.

[33] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying llm-based software engineering agents," *arXiv preprint arXiv:2407.01489*, 2024.

[34] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul *et al.*, "Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions," *arXiv preprint arXiv:2406.15877*, 2024.

[35] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "Spectrum-based multiple fault localization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 88–99.

[36] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 169–180.

[37] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 661–673.

[38] T. Y. Zhuo, "Ice-score: Instructing large language models to evaluate code," in *Findings of the Association for Computational Linguistics: EACL 2024*, 2024, pp. 2232–2242.

[39] W. Tong and T. Zhang, "Codejudge: Evaluating code generation with large language models," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 20 032–20 051.

[40] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "CodeT5+: Open code large language models for code understanding and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Singapore, Dec. 2023, pp. 1069–1088.

[41] Q. Zhu, Q. Liang, Z. Sun, Y. Xiong, L. Zhang, and S. Cheng, "Grammart5: Grammar-integrated pretrained encoder-decoder neural model for code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[42] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4998–5007.

[43] C. Niu, C. Li *et al.*, "Spt-code: Sequence-to-sequence pre-training for learning source code representations," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 2006–2018.

[44] Y. Cai, Y. Lin, C. Liu *et al.*, "On-the-fly adapting code summarization on trainable cost-effective language models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[45] D. Wang *et al.*, "Bridging pre-trained models and downstream tasks for source code understanding," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 287–298.

[46] Y. Ding, B. Steenhoek, K. Pei, G. Kaiser, W. Le, and B. Ray, "Traced: Execution-aware pre-training for source code," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[47] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, Online, Nov. 2020, pp. 1536–1547.

[48] Z. Zhang *et al.*, "Diet code is healthy: Simplifying programs for pre-trained models of code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1073–1084.

[49] N. Jain *et al.*, "ContraCLM: Contrastive learning for causal language model," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, Toronto, Canada, Jul. 2023, pp. 6436–6459.

[50] B. Roziere *et al.*, "Unsupervised translation of programming languages," *Advances in neural information processing systems*, vol. 33, pp. 20 601–20 611, 2020.

[51] A. Lozhkov, R. Li *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[52] D. Guo, Q. Zhu, D. Yang *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[53] Y. Ding, J. Peng, M. J. Min, G. Kaiser, J. Yang, and B. Ray, "Semcoder: Training code language models with comprehensive semantics," *arXiv preprint arXiv:2406.01006*, 2024.

[54] B. Roziere, J. Gehring *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[55] Z. Luo, C. Xu *et al.*, "Wizardcoder: Empowering code large language models with evol-instruct," in *The Twelfth International Conference on Learning Representations*, 2024.

[56] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," 2023.

[57] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin, "Self-edit: Fault-aware code editor for code generation," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Toronto, Canada, Jul. 2023, pp. 769–787.

[58] R. Bairi, A. Sonwane *et al.*, "Codeplan: Repository-level coding using llms and planning," in *Proceedings of the 32nd ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2024.

[59] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, "What makes good in-context demonstrations for code intelligence tasks with llms?" in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 761–773.

[60] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," in *The Eleventh International Conference on Learning Representations*, 2023.

[61] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, "Is self-repair a silver bullet for code generation?" in *The Twelfth International Conference on Learning Representations*, 2023.

[62] J. Li *et al.*, "Skcoder: A sketch-based approach for automatic code generation," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2124–2135.

[63] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," in *The Twelfth International Conference on Learning Representations*, 2024.

[64] Y. Ding, M. J. Min, G. Kaiser, and B. Ray, "Cycle: Learning to self-refine the code generation," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 392–418, 2024.

[65] A. Madaan *et al.*, "Self-refine: Iterative refinement with self-feedback," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[66] S. Arakelyan, R. Das, Y. Mao, and X. Ren, "Exploring distributional shifts in large language models for code analysis," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 16 298–16 314.

[67] Y. Liu, C. Tantithamthavorn, Y. Liu, and L. Li, "On the reliability and explainability of automated code generation approaches," *ACM Transactions on Software Engineering and Methodology*, 2024.

[68] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *IEEE Symposium on Security and Privacy*, 2022, pp. 754–768.

[69] B. Steenhoek, M. M. Rahman *et al.*, "A comprehensive study of the capabilities of large language models for vulnerability detection," *arXiv preprint arXiv:2403.17218*, 2024.

[70] B. Kou, S. Chen, Z. Wang, L. Ma, and T. Zhang, "Do large language models pay similar attention like human programmers when generating code?" *PACMSE*, vol. FSE, no. 100, 2024.

[71] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[72] K. Jesse, T. Ahmed, P. T. Devanbu, and E. Morgan, "Large language models and simple, stupid bugs," in *IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 563–575.

[73] Y. Liu, T. Le-Cong *et al.*, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, 2023.

[74] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No need to lift a finger anymore? assessing the quality of code generation by chatgpt," *IEEE Transactions on Software Engineering*, 2024.

[75] R. Pan, A. R. Ibrahimzada *et al.*, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[76] R. Chillarege, W.-L. Kao, and R. G. Condit, "Defect type and its impact on the growth curve." in *ICSE*, vol. 91, 1991, pp. 246–255.

[77] R. Chillarege, I. S. Bhandari *et al.*, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on software Engineering*, vol. 18, no. 11, pp. 943–956, 1992.

[78] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[79] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 180–182.

[80] Q. Hanam, F. S. d. M. Brito, and A. Mesbah, "Discovering bug patterns in javascript," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 144–156.

[81] P. Gyimesi, B. Vancsics, A. Stocco *et al.*, "Bugsjs: a benchmark of javascript bugs," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 90–101.

[82] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, pp. 286–315, 2009.

[83] X. Zhang, R. Yan, J. Yan, B. Cui, J. Yan, and J. Zhang, "Excepy: A python benchmark for bugs with python built-in types," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 856–866.

[84] A. Rahman, D. B. Bose, R. Shakya, and R. Pandita, "Come for syntax, stay for speed, understand defects: an empirical study of defects in julia programs," *Empirical Software Engineering*, vol. 28, no. 4, p. 93, 2023.

[85] D. Lin *et al.*, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, 2017, pp. 55–56.

[86] P. S. Kochhar, D. Wijedasa, and D. Lo, "A large scale study of multiple programming languages and code quality," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 563–573.

[87] A. Rahman, E. Farhana, C. Parnin, and L. Williams, "Gang of eight: A defect taxonomy for infrastructure as code scripts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 752–764.

[88] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.

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

[90] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with apps," *NeurIPS*, 2021.

[91] X. Du, M. Liu, K. Wang *et al.*, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[92] J. Wei, X. Wang, D. Schuurmans *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.