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

Xiao Yu
The State Key Laboratory of Blockchain and Data Security
Zhejiang University
Hangzhou, China
xiao.yu@zju.edu.cn

Haoxuan Chen
School of Computer Science and Artificial Intelligence
Wuhan University of Technology
Wuhan, China
haoxuan.chen@whut.edu.cn

Lei Liu
Faculty of Electronic and Information Engineering
Xi'an Jiaotong University
Xi'an, China
lei.liu@stu.xjtu.edu.cn

Xing Hu
The State Key Laboratory of Blockchain and Data Security
Zhejiang University
Hangzhou, China
xinghu@zju.edu.cn

Jacky Wai Keung
Department of Computer Science
City University of Hong Kong
Hong Kong, China
jacky.keung@cityu.edu.hk

Xin Xia
The State Key Laboratory of Blockchain and Data Security
Zhejiang University
Hangzhou, China
xin.xia@acm.org

***Abstract*—Evaluating the code generation capabilities of Large Language Models (LLMs) remains an open question. Existing benchmarks like HumanEval and MBPP focus primarily on algorithmic and basic programming tasks, which do not fully capture the intricacies of real-world coding challenges. Recently, more advanced benchmarks—such as CoderEval, EvoCodeBench, and ClassEval—have been introduced to address this gap, evaluating 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 simpler tasks that high-performing models can handle reliably.**

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 high-star 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 417 programming problems translated across multiple languages: 392 in Python, 376 in JavaScript, 372 in TypeScript, 339 in Java, and 353 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 achieves the highest average pass@1 score across languages, closely followed by DeepSeek-V2.5-236B, suggesting that DeepSeek-V2.5-236B provides a viable open-source alternative to GPT-4 for large companies with sufficient GPU resources and privacy concerns. CodeGeeX4-9B, a cost-effective model, emerges as a suitable substitute for GPT-3.5 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], [5]. Recent advances in Large Language Models (LLMs) have further accelerated progress in this field. Various LLMs, including GPT-4 [6], DeepSeek-V2.5 [7], CodeLlama [8], and CodeGeeX [9], have been developed by researchers and organizations through training on massive general and code-specific datasets.

To evaluate the performance of these emerging LLMs on code generation tasks, several benchmarks have been introduced, starting with HumanEval [10] and MBPP [11]. Reporting performance on these benchmarks has seemingly become mandatory for a model to be considered competitive in code generation [12]. Indeed, nearly all new LLMs released in 2023-2024 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 [13]. To address this, more complex benchmarks—such as CoderEval [14], EvoCodeBench [15], ComplexCodeEval [16], and ClassEval [17]—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 [14], while GPT-4 reaches just a 37.0% pass@1 rate for class-level code generation on ClassEval [17], which can discourage developers from using LLMs for such sophisticated code generation tasks. A large-scale survey conducted by Liang et al. [18] found 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 [14], EvoCodeBench [15], ComplexCodeEval [16], and ClassEval [17], current LLMs are not yet ready to generate such sophisticated code at scale. Instead, developers are more likely to use LLMs for simpler, more manageable coding tasks that high-performing models (e.g., GPT-4) 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, Yu et al. [19] 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 slight modifications to each sample's requirements while preserving the original intent and complexity. Where applicable, we also adjust the number and types of input and output parameters to further mitigate data leakage risks. Using GPT-4, we generate reference solutions for each modified programming problem, followed by manual corrections. GPT-4 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 GPT-4 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 417 programming problems translated across multiple languages: 392 in Python, 376 in JavaScript, 372 in TypeScript, 339 in Java, and 353 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, GPT-3.5, DeepSeek-V2.5-236B, Llama 3.1-8B, CodeGeeX4-9B, DeepSeek-Coder-6.7B, CodeLlama-7B, and StarCoder2-7B. Experimental results show that, across five programming languages, GPT-4 achieves the highest average pass@1 score at 67.27%, with DeepSeek-V2.5-236B close behind at 66.08%. This suggests that companies with sufficient resources and privacy concerns could consider deploying DeepSeek-V2.5-236B as an open-source alternative to GPT-4 for everyday coding tasks. CodeGeeX4-9B achieves an average pass@1 score of 48.14%, compared to GPT-3.5's 58.83%, 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-3.5, 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 (55.47% and 47.73%, 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.

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

(3) We for the first time conduct a comprehensive literature review to identify and analyze a set of 51 code generation benchmark studies from 2021. Researchers can use this set as a foundation for further studies.

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 [6] and GPT-3.5 [20], both of which have demonstrated significant capabilities across a wide range of tasks. Additionally, other general-purpose models like DeepSeek-V2.5 [7], Llama 3.1 [8], 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 [9]. 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 2024 by using a forward snowballing approach[1] [28]. The starting year of 2021 is selected, as it marks the publication of the earliest prominent benchmarks for code generation, which included test cases for evaluating LLMs' code generation accuracy (i.e., APPS [29], HumanEval [10], and MBPP [11]). Although earlier code generation benchmarks, such as Concode [30] and JuICe [31], 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 [29], HumanEval [10], and MBPP [11] 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 their 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 51 code generation benchmark.

