RealisticCodeBench: Towards More Realistic Evaluation of Large Language Models for Code Generation

Xiao Yu[1], Haoxuan Chen[2], Lei Liu[3], Xing Hu[1], Jacky Wai Keung[4], and Xin Xia[1*]
[1]The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China,
xiao.yu@zju.edu.cn, xinghu@zju.edu.cn, xin.xia@acm.org
[2]School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Wuhan, China,
haoxuan.chen@whut.edu.cn
[3]Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, China, lei.liu@stu.xjtu.edu.cn
[4]Department of Computer Science, City University of Hong Kong, Hong Kong, China, jacky.keung@cityu.edu.hk

***Abstract*—Evaluating the code generation capabilities of Large Language Models (LLMs) remains an open question. Recently, more advanced benchmarks—such as CoderEval, EvoCodeBench, and ClassEval—have been introduced to evaluate LLMs on practical coding tasks from GitHub repositories, such as non-standalone function generation and class-level code generation. However, even the most sophisticated LLMs struggle with these complex tasks; for instance, GPT-4 achieves only a 37.0% pass@1 on ClassEval. Prior studies show that developers often discard LLM-generated code or abandon code generation models when outputs are incorrect or require extensive debugging, which leads them to rely on LLMs primarily for code generation tasks that high-performing models can reliably handle.**

In response to this gap, we introduce RealisticCodeBench, a benchmark specifically designed to reflect the types of problems developers commonly tackle with LLMs. By mining GitHub repositories for code samples tagged as generated by ChatGPT or Copilot, we collect real-world coding tasks that capture typical LLM usage scenarios. We modify these tasks, generate reference solutions and test cases, and adapt the problems into multiple programming languages. This effort results in RealisticCodeBench, comprising a total of 376 programming problems translated across multiple languages: 361 in Python, 346 in JavaScript, 343 in TypeScript, 307 in Java, and 323 in C++, each with corresponding reference solutions and test cases. We evaluate 12 general-purpose and code-specific LLMs on RealisticCodeBench. Our findings reveal that GPT-4.1 achieves the highest average pass@1 score across languages, closely followed by DeepSeek-V3-671B, suggesting that DeepSeek-V3-671B provides a viable open-source alternative to GPT-4.1 for large companies with sufficient GPU resources and privacy concerns. CodeGeeX4-9B, a cost-effective model, emerges as a suitable substitute for GPT-4o-mini for individual developers and smaller organizations with similar privacy considerations. Additionally, LLM performance discrepancies between HumanEval and RealisticCodeBench suggest that some LLMs are either overly specialized for HumanEval-style problems or insufficiently optimized for real-world coding challenges. Finally, we analyze failed cases, summarize common LLM limitations, and provide implications for researchers and practitioners.

***Index Terms*—Code Generation, Large Language Model, Benchmark, GitHub**

I. Introduction

Code generation, which automatically creates code snippets from natural language descriptions, has been widely adopted to enhance development efficiency and productivity, attracting significant attention in academic research [1], [2], [3], [4]. Recent advances in Large Language Models (LLMs)—trained on massive volumes of both general and code-specific datasets—have further accelerated progress in this field [5]. To evaluate the performance of these emerging LLMs on code generation tasks, several benchmarks have been introduced, starting with HumanEval [6] and MBPP [7]. Reporting performance on these benchmarks has seemingly become mandatory for a model to be considered competitive in code generation [8]. Indeed, nearly all new LLMs released in 2023-2025 highlight code generation results on one or both of these benchmarks. While they have been widely used and provide valuable insights, the programming problems they contain are largely algorithmic and basic programming problems, which do not fully reflect the challenges of real-world coding [9]. To address this, more complex benchmarks—such as CoderEval [10], EvoCodeBench [11], ComplexCodeEval [12], and ClassEval [13]—have been developed to assess LLM performance on more challenging, practical coding tasks collected from real-world GitHub code repositories, such as non-standalone function generation and class-level code generation. These benchmarks offer a deeper understanding of the upper limits of LLM capabilities when tackling intricate programming problems.

However, developers currently tend not to rely on LLMs for overly complex coding tasks, primarily due to the low success rates of LLMs on more challenging benchmarks. For example, GPT-3.5 achieves only a 21% pass@1 rate for non-standalone function generation on CoderEval [10], while GPT-4 reaches just a 37.0% pass@1 rate for class-level code generation on ClassEval [13], which can discourage developers from using LLMs for such sophisticated code generation tasks. A large-scale survey conducted by Liang et al. [14] found

[*]Corresponding author: Xin Xia, xin.xia@acm.org

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00248

that developers often discard LLM-generated code or abandon the use of code generation models when they fail to meet functional or non-functional requirements, when developers struggle to control the models to produce the desired output, or when significant effort is needed to debug and refine the LLM-generated code. In other words, while developers often work on complex programming problems like those in CoderEval [10], EvoCodeBench [11], ComplexCodeEval [12], and ClassEval [13], current LLMs are not yet ready to generate such sophisticated code at scale. Instead, developers are more likely to use LLMs for more manageable coding tasks that high-performing models (e.g., GPT-4.1) can generate correctly without requiring extensive debugging or modification. Therefore, to better align benchmarks with current developer practices of using LLMs for code generation, we need to shift our focus toward understanding the types of code developers are actually generating with LLMs daily and create benchmarks based on these practical use cases.

To achieve this, we collect real-world coding tasks that reflect typical LLM code generation scenarios by mining high-star GitHub repositories for code samples explicitly labeled as generated by ChatGPT or Copilot. Specifically, our previous study [15] find that nearly all LLM-generated code on GitHub is produced by tools like ChatGPT or Copilot, with very few samples from other LLMs. Developers frequently annotate such code snippets with comments like "the code is generated by ChatGPT," indicating they are created using these tools. Using search terms like "generated by ChatGPT", we leverage the GitHub REST API to locate and collect relevant Python, Java, JavaScript, TypeScript, and C++ code samples from high-star projects, which represent how developers use LLMs for code generation in real-world scenarios. After collecting the samples, we carefully filter out overly simplistic, repetitive, or difficult-to-test codes.

We then make modifications to each sample's requirements while preserving the original intent and complexity as much as possible. Where applicable, we also adjust the number and types of input and output parameters to further mitigate data leakage risks. Using ChatGPT-4o, we generate reference solutions for each modified programming problem, followed by manual corrections. ChatGPT-4o also creates multiple test cases based on the problem descriptions and reference solutions, which are refined manually to ensure accuracy and adequate line and branch coverage. Next, we use ChatGPT-4o to generate multi-language versions of each programming problem, followed by manual validation of the accuracy of the translated solutions, test cases, and coverage. It is important to note that some programming problems do not translate directly across languages due to language-specific data types or operations. In such cases, we retain the problems as language-specific to reflect real-world development practices. Finally, we invite 13 experienced engineers to assess whether the programming problems, including their multi-language versions, represent realistic development scenarios and if proprietary developers would also likely use LLMs to solve them. Only problems approved by a majority (at least 10 out of 13 engineers) are retained. Ultimately, we construct our benchmark, RealisticCodeBench, comprising 376 programming problems translated across multiple languages: 361 in Python, 346 in JavaScript, 343 in TypeScript, 307 in Java, and 323 in C++. Each problem includes corresponding reference solutions and test cases, spanning 9 distinct domains such as data structures and algorithms, text processing, file handling, data visualization and graphic applications, network programming, and frontend development. This provides a comprehensive assessment of LLM capabilities on coding challenges that developers currently address with LLM assistance.

Based on RealisticCodeBench, we conduct extensive experiments on 12 general-purpose and code-specific models commonly studied in recent benchmarks, such as GPT-4.1, GPT-4o-mini, DeepSeek-V3-671B, Llama 3.1-8B, CodeGeeX4-9B, DeepSeek-Coder-6.7B, CodeLlama-7B, and StarCoder2-7B. Across five programming languages, GPT-4.1 achieves the highest average pass@1 score at 60.65%, with DeepSeek-V3-671B close behind at 58.86%. This suggests that companies with sufficient resources and privacy concerns could consider deploying DeepSeek-V3-671B as an open-source alternative to GPT-4.1 for everyday coding tasks. CodeGeeX4-9B achieves an average pass@1 score of 45.75%, compared to GPT-4o-mini's 53.11%, showing only a moderate gap between them. Thus, individual developers and smaller organizations with similar privacy concerns can deploy CodeGeeX4-9B as an affordable substitute for GPT-4o-mini, using a setup with two NVIDIA GeForce RTX 3090 (24GB) GPUs (approximately $3,000) to balance privacy, cost, and code generation performance. Furthermore, we observe substantial performance discrepancies of some LLMs between HumanEval and RealisticCodeBench. While models like CodeGeeX4-9B reach impressive pass@1 scores on HumanEval (82.3%) and DeepSeek-Coder-6.7B scores 78.6%, their performance drops substantially on RealisticCodeBench's Python subset (54.02% and 45.15%, respectively). This suggests that current LLMs may either be overly specialized for HumanEval-style problems or lack optimization for practical coding tasks. Finally, by analyzing failed cases, we identify critical areas where LLMs fall short in RealisticCodeBench, offering insights into potential improvements for practical code generation capabilities.

In summary, our contributions are as follows:

(1) We propose RealisticCodeBench, a benchmark that aligns with the types of coding problems developers typically solve with LLMs in practical development settings. Our benchmark is available in [16].

(2) We systematically benchmark 12 LLMs' code generation capabilities using RealisticCodeBench. Based on the results, we provide implications for researchers and practitioners.