The benchmarks identified can be broadly classified into two categories. The first category, comprising 22 papers [11], [32], [33], [34], [35], [36], [37], [38], [36], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], focuses on domain-specific code generation abilities, such as generating security code [39], [42], VHDL code [46], bioinformatics code [40], Verilog code [36], data science code [11], [33], [34], AI code [41], object-oriented code in Java [43], parallel code [47], Infrastructure-as-Code (IaC) programs [49], etc. The second category focuses on evaluating general code generation capabilities, which aligns with the goals of our benchmark. Due to space constraints, we only limit our focus to the capabilities of the general benchmarks. Tables I and II provide an overview of 29 selected 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 problem's input information (usually the requirements and function signature) ("#Tokens"), and the best model performance (usually GPT-4) in terms of pass@1. Table I lists benchmarks where the best LLM's pass@1 exceeds 60%, while Table II includes those where the best LLM's pass@1 falls below 60%. In the tables, "_" indicates that the corresponding information was not provided in the benchmark paper.

Table I includes a total of **13 benchmarks where the**

[1]The literature search was conducted on October 2024.

TABLE I: The current general code generation benchmarks with pass@1 scores above 60%

Benchmark	**Year**	**Language**	**Source**	**Granularity**	**#Tasks**	**#LOC**	**#Tokens**	**Pass@1**
HumanEval [10]	2021	Python	Manual	Function	164	11.5	24.4	90.2 % (GPT-4o)
MBPP [11]	2021	Python	Manual	Function	974	6.8	24.2	83.5 % (GPT-4)
MultiPL-E [52]	2022	Multilingual	HumanEval, MBPP	Function	164, 974	11.5, 6.8	24.4, 24.2	90.2 % (GPT-4o), 83.5 % (GPT-4)
Multi-HumanEval [53]	2022	Multilingual	HumanEval	Function	164	11.5	24.4	90.2 % (GPT-4o)
MBXP [53]	2022	Multilingual	MBPP	Function	974	6.8	24.2	83.5 % (GPT-4)
HumanEval+ [54]	2023	Python	HumanEval	Function	164	11.5	24.4	76.2 % (GPT-4)
HumanEval-X [9]	2023	Multilingual	HumanEval	Function	820	11.5	24.4	90.2 % (GPT-4o)
StudentEval [55]	2023	Python	Manual	Function	48	-	-	63.6 % (StarChat-Alpha)
EvoEval [56]	2024	Python	HumanEval	Function	828	-	-	62.0 % (GPT-4)
ENAMEL [57]	2024	Python	HumanEval	Function	142	11.5	24.4	83.1 % (GPT-4)
ScenEval [58]	2024	Java	W3Resources, Stack Overflow, Textbooks	Statement, Function, Class	12864	1-50	-	75.6 % (ChatGPT)
RACE [59]	2024	Python	HumanEval+, MBPP+, ClassEval, LeetCode	Class, Function	923	-	-	70.1 % (GPT-4-o1-mini)
LBPP [12]	2024	Python	Manual	Function	161	-	-	64.0 % (Claude-3.5-Sonnet)
RealisticCodeBench	**2024**	**Multilingual**	**GitHub**	**Function, Class**	**417**	**30.6**	**95.2**	83.46% (GPT-4)

pass@1 exceeds 60% in LLM evaluations. Among them, 4 benchmarks [10], [11], [55], [12] are manually created, 2 [58], [59] are created by collecting open-source data, while 7 [52], [53], [54], [9], [56], [57], [59] are adapted or extended versions of HumanEval, and 3 [52], [53], [59] are adapted or extended versions of MBPP. Chen et al. [10] first proposed the HumanEval dataset, which consists of 164 handwritten Python programming problems, primarily involving pure algorithm and string manipulation. At that time, the State-Of-The-Art (SOTA) model Codex-12B achieved a pass@1 of 28.8% on this dataset. Subsequently, Cassano et al. [52], Athiwaratkun et al. [53], and Zheng et al. [9] extended HumanEval to other language versions, forming MultiPL-E, Multi-HumanEval, and Humaneval-X. Currently, the latest SOTA LLMs achieve impressive results on HumanEval (Python) [12], with pass@1 reaching 90.2% for GPT-4o when employing a model debugger strategy [60] as shown on the leaderboard [2], and 84.1% for GPT-4 in a zero-shot setting [27]. Furthermore, Liu et al. [54] proposed HumanEval+, which adds more test cases to each programming problem in HumanEval to ensure a more rigorous evaluation, achieving 76.2% pass@1 on GPT-4. Later, Xia et al. [56] revamped HumanEval to create new, more innovative, challenging, and diverse tasks, forming EvoEval, achieving a pass@1 of 62.0% on GPT-4. Recently, Qiu et al. [57] proposed ENAMEL, which selected 142 tasks from HumanEval with relatively high complexity to test LLMs' ability to generate efficient (low time complexity) code, achieving a pass@1 of 83.1% on GPT-4.