II. Background and Related Work

A. LLMs for Code Generation

Code generation involves creating code snippets based on given natural language requirements. General LLMs are typically trained on a combination of general textual data, code corpora, and instructions. Among the most well-known general LLMs are GPT-4 [17] and GPT-3.5 [18], both of which have demonstrated significant capabilities across a wide range of tasks. Additionally, other general-purpose models like DeepSeek-V3 [19], Llama 3.1 [20], Phi-3 [21], Mistral [22], and ChatGLM [23] have gained attention for their capabilities. Technical reports for these models often emphasize their strengths not only in general natural language processing tasks but also their promising potential in code generation.

On the other hand, specialized code LLMs are primarily trained on large-scale code-specific datasets with tailored instructions, often outperforming general-purpose LLMs in code generation tasks. Notable examples include CodeGen [24], StarCoder [25], CodeLlama [26], DeepSeek-Coder [27], and CodeGeeX [28]. For instance, DeepSeek-Coder is trained from scratch on 2 trillion tokens, with a composition of 87% code and 13% natural languages in both English and Chinese. StarCoder2 is trained on 17 programming languages from the Stack v2 [25]. These models are designed to focus more specifically on understanding and generating code, typically demonstrating superior performance in handling code-related tasks compared to general LLMs.

B. Code Generation Benchmarks

Literature Search: To understand the progress of code generation benchmarks, we conduct a literature search covering publications from 2021 to 2025 by using a forward snowballing approach [35] [1]. The starting year of 2021 is selected, as it marks the publication of the earliest prominent benchmarks for code generation, which include test cases for evaluating LLMs' code generation accuracy (i.e., APPS [36], HumanEval [6], and MBPP [7]). Although earlier code generation benchmarks, such as Concode [37] and JuICe [38], were proposed before 2021, they mainly focused on evaluating deep learning models, like LSTM and Transformer, rather than LLMs. Moreover, these datasets lacked test cases, relying instead on metrics like exact accuracy and BLEU to compare model performance. Consequently, they are rarely used in later research evaluating LLMs for code generation.

Therefore, our search process begins by gathering all papers that cite APPS [36], HumanEval [6], and MBPP [7] using Google Scholar. We then filter these citations to identify papers proposing new benchmarks or significantly extending existing ones in the context of code generation, considering only studies written in English with full text available. We exclude papers that introduce benchmarks for unrelated fields (e.g., program repair, code completion, or code translation) and focus solely on those proposing code generation benchmarks. For each selected paper, we recursively examine its citations, focusing on new or updated benchmarks developed. This process continues until no further relevant papers are found, ensuring that no significant benchmark developments are missed during the search. Finally, the overall search process results in 57 code generation benchmarks. The identified benchmarks can be broadly classified into three categories. The first category, comprising 25 papers [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], focuses on domain-specific code generation abilities, such as generating security code [46], [49], [61], VHDL code [53], bioinformatics code [47], Verilog code [43], data science code [7], [40], [41], [59], AI code [48], object-oriented code in Java [50], parallel code [54], Infrastructure-as-Code (IaC) programs [56], web design [60], etc. The second category, comprising 21 papers focusing on non-realistic code generation development scenarios [6], [7], [64], [65], [66], [28], [67], [68], [69], [8], [36], [70], [71], [72], [10], [73], [74], [75], [76], [77], [78], includes one subset of benchmarks such as HumanEval [6], MBPP [7], or their modified versions [64], [65], [66], [28], [68], [69], [33], [64], [65], [33] that focus on pure-method algorithm or logic tasks and often exhibit exceptionally high performance on state-of-the-art models. The other subset [74], [76], [72], [73], [75], such as LiveCodeBench [74] and CodeElo [76], focus on algorithmic tasks for competitive programming, but these competitive programming problems are rarely encountered in real-world development scenarios. The third category, which includes 11 papers [29], [12], [34], [11], [10], [13], [30], [32], [31], [33], [9], focuses on evaluating general code generation capabilities that reflect real-world development scenarios, which aligns with the goals of our benchmark. Due to space constraints, we only discuss the difference between these benchmarks and our RealisticCodeBench.

Table I overviews the 11 benchmarks, including details such as the year of introduction, target programming language, the source of programming problems, target code granularity, the number of programming problems ("#Tasks"), average lines of code ("#LOC") in reference solutions, average token lengths of the task prompt (usually the requirements and function signature) ("#Tokens"), and the best model performance (usually GPT-4) in pass@1. In the table, "_" indicates that the corresponding information was not provided in the benchmark paper. Among them, the two benchmarks without a pass@1 values, the MCoNaLa benchmark [29] focuses solely on statement-level code generation scenarios collected from Stack Overflow. In contrast, ComplexCodeEval [12] includes function-level tasks sourced from real and complex development environments in GitHub repositories. However, it lacks test cases to accurately assess the generated code.

For benchmarks involving complex, non-standalone functions and classes [34], [11], [10], [13], [30], [32], low pass@1 scores are mainly due to the intricate dependencies inherent to these tasks. For example, EvoCodeBench [11], DevEval [30], and HumanEvo [34] focus on complex function dependencies or repository-level dependencies, resulting in pass@1 scores of 20.7%, 53.0%, and 34.5% on GPT-4, respectively. Similarly, ClassEval [13], a benchmark of 100 manually created Python problems that simulate real-world class generation scenarios, yielded a 37.0% pass@1 score on GPT-4. In addition, two benchmarks were created from open source data. Paul et al. [31] developed ScenEval, collecting various statements, methods, and classes from open source platforms

[1] This literature review was conducted in April 2025.

TABLE I: The current general code generation benchmarks that reflect the real development scenarios

Benchmark	**Year**	**Language**	**Source**	**Granularity**	**#Tasks**	**#LOC**	**#Tokens**	**Pass@1**
MCoNaLa [29]	2023	Python	Conala	Statement	896	1	27.6	-
CoderEval [10]	2024	Python, Java	GitHub	Function	230	30	108.2	21.0 % (GPT-3.5)
EvoCodeBench [11]	2024	Python	GitHub	Function, Repository	275	-	-	20.7 % (GPT-4)
ClassEval [13]	2024	Python	Manual	Class	100	45.7	123.7	37.0 % (GPT-4)
DevEval [30]	2024	Python	PyPI	Function, Repository	1874	-	-	53.0 % (GPT-4)
NCB [9]	2024	Python, Java	Online Services	Function	402	-	-	52.8 % (GPT-4)
ScenEval [31]	2024	Java	W3Resources, Stack Overflow, Textbooks	Statement, Function, Class	12864	1-50	-	75.6 % (ChatGPT)
BigCodeBench [32]	2024	Python	GitHub, Huagging face, Croissant	Function	1140	10	-	51.1 % (GPT-4o)
ComplexCodeEval [12]	2024	Python, Java	PyPI, GitHub	Function	11081	35.9	278.8	-
RACE [33]	2024	Python	HumanEval+, MBPP+, ClassEval, LeetCode	Class, Function	923	-	-	70.1 % (GPT-4-o1-mini)
HumanEvo [34]	2025	Python, Java	PyPI, GitHub	Function, Repository	400	-	-	34.5 % (GPT-4)
RealisticCodeBench	**2025**	**Multilingual**	**GitHub**	**Function, Class**	**376**	**42.2**	**124.4**	74.52% (GPT-4.1)

like W3Resources, Stack Overflow, and textbooks to cover a wide range of scenarios. ChatGPT achieved a pass@1 of 75.6% on this relatively simple benchmark. Zheng et al. [33] combined datasets such as HumanEval, MBPP, ClassEval, and LeetCode to form RACE, a moderate complexity dataset where GPT-4o-mini and Claude-3.5-Sonnet achieved pass@1 rates of 70.1% and 62.3%, respectively.

In particular, the NCB benchmark [9] shares similarities with our benchmark, containing 402 high-quality Python and Java problems carefully selected from natural user queries on the CodeGeeX online coding platform. However, NCB's query problems are not necessarily solvable by LLMs, with GPT-4 achieving a pass@1 of 52.8%. In contrast, our benchmark includes only LLMs-solvable problems, with developers accepting and uploading these LLM-generated solutions to GitHub. By collecting LLM-generated code from GitHub, our benchmark more accurately reflects scenarios where developers use LLMs in real-world coding tasks. To better align benchmarks with current LLM usage practices, we, therefore, introduce RealisticCodeBench, a benchmark designed to reflect the types of problem developers commonly tackle with LLMs.

III. RealisticCodeBench

Figure 1 outlines RealisticCodeBench's construction process. The pipeline consists of two primary steps: 1) collecting and filtering high-quality code generated by ChatGPT/Copilot from GitHub (Section III-A), and 2) constructing the benchmark using a semi-automated pipeline supported by ChatGPT-4o in a conversational window format, which includes adapting problem requirements, writing reference solutions and test cases, and generating multi-language versions of each programming problem (Section III-B). The entire process of constructing the benchmark, which includes 376 programming problems across various languages, requires approximately 700 person-hours to complete.

A. Data Collection

ChatGPT/Copilot-Generated Code Collection. Our previous study [15] find that nearly all code samples generated by LLMs on GitHub are created using tools like ChatGPT or Copilot, with very few produced by other LLMs. Developers often annotate their code with comments such as "*the code is generated by ChatGPT/Copilot*" to indicate its origin. These annotations typically follow the format x+y+z, where x is a verb from {generated, written, created, implemented, authored, coded}, y is a preposition from {by, through, using, via, with}, and z is a tool identifier from {ChatGPT, Copilot, GPT-3, GPT-4}. Following their approach, we use these triplets x+y+z, such as "*generated by ChatGPT*" to locate and collect relevant code snippets via the GitHub REST API. We specifically focus on code written in Python, Java, JavaScript, TypeScript, and C++, as these languages not only dominate the landscape of LLM-generated code on GitHub but are also widely used across various real-world development domains. To ensure the quality of collected samples, we first prioritize repositories with high star ratings to source reputable code. However, many projects containing ChatGPT-generated code are new and have not accumulated sufficient stars; thus, we also evaluate the detail level of README documents and the scale of code volume. Projects with clear documentation standards, complete code logic, and practical technical reference value are also included.

Suitable Programming Problems Filtering. Although we initially collect over 2,100 ChatGPT/Copilot-generated code samples from GitHub, not all are suitable for inclusion in our benchmark. We first manually filter out overly simplistic code—specifically code with very few lines. This simplicity is evaluated relative to each language's practical context (e.g., code that merely calculates the Euclidean distance between two points, which can be implemented in a single line of Python and thus is filtered out). Additionally, we exclude samples whose solutions are difficult to test (e.g., those involving frontend-backend interactions or dependencies on external data or upstream logic). Finally, we review the remaining samples to remove overly similar tasks (e.g., multiple samples that validate if a string is a valid email address), ensuring the benchmark contains a diverse set of programming problems. After this filtering process, we obtain 172 refined Python code samples, 26 refined Java samples, 75 refined JavaScript samples, 57 refined TypeScript samples, and 46 refined C++ samples. The GitHub links to these code samples, along with detailed information about the GitHub repositories hosting them (e.g., star counts, fork counts, contributor numbers, and commit counts), are available in [16].

B. Benchmark Construction

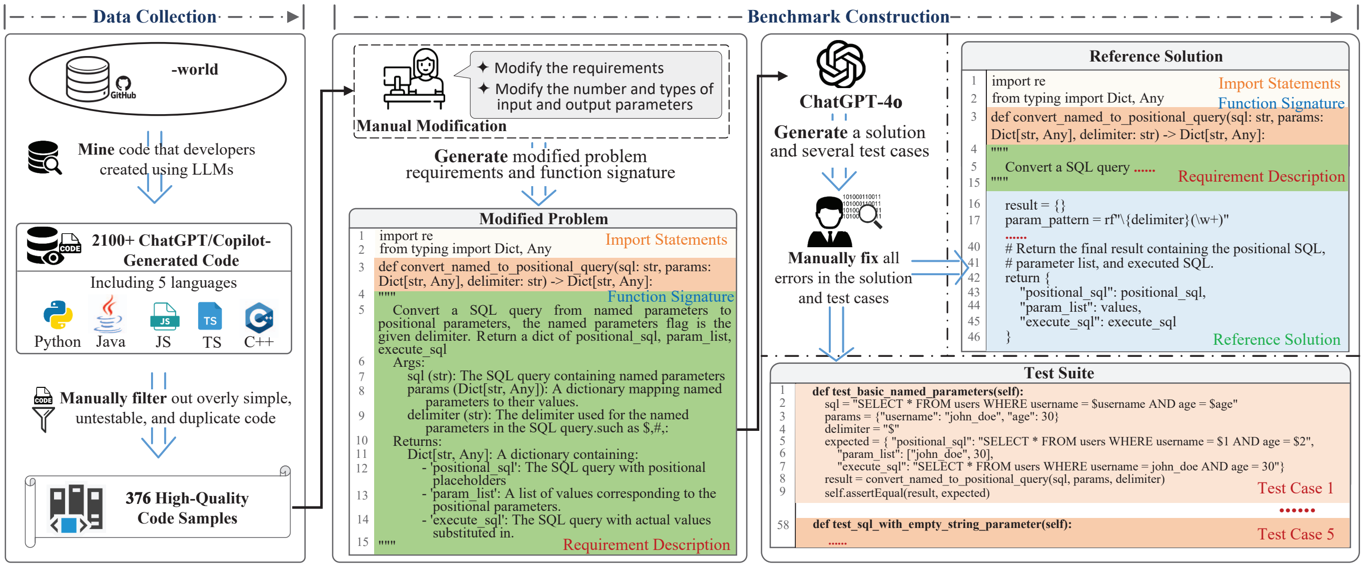
Once we have collected ChatGPT/Copilot-generated code samples from GitHub, we move forward with constructing our benchmark. As shown in Figure 1, each programming problem in RealisticCodeBench includes an input description (comprising the function signature and requirement description). Additionally, the benchmark contains a reference solution for each programming problem, which serves as a reference implementation, along with a test suite to verify the correctness of the generated code. Typically, LLMs generate code snippets based on the input descriptions, and the correctness of these snippets is validated using the provided test suite.

Modification of Programming Problems. Since most of the original code samples only indicate that they are generated by ChatGPT or Copilot without describing their functionality, we first leverage ChatGPT-4o's advanced capabilities in code comment generation [79] to produce concise summaries for each code sample. This is solely intended to help us clearly understand the code's core functionality, facilitating the modification of programming problems. Data leakage is a concern because many LLMs are pre-trained on code from GitHub, which can lead to inadvertent memorization of specific content [80], [81]. Consequently, these models may solve programming tasks by recalling solutions they encountered during pre-training. To mitigate this risk, we apply substantial modifications to the requirements of the original code samples, while striving to preserve the code's original intent and task complexity. Additionally, we modify the number and types of input and output parameters where feasible. In the adapted function signatures—consistent with mainstream benchmarks like HumanEval, which outline implementation requirements for LLMs. These requirements include only basic specifications: they omit specific steps for task solutions, avoid prelisting boundary conditions, and instead clarify the function's objectives, input parameters, and return value constraints for each programming language. For instance, one GitHub project with over 30 stars includes a method that converts a SQL string with named parameters (e.g., $variable) to a format compatible with asyncpg (using $1, $2, etc.) and returns the new SQL string and the list of values in the correct order [82]. The input parameters are defined as *sql* (the original SQL string with named parameters) and *params* (a dictionary of parameters), while the output is a tuple (*new_sql_string*, *list_of_values*). In our modified requirement (as shown in Figure 1), we specify: *Convert a SQL query from named parameters to positional parameters, the named parameters flag is the given delimiter. Return a dictionary of positional_sql, param_list, execute_sql*, increasing inputs to three and changing output to a dictionary.

Reference Solution Generation. We then use ChatGPT-4o to generate solutions for each adapted programming problem by providing the problem description (including the function signature and requirement description) as prompts. To effectively prevent potential data leakage risks that may occur during the process of generating solutions, we have defaulted to enabling the chat history closure function provided by OpenAI [2] when using ChatGPT-4o. When chat history is disabled, the subsequent conversation content initiated will not be used for model training and optimization. Although ChatGPT-4o is a highly capable tool, it can still produce incorrect code during generation. Therefore, each solution is meticulously reviewed by three programmers, each with over four years of coding experience, to ensure accuracy. If any bugs are identified by one of the programmers, they revise the code to correct the errors. The revised version is then reviewed by the other two programmers to confirm that the corrections are accurate, ensuring that the reference solutions are both reliable and error-free. These reference solutions are not used directly as evaluation benchmarks but are included to support the development of test cases and facilitate future research efforts.

Test Case Generation. We also utilize ChatGPT-4o to generate high-quality test cases for each adapted programming problem. The prompt starts with the instruction: "Please create test cases for this programming problem and the reference solution. Ensure that the test cases cover a wide range of inputs, including typical use cases, edge cases, corner cases, and invalid inputs." Following this, the prompt includes the problem description and reference solution. After the test cases are generated, the same three programmers review and correct any issues related to formatting or outputs. If an error is identified, the programmer revises the test cases. The updated cases are then reviewed by the other two programmers to validate corrections. Once this process is complete, the line and branch coverage for each function is reassessed. We use PyTest for Python, JUnit for Java, JavaScript, and TypeScript, and Catch2 for C++ to calculate this coverage. If coverage is still below 100%, one of the programmers manually writes

[2] https://openai.com/index/new-ways-to-manage-your-data-in-chatgpt/


Data Collection
-world
GitHub
Mine code that developers created using LLMs
CODE
2100+ ChatGPT/Copilot-Generated Code
Including 5 languages
JS
TS
Python
Java
JS
TS
C++
Manually filter out overly simple, untestable, and duplicate code
376 High-Quality Code Samples
Benchmark Construction
✦ Modify the requirements
✦ Modify the number and types of input and output parameters
Manual Modification
Generate modified problem requirements and function signature
Modified Problem
Import Statements
Function Signature
Requirement Description
ChatGPT-4o
Generate a solution and several test cases
Manually fix all errors in the solution and test cases
Reference Solution
Import Statements
Function Signature
Requirement Description
Reference Solution
Test Suite
Test Case 1
••••••
Test Case 5

```
import re
from typing import Dict, Any
def convert_named_to_positional_query(sql: str, params: Dict[str, Any], delimiter: str) -> Dict[str, Any]:
"""
  Convert a SQL query from named parameters to positional parameters, the named parameters flag is the given delimiter. Return a dict of positional_sql, param_list, execute_sql
  Args:
    sql (str): The SQL query containing named parameters
    params (Dict[str, Any]): A dictionary mapping named
        parameters to their values.
    delimiter (str): The delimiter used for the named
        parameters in the SQL query.such as $,#,:
  Returns:
    Dict[str, Any]: A dictionary containing:
      - 'positional_sql': The SQL query with positional
        placeholders
      - 'param_list': A list of values corresponding to the
        positional parameters.
      - 'execute_sql': The SQL query with actual values
        substituted in.
"""
```

```
import re
from typing import Dict, Any
def convert_named_to_positional_query(sql: str, params: Dict[str, Any], delimiter: str) -> Dict[str, Any]:
"""
  Convert a SQL query ......
"""

  result = {}
  param_pattern = rf"\{delimiter}(\w+)"
  ......
  # Return the final result containing the positional SQL,
  # parameter list, and executed SQL.
  return {
    "positional_sql": positional_sql,
    "param_list": values,
    "execute_sql": execute_sql
  }
```

```
  def test_basic_named_parameters(self):
    sql = "SELECT * FROM users WHERE username = $username AND age = $age"
    params = {"username": "john_doe", "age": 30}
    delimiter = "$"
    expected = { "positional_sql": "SELECT * FROM users WHERE username = $1 AND age = $2",
      "param_list": ["john_doe", 30],
      "execute_sql": "SELECT * FROM users WHERE username = john_doe AND age = 30"}
    result = convert_named_to_positional_query(sql, params, delimiter)
    self.assertEqual(result, expected)

  def test_sql_with_empty_string_parameter(self):
    ......
```

Fig. 1: The overview of the construction pipeline for RealisticCodeBench

additional test cases to achieve full complete coverage where possible. These additional test cases are also reviewed by the other two programmers to ensure their correctness.

Multi-Language Version Creation. To create multi-language versions of each programming problem in RealisticCodeBench, we leverage ChatGPT-4o for translation and adaptation across Python, Java, JavaScript, TypeScript, and C++. The process begins with a structured prompt containing the original problem, reference solution, and specific instructions to adapt code to each language's conventions. This includes placing docstrings before function declarations in languages like Java, JavaScript, TypeScript, and C++, and modifying symbols in docstrings (e.g., replacing single quotes with double quotes where necessary). Additionally, we ensure that function parameter types are accurately matched to the syntax and typing conventions of each language. For both reference solutions and test cases, we tailor naming conventions to each language's standards. JavaScript, Java, and TypeScript adopt camelCase, while C++ and Python adhere to snake_case. Certain programming tasks may not translate directly across languages due to unique data types or operations. In such cases, we retain the language-specific nature of the problem to reflect real-world coding practices. For example, a Python programming problem that calculates and returns the memory size of an object (such as a PyTorch tensor or NumPy array) remains in Python, as PyTorch and NumPy are specific to the Python ecosystem and do not have direct equivalents in Java, JavaScript, TypeScript, or C++. After initial translation, the same three programmers thoroughly review each version and test cases, making any necessary corrections or adjustments for coding standards and language nuances. If any language version has incomplete line and branch coverage, additional test cases are created and validated to address gaps.

Expert Review. We engage 13 experienced engineers (recruited via our industry connections as volunteers) to assess if the programming problems collected from GitHub could also represent coding tasks proprietary developers might address using LLMs (the three programmers mentioned earlier are not included in this group). Nearly three-quarters of these engineers come from major IT companies (e.g., Microsoft, Huawei, ByteDance, Tencent, Alibaba, Bilibili, and Meituan), while the rest are from smaller IT companies. With an average of 7.7 years of software development experience (ranging from 4 to 11 years and a median of 6 years), these engineers bring valuable industry insights to our benchmark validation. Over the past one to two years, they have used either their company's internal LLM tools or external tools like ChatGPT in their daily coding tasks. We task these engineers with assessing whether the programming problems (including their multi-language variants) align with realistic development scenarios—specifically, whether developers would realistically use LLMs to solve such similar problems in practical software development. We distribute the tasks to engineers via online documents (e.g., Google Docs), and they provide independent feedback by crossing out tasks that fail to meet these criteria. Only those programming problems approved by a majority (at least 10 out of 13 engineers) are retained, ensuring the benchmark mirrors tasks developers are likely to employ LLMs for in practical projects. Ultimately, 4 programming problems are excluded. For example, one problem involves creating a logging class to print log information, as developers typically use established logging frameworks (e.g., logging in Python or Log4j in Java) rather than implementing custom logging logic.

C. Benchmark Characteristics

The final benchmark comprises 376 programming problems translated across multiple languages: 361 in Python, 346 in JavaScript, 343 in TypeScript, 307 in Java, and 323 in C++, each accompanied by reference solutions and test cases. These

TABLE II: The detailed statistics of the benchmark

Language	Param (min)	Param (max)	Param (mean)	Sol (min)	Sol (max)	Sol (mean)	Prompt (mean)
Python	1	10	2.05	3	166	35.03	125.02
Java	1	8	1.98	8	205	55.43	115.93
JavaScript	1	10	2.12	3	181	37.57	121.81
TypeScript	1	9	2.09	4	189	39.84	127.74
C++	1	8	1.98	6	192	42.94	131.38
Average	1	9	2.04	4.8	192	42.16	124.4

376 problems include 364 function-based tasks and 12 class-based code generation tasks. Table II shows the detailed statistics of the benchmark across the five languages, including the number of function parameters per task (Param), the number of lines of reference solution code (Sol), and the number of tokens in each task prompt (Prompt). The average number of parameters across the five languages is 2.04. The average number of LOC in the reference solutions across these languages is 42.2. The average token length of code generation prompts—including requirement descriptions and function signatures—is 124.4. This complexity is greater than that seen in benchmarks such as HumanEval and MBPP but lower than that in benchmarks designed for more complex development scenarios like ClassEval. This suggests that our tasks and code generation requirements are more challenging than those in HumanEval but less so than those in ClassEval.

These 376 tasks cover nine distinct domains: data structures and algorithms, text processing, file handling, mathematical problems and scientific computing, date and time processing, data visualization and graphic applications, network programming, frontend development, and security. To systematically define these domains, we adopt a rigorous two-stage labeling process involving three authors. First, two authors independently label 40% of the tasks selected at random; any discrepancies arising from this stage are resolved in consultation with the third author to unify domain definitions. They then proceed to independently label the remaining 60% of the tasks, with residual discrepancies addressed through collaborative discussion to ensure consistency across all annotations. This diversity in task domains also introduces a range of complex input data types, which are classified into eight categories: strings, sequences, numbers, matrices, dictionaries, functions, complex data, and files. The first six are common basic data types. Among these, sequences refer to ordered data structures such as arrays and tuples; numbers include integers, floating-point values, and boolean values (represented as 0 or 1); and complex data encompasses language-specific unique data types—for example, DataFrames in Python, Objects in JavaScript/TypeScript, and Structs in C++. Files include development-related data files in various formats (e.g., CSV, XLSX, JSON, JSONL, XML, YAML) as well as image files and office documents (e.g., PDF, DOCX, DOC). This diverse input not only mirrors the complexity of real-world software development but also strengthens the broad applicability of the benchmark.

TABLE III: The overview of the 12 evaluated LLMs

	Model Name	**Organization**	**Sizes**	**Release Time**	**Open-Source**
General	GPT-4.1 [17]	OpenAI	-	2025	
	GPT-4o-mini [83]	OpenAI	-	2024	
	DeepSeek-V3 [19]	DeepSeek	671B	2024	✓
	Llama 3.1 [20]	Meta	8B	2024	✓
	Phi-3 [21]	Microsoft	7B	2024	✓
	Mistral [22]	Mistral AI	7B	2024	✓
	ChatGLM [23]	THUDM	6B	2024	✓
Coding	CodeGeex4 [28]	THUDM	9B	2023	✓
	DeepSeek-Coder [27]	DeepSeek	6.7B	2024	✓
	StarCoder2 [25]	BigCode	7B	2024	✓
	CodeGen2.5 [24]	Salesforce	7B	2023	✓
	CodeLlama [26]	Meta	7B	2023	✓

IV. Experimental Setup

We aim to comprehensively evaluate a diverse range of general-purpose and code-specific models that have been widely studied in recent code generation benchmarks [13]. Table III provides an overview of the LLMs examined, with the "Organization" column indicating the institution that developed the LLM, the "Sizes" column indicating model sizes in billions of parameters, the "Release Time" showing when the LLM was released, and "Open-Source" indicating whether the model's weights are publicly available. Overall, we evaluate 12 LLMs to ensure a thorough examination of the generalizability. Due to resource constraints, we limit our investigation to open-source models (except DeepSeek-V3) with parameter sizes of 10 billion, excluding smaller models (under 5 billion parameters) due to their limited efficacy. Additionally, we focus on models with relatively similar parameter sizes to minimize the impact of size differences and facilitate clearer performance comparisons across models. For closed-source modwels like GPT-4.1 and GPT-4o-mini, we use the OpenAI API interface [3]. For DeepSeek-V3, we rely on the DeepSeek API interface, as this model, while open-sourced, requires 8 GPUs with 80GB memory each to run in BF16 format for inference. For other open-source models, we obtain publicly released versions, with a preference for instruct versions trained using instruction fine-tuning, from official repositories and follow the provided documentation for setup and usage. These open-source models are run on a computational infrastructure featuring two NVIDIA GeForce RTX 3090-24GB GPUs. The maximum generation length for each solution is limited to 512 tokens to maintain consistency across models and prevent excessively long outputs.

We assess code generation performance using two distinct search strategies. In the greedy search setting, we generate a single code solution (n=1) per task by greedily choosing the most likely next token at each step, providing a deterministic

[3] During benchmark construction, we use ChatGPT-4o in a conversational window format; however, during evaluation, we employ different OpenAI models (GPT-4.1 and GPT-4o-mini) via their API. Critically, the evaluation process is entirely independent of benchmark construction, thus minimizing the risk of data leakage.

evaluation of the models' performance, with temperature=0 and top-p=1.0. Additionally, we use nucleus sampling to generate 10 code solutions (n=10) per task, with top-p=0.95 and temperature=0.8, to explore the models' ability to produce diverse outputs. Following established practices in code generation evaluation [6], [1], [10], we employ the pass@k metric to assess the functional correctness of generated code. For each problem, LLMs generate n code solutions, k of which are randomly selected for testing against reference test cases. The pass@k score measures the percentage of RealisticCodeBench problems, where at least one of the k-generated solutions is correct (i.e., passes all test cases). In our experiments, we report pass rates for k = 1, 3, and 5. For greedy search, we set n = 1 to compute pass@1, while for sampling-based evaluation, n = 10 is used to calculate pass@3 and pass@5. To mitigate high sampling variance, we adopt HumanEval's [6] unbiased estimator of pass@3 and pass@5, ensuring reliable and consistent evaluations of LLM performance across our benchmark.

V. Experimental Results

A. RQ1: How do LLMs perform on our RealisticCodeBench benchmark?

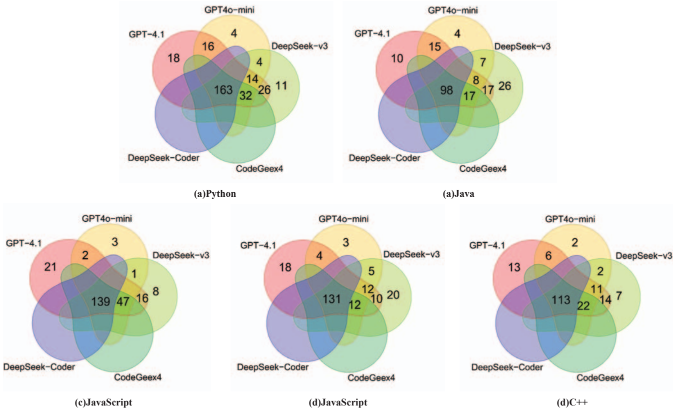
Table IV presents the pass@1, pass@3, and pass@5 metrics for the 12 evaluated LLMs on our RealisticCodeBench benchmark, with the top performances for both general and coding-specific LLMs highlighted in bold. GPT-4.1 achieves the highest average pass@1 across the five languages, with an average pass@1 score of 60.65%, followed by DeepSeek-V3 and GPT-4o-mini, which achieves an average pass@1 of 58.86% and 53.11%, respectively. GPT-4.1's average pass@1 surpasses that of DeepSeek-V3 by 1.79%. In Python, GPT-4.1 leads by a margin of 5.27%, yet the gap is much smaller in JavaScript and C++ (from 3.1% to 4.05%), with DeepSeek-V3 even outperforming GPT-4.1 in Java and TypeScript by 2.6% and 0.87%. Compared to GPT-4o-mini, DeepSeek-V3 achieves a 5.75% higher average pass@1. This performance trend remains consistent for pass@3, while for average pass@5, DeepSeek-V3 slightly surpasses GPT-4.1. Overall, these results highlight the superior code generation capabilities of GPT-4.1, DeepSeek-V3, and GPT-4o-mini. As an open-source model, DeepSeek-V3 offers a viable alternative for organizations capable of deploying 8 GPUs with 80GB of memory for inference, making it a competitive substitute for GPT-4.1 in code generation tasks. Among the smaller-parameter open-source models, CodeGeeX4 stands out as the best performer, achieving an average pass@1 score of 45.75%, with DeepSeek-Coder following closely at 38.08%. Notably, the difference between CodeGeeX4 and GPT-4o-mini in the programming languages is not large, ranging from 1.73% in JavaScript to 10.52% in Python. Additionally, among these small-parameter open-source models, code LLMs generally outperform general LLMs, primarily because code LLMs have been trained on more source code.

Figure 2 illustrates the number of problems each of the top five LLMs solved on their first attempt across five programming languages. The central overlapping sections of the Venn diagrams show the programming problems all models can solve, indicating a shared baseline competence. However, the distinct segments unique to each model highlight their specific strengths. GPT-4.1 and DeepSeek-V3 stand out with the largest unique areas, demonstrating their superior performance in solving programming problems that other models cannot, which underscores their stronger performance.

TABLE IV: The pass@1, pass@3, and pass@5 scores (%) of the 12 LLMs on our RealisticCodeBench benchmark

	Model	Python	Java	JavaScript	TypeScript	C++	Average
				Pass@1			
General	GPT-4.1	**74.52**	53.75	**65.03**	54.52	**55.42**	**60.65**
	GPT-4o-mini	64.54	48.53	55.49	48.69	48.30	53.11
	DeepSeek-V3	69.25	**56.35**	60.98	**55.39**	52.32	58.86
	Llama 3.1	44.87	22.48	40.17	32.36	21.67	32.31
	Phi-3	42.66	20.20	43.35	32.94	22.29	32.29
	Mistral	32.13	22.48	31.21	17.20	21.36	24.88
	ChatGLM	21.84	10.42	21.10	15.16	8.05	15.31
Coding	CodeGeex4	**54.02**	**37.46**	**53.76**	**41.69**	**41.80**	**45.75**
	DeepSeek-Coder	45.15	31.92	40.17	38.19	34.98	38.08
	StarCoder2	42.11	25.41	38.15	32.94	30.34	33.79
	CodeGen2.5	40.78	24.10	36.42	29.57	20.12	30.20
	CodeLlama	43.24	22.80	36.71	34.99	30.65	33.68
				Pass@3			
General	GPT-4.1	**77.63**	56.28	**68.45**	**58.84**	**59.28**	**64.10**
	GPT-4o-mini	69.47	53.56	59.47	53.36	52.64	57.70
	DeepSeek-V3	75.32	**57.63**	64.16	57.61	54.17	61.78
	Llama 3.1	46.25	26.41	44.37	35.22	23.39	35.13
	Phi-3	44.39	23.64	45.81	34.73	25.95	34.90
	Mistral	34.54	23.95	35.62	20.56	22.91	27.52
	ChatGLM	24.20	11.30	22.67	17.14	9.74	17.01
Coding	CodeGeex4	**55.79**	**41.33**	**54.98**	**44.83**	**45.27**	**48.44**
	DeepSeek-Coder	48.42	33.78	44.13	41.30	36.06	40.74
	StarCoder2	46.71	27.67	42.48	36.32	32.47	37.13
	CodeGen2.5	42.05	25.43	40.36	33.58	22.81	32.85
	CodeLlama	44.90	25.49	40.22	37.96	34.94	36.70
				Pass@5			
General	GPT-4.1	**79.89**	**60.04**	**71.68**	**63.29**	**63.47**	**67.67**
	GPT-4o-mini	72.26	56.05	63.37	57.14	55.82	60.93
	DeepSeek-V3	77.55	59.17	68.04	61.35	59.08	65.04
	Llama 3.1	50.94	29.23	46.91	36.76	25.39	37.85
	Phi-3	46.23	25.70	46.27	36.04	29.48	36.74
	Mistral	35.70	26.22	36.76	23.65	25.04	29.47
	ChatGLM	26.30	13.25	24.18	19.97	10.32	18.80
Coding	CodeGeex4	**58.83**	**42.46**	**59.24**	**54.03**	**46.73**	**52.26**
	DeepSeek-Coder	50.81	36.19	45.62	43.57	38.34	42.91
	StarCoder2	49.39	28.76	46.55	39.68	35.08	39.89
	CodeGen2.5	44.37	28.41	41.29	35.04	24.23	34.67
	CodeLlama	48.56	28.57	43.03	38.18	36.37	38.94

There are notable differences in pass@1 scores across the five programming languages. Python consistently shows higher pass rates across all models, with GPT-4.1 achieving an 74.52% pass@1, while languages like Java and C++ have comparatively lower scores. This disparity may stem from Python's extensive presence in LLM training data and its simpler syntax, which likely contributes to better performance on Python tasks. Across all models, the improvement from pass@1 to pass@3 and pass@5 remains relatively modest. For instance, GPT-4.1's pass rate rises from 60.65% at pass@1 to 67.67% at pass@5, DeepSeek-V3 improves from 58.86% to 65.04%, and GPT-4o-mini from 53.11% to 60.93%. We calculate the Levenshtein distance and conduct manual inspections for cases where models failed to solve the problem, revealing that the generated code among the five responses remained



(a)Python (a)Java

(c)JavaScript (d)JavaScript (d)C++

Fig. 2: The number of problems solved by the models

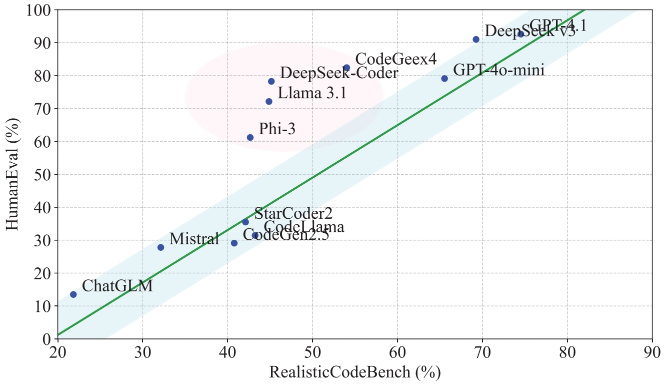


Fig. 3: The performance comparison of pass@1 for 12 LLMs between HumanEval and RealisticCodeBench (Python)

relatively similar. This observation indicates a lack of diversity in the generated solutions, suggesting that LLMs may not possess the depth of understanding necessary to solve certain complex problems effectively, even when allowed to generate multiple attempts.

> **Answer to RQ1:** GPT-4.1 achieves the highest average pass@1 across five languages, closely followed by the open-source model DeepSeek-V3. Among smaller-parameter open-source models, CodeGeeX4 stands out with strong performance, with a small gap from GPT-4o-mini.

B. RQ2: How does the performance of LLMs differ between RealisticCodeBench and HumanEval?

In this section, we compare the pass@1 performance of 12 LLMs on RealisticCodeBench and HumanEval. We omit more complex benchmarks like ClassEval and CoderEval, where all LLMs's pass@1 scores are generally low, making it challenging to assess performance correlation with RealisticCodeBench. Figure 3 displays a scatter plot illustrating the pass@1 performance of the 12 LLMs on HumanEval and RealisticCodeBench (Python). The scatter plot includes a green dashed line representing a linear fit and a light blue region indicating variance, suggesting that 8 of the LLMs exhibit linearly proportional growth in performance between HumanEval and RealisticCodeBench. This observation implies that, in most cases, RealisticCodeBench reflects the coding abilities of LLMs similarly to HumanEval. For instance, the performance gap between GPT-4.1 and DeepSeek-V3 remains relatively small across both HumanEval and RealisticCodeBench (Python). However, some models such as CodeGeeX4, Llama 3.1, DeepSeek-Coder, and Phi-3 display notably mismatched performances, as highlighted in the red-shaded area. Specifically, CodeGeeX4 drops substantially from a pass@1 of 82.3% on HumanEval to 54.02% on RealisticCodeBench (Python); Llama 3.1 decreases from 72.6% to 44.87%; DeepSeek-Coder falls from 78.6% to 45.15%; and Phi-3 declines from 61.0% to 42.66%.

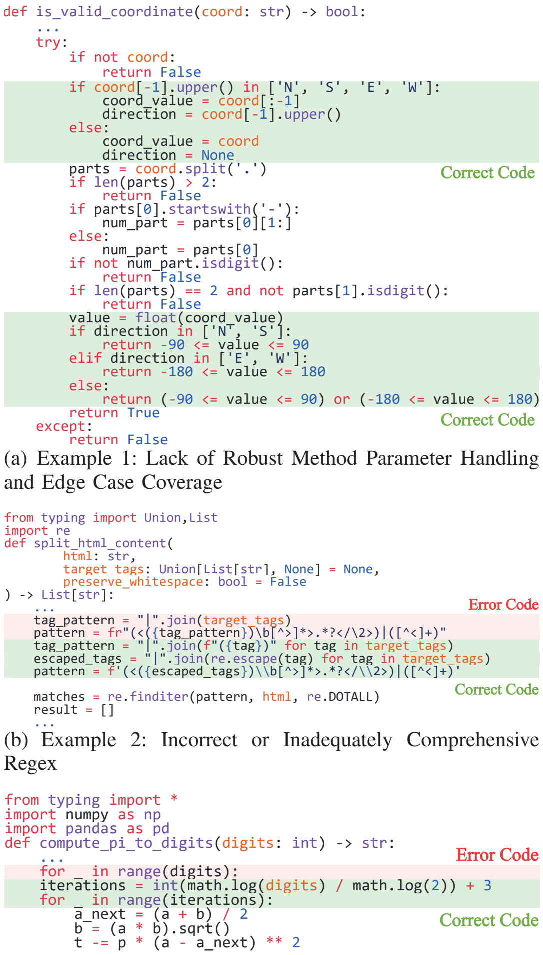
Several factors may explain this phenomenon. First, some LLMs' training sets might be overly optimized for HumanEval-style problems. Previous studies [74], [9], [84], [8] indicate that high performance on HumanEval often results from overfitting, as it is widely used and its data may contaminate LLM pre-training datasets. For example, Achiam et al. [17] reported that 25% of HumanEval had been contaminated in ChatGPT pre-training corpus. Additionally, contamination may arise from instruction fine-tuning datasets [8], as noted by Phi [85], [86], which reported considerable overlap between synthetic prompts and specific test samples in HumanEval. Second, RealisticCodeBench poses more challenging tasks than HumanEval, as it is designed to better reflect real-world coding scenarios where developers intend to use LLMs. RealisticCodeBench also adjusts requirements and parameters to prevent data leakage, thus revealing limitations in the generalization abilities of models like DeepSeek-Coder, Llama 3.1, and CodeGeeX4 when faced with real-world requirements and leakage-free tasks.

> **Answer to RQ2:** LLMs generally perform worse on RealisticCodeBench compared to HumanEval, with substantial performance drops observed in models such as CodeGeeX4, Llama 3.1, DeepSeek-Coder, and Phi-3.

C. RQ3: What are the common errors during code generation on RealisticCodeBench?

We further analyze cases where the highest-performing GPT-4.1 generates incorrect code within five attempts. In instances where GPT-4.1 fails to produce a correct solution during these first five tries, we extend the generation process to ten attempts. Most of these additional attempts yield correct solutions, suggesting that generating multiple responses with GPT-4.1 often leads to accurate answers. However, some problems remain unresolved. We identify three primary types of issues, as illustrated in Figure 4.

(1) **Lack of Robust Method Parameter Handling and Edge Case Coverage:** A common issue in GPT-4.1's generated code involves inadequate parameter validation, missing range checks, or incomplete format verification. For example,



```python
def is_valid_coordinate(coord: str) -> bool:
    ...
    try:
        if not coord:
            return False
        if coord[-1].upper() in ['N', 'S', 'E', 'W']:
            coord_value = coord[:-1]
            direction = coord[-1].upper()
        else:
            coord_value = coord
            direction = None
        parts = coord.split('.')
        if len(parts) > 2:
            return False
        if parts[0].startswith('-'):
            num_part = parts[0][1:]
        else:
            num_part = parts[0]
        if not num_part.isdigit():
            return False
        if len(parts) == 2 and not parts[1].isdigit():
            return False
        value = float(coord_value)
        if direction in ['N', 'S']:
            return -90 <= value <= 90
        elif direction in ['E', 'W']:
            return -180 <= value <= 180
        else:
            return (-90 <= value <= 90) or (-180 <= value <= 180)
        return True
    except:
        return False
```

(a) Example 1: Lack of Robust Method Parameter Handling and Edge Case Coverage

```python
from typing import Union,List
import re
def split_html_content(
        html: str,
        target_tags: Union[List[str], None] = None,
        preserve_whitespace: bool = False
) -> List[str]:
    ...
    tag_pattern = "|".join(target_tags)
    pattern = fr"(<({tag_pattern})\b[^>]*>.*?</\2>)|([^<]+)"
    tag_pattern = "|".join(f"({tag})" for tag in target_tags)
    escaped_tags = "|".join(re.escape(tag) for tag in target_tags)
    pattern = f'(<({escaped_tags})\\b[^>]*>.*?</\\2>)|([^<]+)'

    matches = re.finditer(pattern, html, re.DOTALL)
    result = []
    ...
```

(b) Example 2: Incorrect or Inadequately Comprehensive Regex

```python
from typing import *
import numpy as np
import pandas as pd
def compute_pi_to_digits(digits: int) -> str:
    ...
    for _ in range(digits):
    iterations = int(math.log(digits) / math.log(2)) + 3
    for _ in range(iterations):
        a_next = (a + b) / 2
        b = (a * b).sqrt()
        t -= p * (a - a_next) ** 2
```

(c) Example 3: Incorrect Mathematical Formula Application

Fig. 4: The three common error cases in GPT-4.1 code generation

in Example 1, when tasked with implementing a function to check if a string conforms to the specification for latitude and longitude identifiers, GPT-4.1 only verifies whether the format consists of numbers, while failing to check the valid value ranges. Latitudes should be within the range of -90 to 90, and longitudes within -180 to 180, but the generated code lacks this critical validation. By incorporating 1-2 specific test cases into the prompt, such as invalid values like "91.5" for latitude or "-181.3" for longitude, GPT-4.1 can generate corrected code, demonstrating its reliance on explicit prompt guidance for such constraints.

(2) **Incorrect or Inadequately Comprehensive Regex:** Another common issue is incorrect or inadequately comprehensive regex implementation. For example, in Example 2, when tasked with splitting an HTML string into tag blocks and non-tag text blocks based on specified markers, GPT-4.1 generates a flawed regular expression. This regex fails to handle cases where there are multiple consecutive segments of non-tag text within the HTML structure, leading to incorrect partitioning of content. This underscores that when using GPT-4.1 to generate regex-involving code, users should verify coverage of all scenarios or explicitly provide pattern examples.

(3) **Incorrect Mathematical Formula Application:** For problems involving mathematical algorithms, GPT-4.1 occasionally misinterprets algorithmic properties when the prompt does not explicitly clarify them. For example, in Example 3, when tasked with generating code for the Gauss-Legendre algorithm to calculate π to a specified number of decimal places, GPT-4.1 demonstrates a misunderstanding of the algorithm. The Gauss-Legendre algorithm exhibits quadratic convergence, requiring only $\log_2(\text{digits})$ iterations rather than digits iterations, yet the generated code uses the incorrect number of iterations. When we explicitly state the quadratic convergence property and the correct iteration formula in the prompt, GPT-4.1 generates accurate code, emphasizing the need for precise algorithmic descriptions when dealing with complex mathematical computations.

> 📢 **Answer to RQ3:** Common errors in GPT-4.1's code generation on RealisticCodeBench include insufficient handling of edge cases and method parameter robustness, incorrect or inadequately comprehensive regex, and the misapplication of mathematical formulas.

VI. Discussion

A. *Implications*

Unlike widely-used benchmarks like HumanEval-focused on algorithmic and basic programming tasks—our benchmark reflects the types of code developers commonly generate with LLMs in real-world development scenarios. Compared to other GitHub-derived benchmarks like CoderEval and CodeEvoBench-designed to test the upper limits of LLM capabilities and provide insights for improving LLM performance in handling complex tasks—our benchmark offers a complementary perspective. While we recognize these benchmarks' value, ours serves as a practical supplement, offering insights from real-world LLM usage scenarios. In addition, similar to CoderEval and CodeEvoBench, our benchmark remains challenging even for advanced models: GPT-4.1 achieves a pass@1 rate of only 74.52% in Python, indicating 25.48% of tasks cannot be solved correctly in one attempt. This highlights the need for continuous improvement across all LLM categories, from small-parameter models to large-scale architectures, and validates the utility of our benchmark—since even state-of-the-art models do not exhibit perfect performance. Therefore, we recommend that **newly developed LLMs be evaluated using our benchmark to give developers a clearer understanding of model performance on tasks that reflect current, practical coding needs that LLMs can address reliably**.

Given data privacy concerns, as noted by Liang et al. [14], 41% of developers fear LLMs accessing private codebases due to data privacy concerns. Our findings indicate that open-source models like DeepSeek-V3 and CodeGeeX4-9B offer privacy-conscious alternative. The performance differential between DeepSeek-V3 and GPT-4.1, with an average pass@1 gap of only 1.79%, suggests that DeepSeek-V3, despite requiring a robust hardware setup of 8 GPUs with 80GB each, is a feasible choice for well-resourced enterprises that prioritize data privacy. CodeGeeX4-9B shows competitive performance compared to the proprietary model GPT-4o-mini on some programming languages; for instance, in Python, it achieves a pass@1 rate of 54.02% and a pass@5 rate of 58.83%, narrowing the accuracy gap with GPT-4o-mini (pass@1 of 64.54%) to only 5.71% when generating multiple solutions. Moreover, CodeGeeX4-9B's operational feasibility on a server equipped with dual NVIDIA GeForce RTX 3090 (24GB) GPUs—costing around $3,000—makes it a cost-effective option for individual developers and small firms. However, for deploying larger models with parameters exceeding 9B, higher-end GPUs like the NVIDIA A100 or A800 would be required, with starting costs around $20,000. Thus, **for enterprises with substantial funding and a focus on data privacy, DeepSeek-V3 is recommended, while CodeGeeX4-9B is advised for privacy-conscious developers or smaller companies on tighter budgets**.

The error case analysis in Section V-C underscores **the need for research focused on enhancing LLM robustness in handling boundary conditions, domain-specific formulas, and accurate and comprehensive regular expressions implementations**. For tasks requiring robust parameter handling and comprehensive edge case coverage, developers should include specific test cases within the prompt to highlight these aspects effectively. For mathematical or formula-based problems, developers should provide explicit formulas within the prompt to guide the model toward accurate computations, thereby reducing the risk of errors due to incorrect formula application. These strategies can collectively enhance LLM reliability in code generation tasks.

B. Threats to Validity

We evaluate a single closed-source LLM (the GPT series from OpenAI), despite the existence of other closed-source models such as Google's Gemini. The decision to focus on OpenAI's GPT models is based on their widespread use and demonstrated effectiveness. However, this may introduce selection bias, as other models might perform differently under similar conditions. Moreover, Liang et al. [14] found that 41% of developers are hesitant to use LLMs due to concerns that code generation tools could access their private codebases. To address this, we prioritize the exploration of open-source LLMs. In total, we examine five general-purpose open-source LLMs and five code-specific open-source LLMs to mitigate bias and broaden our analysis. Additionally, our computational resources—two NVIDIA GeForce RTX 3090 GPUs—limit our ability to evaluate larger open-source models like StarCoder 15B and DeepSeek-Coder-V2 16B, which trigger out-of-memory errors during testing. As a result, our analysis is restricted to models with a maximum size of 10 billion parameters. We plan to expand our evaluation to include larger LLMs as more computational resources become available.

Our benchmark focuses exclusively on code explicitly labeled as LLM-generated on GitHub, with the goal of reflecting how open-source developers use LLMs. However, it is likely that only some developers annotate their LLM-generated code as such—a factor that could skew the task distribution within our benchmark. Further, the limited number of programming problems (376) may not fully capture the diversity of real-world coding tasks; this scale is currently constrained by the extensive manual effort required (approximately 700 person-hours). With the increasing use of LLMs in open-source development, we plan to expand our benchmark by incorporating more programming problems from GitHub and other repositories.

To mitigate potential risks of data leakage, we adapt the programming problems derived from GitHub code, altering the types and quantities of input/output parameters. We calculate the Levenshtein distance between the original GitHub code and the LLM-generated code, finding substantial differences. For example, the Levenshtein distance between the original GitHub Python code and the GPT-4.1-generated code for the adapted problem is 509.27. Additionally, we manually review the original GitHub code and the LLM-generated code, confirming that they are indeed very dissimilar, suggesting minimal risk of data leakage.

VII. Conclusion

We develop RealisticCodeBench to better align with the types of coding tasks developers commonly address using LLMs. This benchmark includes 361 Python, 346 JavaScript, 343 TypeScript, 307 Java, and 323 C++ problems, reflecting developers' everyday coding needs. Experimental evaluations of 12 LLMs reveal that, while GPT-4.1 achieves the highest average pass@1, open-source models like DeepSeek-V3 and CodeGeeX4 can serve as viable alternatives for companies and smaller organizations focused on privacy, cost-efficiency, and robust code generation. In comparing performance gaps between HumanEval and RealisticCodeBench, we find that some LLMs may be overly optimized for HumanEval-style problems rather than practical coding applications. Lastly, our analysis of failed cases highlights critical areas where LLMs fall short in RealisticCodeBench, identifying opportunities for improvement in handling complex, real-world coding tasks.

Acknowledgements

This research is supported by National Key R&D Program of China (No. 2024YFB4506400).

References

[1] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[2] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[3] Z. Sun, X. Du, Z. Yang, L. Li, and D. Lo, "Ai coders are among us: Rethinking programming language grammar towards efficient code generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1124–1136.

[4] X. Yu, L. Liu, X. Hu, J. W. Keung, J. Liu, and X. Xia, "Fight fire with fire: How much can we trust chatgpt on source code-related tasks?" *IEEE Transactions on Software Engineering*, vol. 50, no. 12, pp. 3435–3453, 2024.

[5] X. Yu, Z. Zhang, F. Niu, X. Hu, X. Xia, and J. Grundy, "What makes a high-quality training dataset for large language models: A practitioners' perspective," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*. ACM, 2024, pp. 656–668.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

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

[8] A. Matton, T. Sherborne, D. Aumiller, E. Tommasone, M. Alizadeh, J. He, R. Ma, M. Voisin, E. Gilsenan-McMahon, and M. Gallé, "On leakage of code generation evaluation datasets," *arXiv preprint arXiv:2407.07565*, 2024.

[9] S. Zhang, H. Zhao, X. Liu, Q. Zheng, Z. Qi, X. Gu, Y. Dong, and J. Tang, "Naturalcodebench: Examining coding performance mismatch on humaneval and natural user queries," in *Findings of the Association for Computational Linguistics ACL 2024*, 2024, pp. 7907–7928.

[10] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[11] J. Li, G. Li, X. Zhang, Y. Dong, and Z. Jin, "Evocodebench: An evolving code generation benchmark aligned with real-world code repositories," *arXiv preprint arXiv:2404.00599*, 2024.

[12] J. Feng, J. Liu, C. Gao, C. Y. Chong, C. Wang, S. Gao, and X. Xia, "Complexcodeeval: A benchmark for evaluating large code models on more complex code," *arXiv preprint arXiv:2409.10280*, 2024.

[13] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[14] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[15] X. Yu, L. Liu, X. Hu, J. Liu, and X. Xia, "Where are large language models for code generation on github?" *arXiv preprint arXiv:2406.19544*, 2024.

[16] Anonymity, "Supplemental materials," https://github.com/XIAOYU-CS/RealisticCodeBench, 2025.

[17] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[18] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[19] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[20] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[21] M. Abdin, S. A. Jacobs, A. A. Awan, J. Aneja, A. Awadallah, H. Awadalla, N. Bach, A. Bahree, A. Bakhtiari, H. Behl *et al.*, "Phi-3 technical report: A highly capable language model locally on your phone," *arXiv preprint arXiv:2404.14219*, 2024.

[22] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[23] T. GLM, A. Zeng, B. Xu, B. Wang, C. Zhang, D. Yin, D. Rojas, G. Feng, H. Zhao, H. Lai *et al.*, "Chatglm: A family of large language models from glm-130b to glm-4 all tools," *arXiv preprint arXiv:2406.12793*, 2024.

[24] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," *arXiv preprint arXiv:2305.02309*, 2023.

[25] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[26] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[27] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[28] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5673–5684.

[29] Z. Wang, G. Cuenca, S. Zhou, F. F. Xu, and G. Neubig, "Mconala: a benchmark for code generation from multiple natural languages," *arXiv preprint arXiv:2203.08388*, 2022.

[30] J. Li, G. Li, Y. Zhao, Y. Li, H. Liu, H. Zhu, L. Wang, K. Liu, Z. Fang, L. Wang *et al.*, "Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories," *arXiv preprint arXiv:2405.19856*, 2024.

[31] D. G. Paul, H. Zhu, and I. Bayley, "Sceneval: A benchmark for scenario-based evaluation of code generation," *arXiv preprint arXiv:2406.12635*, 2024.

[32] 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.

[33] J. Zheng, B. Cao, Z. Ma, R. Pan, H. Lin, Y. Lu, X. Han, and L. Sun, "Beyond correctness: Benchmarking multi-dimensional code generation for large language models," *arXiv preprint arXiv:2407.11470*, 2024.

[34] D. Zheng, Y. Wang, E. Shi, R. Zhang, Y. Ma, H. Zhang, and Z. Zheng, "Humanevo: An evolution-aware benchmark for more realistic evaluation of repository-level code generation," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 764–764.

[35] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, 2014, pp. 1–10.

[36] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.

[37] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," *arXiv preprint arXiv:1808.09588*, 2018.

[38] R. Agashe, S. Iyer, and L. Zettlemoyer, "Juice: A large scale distantly supervised dataset for open domain context-based code generation," *arXiv preprint arXiv:1910.02216*, 2019.

[39] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: A natural and reliable benchmark for data science code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 18 319–18 345.

[40] S. Chandel, C. B. Clement, G. Serrato, and N. Sundaresan, "Training and evaluating a jupyter notebook data science assistant," *arXiv preprint arXiv:2201.12901*, 2022.

[41] J. Huang, C. Wang, J. Zhang, C. Yan, H. Cui, J. P. Inala, C. Clement, N. Duan, and J. Gao, "Execution-based evaluation for data science code generation models," *arXiv preprint arXiv:2211.09374*, 2022.

[42] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[43] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[44] X. Tang, Y. Liu, Z. Cai, Y. Shao, J. Lu, Y. Zhang, Z. Deng, H. Hu, K. An, R. Huang *et al.*, "Ml-bench: Evaluating large language models and agents for machine learning tasks on repository-level code," *arXiv e-prints*, pp. arXiv–2311, 2023.

[45] R. Li, J. Fu, B.-W. Zhang, T. Huang, Z. Sun, C. Lyu, G. Liu, Z. Jin, and G. Li, "Taco: Topics in algorithmic code generation dataset," *arXiv preprint arXiv:2312.14852*, 2023.

[46] M. L. Siddiq and J. C. Santos, "Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation

techniques," in *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, 2022, pp. 29–33.

[47] X. Tang, B. Qian, R. Gao, J. Chen, X. Chen, and M. B. Gerstein, "Biocoder: a benchmark for bioinformatics code generation with large language models," *Bioinformatics*, vol. 40, no. Supplement_1, pp. i266–i276, 2024.

[48] Y. Xia, Y. Chen, T. Shi, J. Wang, and J. Yang, "Aicodereval: Improving ai domain code generation of large language models," *arXiv preprint arXiv:2406.04712*, 2024.

[49] Y. Fu, E. Baker, and Y. Chen, "Constrained decoding for secure code generation," *arXiv preprint arXiv:2405.00218*, 2024.

[50] J. Cao, Z. Chen, J. Wu, S.-C. Cheung, and C. Xu, "Javabench: A benchmark of object-oriented code generation for evaluating large language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 870–882.

[51] Q. Shi, M. Tang, K. Narasimhan, and S. Yao, "Can language models solve olympiad programming?" *arXiv preprint arXiv:2404.10952*, 2024.

[52] T. Wu, W. Wu, X. Wang, K. Xu, S. Ma, B. Jiang, P. Yang, Z. Xing, Y.-F. Li, and G. Haffari, "Versicode: Towards version-controllable code generation," *arXiv preprint arXiv:2406.07411*, 2024.

[53] P. Vijayaraghavan, L. Shi, S. Ambrogio, C. Mackin, A. Nitsure, D. Beymer, and E. Degan, "Vhdl-eval: A framework for evaluating large language models in vhdl code generation," *arXiv preprint arXiv:2406.04379*, 2024.

[54] D. Nichols, J. H. Davis, Z. Xie, A. Rajaram, and A. Bhatele, "Can large language models write parallel code?" in *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, 2024, pp. 281–294.

[55] P. Haller, J. Golde, and A. Akbik, "Pecc: Problem extraction and coding challenges," *arXiv preprint arXiv:2404.18766*, 2024.

[56] P. T. J. Kon, J. Liu, Y. Qiu, W. Fan, T. He, L. Lin, H. Zhang, O. M. Park, G. S. Elengikal, Y. Kang *et al.*, "Iac-eval: A code generation benchmark for infrastructure-as-code programs."

[57] D. Zan, B. Chen, Z. Lin, B. Guan, Y. Wang, and J.-G. Lou, "When language model meets private library," *arXiv preprint arXiv:2210.17236*, 2022.

[58] D. Zan, B. Chen, D. Yang, Z. Lin, M. Kim, B. Guan, Y. Wang, W. Chen, and J.-G. Lou, "Cert: continual pre-training on sketches for library-oriented code generation," *arXiv preprint arXiv:2206.06888*, 2022.

[59] S. Ouyang, D. Huang, J. Guo, Z. Sun, Q. Zhu, and J. M. Zhang, "Ds-bench: A realistic benchmark for data science code generation," *arXiv preprint arXiv:2505.15621*, 2025.

[60] K. Xu, Y. Mao, X. Guan, and Z. Feng, "Web-bench: A llm code benchmark based on web standards and frameworks," *arXiv preprint arXiv:2505.07473*, 2025.

[61] C. Dilgren, P. Chiniya, L. Griffith, Y. Ding, and Y. Chen, "Secrepobench: Benchmarking llms for secure code generation in real-world repositories," *arXiv preprint arXiv:2504.21205*, 2025.

[62] Y. Peng, J. Wan, Y. Li, and X. Ren, "Coffe: A code efficiency benchmark for code generation," *arXiv preprint arXiv:2502.02827*, 2025.

[63] Y. Cui, "Tests as prompt: A test-driven-development benchmark for llm code generation," *arXiv preprint arXiv:2505.09027*, 2025.

[64] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman *et al.*, "Multipl-e: A scalable and extensible approach to benchmarking neural code generation," *arXiv preprint arXiv:2208.08227*, 2022.

[65] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang *et al.*, "Multi-lingual evaluation of code generation models," *arXiv preprint arXiv:2210.14868*, 2022.

[66] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[67] H. M. Babe, S. Nguyen, Y. Zi, A. Guha, M. Q. Feldman, and C. J. Anderson, "Studenteval: a benchmark of student-written prompts for large language models of code," *arXiv preprint arXiv:2306.04556*, 2023.

[68] C. S. Xia, Y. Deng, and L. Zhang, "Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm," *arXiv preprint arXiv:2403.19114*, 2024.

[69] R. Qiu, W. W. Zeng, H. Tong, J. Ezick, and C. Lott, "How efficient is llm-generated code? a rigorous & high-standard benchmark," *arXiv preprint arXiv:2406.06647*, 2024.

[70] Y. Hao, G. Li, Y. Liu, X. Miao, H. Zong, S. Jiang, Y. Liu, and H. Wei, "Aixbench: A code generation benchmark dataset," *arXiv preprint arXiv:2206.13179*, 2022.

[71] Z. Wang, S. Zhou, D. Fried, and G. Neubig, "Execution-based evaluation for open-domain code generation," *arXiv preprint arXiv:2212.10481*, 2022.

[72] L. Fu, H. Chai, S. Luo, K. Du, W. Zhang, L. Fan, J. Lei, R. Rui, J. Lin, Y. Fang *et al.*, "Codeapex: A bilingual programming evaluation benchmark for large language models," *arXiv preprint arXiv:2309.01940*, 2023.

[73] J. Dai, J. Lu, Y. Feng, R. Ruan, M. Cheng, H. Tan, and Z. Guo, "Mhpp: Exploring the capabilities and limitations of language models beyond basic code generation," *arXiv preprint arXiv:2405.11430*, 2024.

[74] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," *arXiv preprint arXiv:2403.07974*, 2024.

[75] M. A. M. Khan, M. S. Bari, D. Long, W. Wang, M. R. Parvez, and S. Joty, "Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 6766–6805.

[76] S. Quan, J. Yang, B. Yu, B. Zheng, D. Liu, A. Yang, X. Ren, B. Gao, Y. Miao, Y. Feng *et al.*, "Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings," *arXiv preprint arXiv:2501.01257*, 2025.

[77] W. Hu, J. Duan, C. Wei, L. Zhang, Y. Zhang, and K. Xu, "Dynacode: A dynamic complexity-aware code benchmark for evaluating large language models in code generation," *arXiv preprint arXiv:2503.10452*, 2025.

[78] S. Wang, Z. Wang, D. Ma, Y. Yu, R. Ling, Z. Li, F. Xiong, and W. Zhang, "Codeflowbench: A multi-turn, iterative benchmark for complex code generation," *arXiv preprint arXiv:2504.21751*, 2025.

[79] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang *et al.*, "Automatic code summarization via chatgpt: How far are we?" *arXiv preprint arXiv:2305.12865*, 2023.

[80] A. Elangovan, J. He, and K. Verspoor, "Memorization vs. generalization: Quantifying data leakage in nlp performance evaluation," *arXiv preprint arXiv:2102.01818*, 2021.

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

[82] jerber, "fastgql," https://github.com/jerber/fastgql/blob/4c308e742685e0a1cf4dc6d05f29cfbaea2d039a/fastgql/query_builders/sql/query_builder.py#L464, 2025.

[83] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[84] D. Zheng, Y. Wang, E. Shi, R. Zhang, Y. Ma, H. Zhang, and Z. Zheng, "Towards more realistic evaluation of llm-based code generation: an experimental study and beyond," *arXiv preprint arXiv:2406.06918*, 2024.

[85] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi *et al.*, "Textbooks are all you need," *arXiv preprint arXiv:2306.11644*, 2023.

[86] Y. Li, S. Bubeck, R. Eldan, A. Del Giorno, S. Gunasekar, and Y. T. Lee, "Textbooks are all you need ii: phi-1.5 technical report," *arXiv preprint arXiv:2309.05463*, 2023.