Another classic general code generation benchmark is the MBPP dataset proposed by Austin et al. [11], which includes 974 manually created short Python programs. The problems range from simple numeric manipulations to tasks requiring basic usage of standard library functions. Subsequently, Cassano et al. [52] and Athiwaratkun et al. [53] expanded MBPP to create the MultiPL-E and MBXP multilingual datasets. Currently, the latest SOTA model, GPT-4, achieves a pass@1 of up to 83.5% on MBPP with the use of multi-agent strategies [61], as shown on the leaderboard [3].

Among other manually created datasets, Babe et al. [55] introduced StudentEval, which contains 48 introductory Python problems from first-year computer science courses, including quizzes, lab exercises, and homework tasks with minor modifications to prevent releasing answers to current assignments. StarChat-Alpha-15.5B achieved the highest pass@1 of 63.6% on this benchmark. Later, Matton et al. [12] invited annotators with competitive programming expertise to create LBPP, a collection of 161 Python tasks similar in type but more challenging than those in HumanEval. On LBPP, Claude-3.5-Sonnet achieved a pass@1 rate of 64.0%.

Recently, two additional benchmarks were created from open-source data. Paul et al. [58] developed ScenEval, collecting various statements, methods, and classes from open-source platforms 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. [59] combined datasets such as HumanEval, MBPP, ClassEval, and LeetCode to form RACE, a dataset of moderate complexity where GPT-4o-mini and Claude-3.5-Sonnet achieved pass@1 rates of 70.1% and 62.3%, respectively.

[2]https://paperswithcode.com/sota/code-generation-on-humaneval

[3]https://paperswithcode.com/sota/code-generation-on-mbpp

Motivation 1: Many widely-used benchmarks in Table I predominantly focus on algorithmic and basic programming tasks, allowing SOTA LLMs to achieve relatively high pass@1 (often exceeding 60%). However, these tasks fail to capture the complexity and diversity of real-world coding scenarios that developers face.

From Table II, there are a total of **14 benchmarks where the pass@1 is below 60%** in existing LLM evaluations, and 2 benchmarks [63], [16] without a pass@1 metric. Among them, APPS [29], AixBench [62], and ODEX [64] have limited use, and there have been no studies evaluating the latest LLMs on these benchmarks. Consequently, their pass@1 performance only remains around 50%, based on the state-of-the-art Codex series models available in 2022. However, at that time, HumanEval had a pass@1 of only 28.81% with Codex-12B [10], suggesting that these three benchmarks would perform much better on today's SOTA models, likely far exceeding 60%. Furthermore, APPS consists of varying levels of competitive programming problems, while AixBench and ODEX focus on simple coding tasks, neither of which fully reflect real-world development needs.

For the two benchmarks without a pass@1 metric, the MCoNaLa benchmark [63] focuses solely on statement-level code generation scenarios collected from Stack Overflow. In contrast, ComplexCodeEval [16] includes function-level tasks sourced from real and complex development environments in GitHub repositories. However, it lacks test cases needed to accurately assess the code generation accuracy of LLMs.

Notably, the NCB benchmark [13] 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, resulting in a pass@1 of 52.8% for GPT-4. In contrast, our benchmark includes only problems that LLMs can solve, 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.

Excluding the benchmarks mentioned above, the remaining benchmarks fall into two categories: those targeting algorithmic tasks for competitive programming [65], [68], [71], [70] and those focused on generating complex non-standalone functions or classes [66], [15], [14], [17], [67], [69]. CodeApex [65], LiveCodeBench [70], and XCoderEval [71] feature algorithm challenges from platforms like LeetCode and Codeforces, with pass@1 scores of 58.4%, 40.3%, and 30.5%, respectively. However, such competitive programming problems are rarely encountered in real-world development scenarios. For benchmarks involving complex, non-standalone functions and classes, low pass@1 scores are mainly due to the intricate dependencies inherent to these tasks. For instance, HumanEvo [66], EvoCodeBench [15], and DevEval [67] focus on complex function dependencies or repository-level dependencies, resulting in pass@1 scores of 27.0%, 20.7%, and 53.0% on GPT-4, respectively. Similarly, ClassEval [17], a benchmark of 100 manually created Python problems simulating real-world class generation scenarios, yielded a 37.0% pass@1 score on GPT-4.

Motivation 2: Although these benchmarks focus on practical coding tasks involving non-standalone function generation and class-level code generation, developers often avoid using LLMs for complex challenges due to low success rates [18]. Instead, they typically apply LLMs to simpler tasks that models like GPT-4 can handle.

To better align benchmarks with current LLM usage practices, we therefore introduce RealisticCodeBench—a benchmark designed to reflect the types of problems developers commonly tackle with LLMs.

III. RealisticCodeBench

Figure 1 provides an overview of the process for constructing RealisticCodeBench. 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 GPT-4, 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 417 programming problems across various languages, requires approximately 700 person-hours to complete.

A. Data Collection

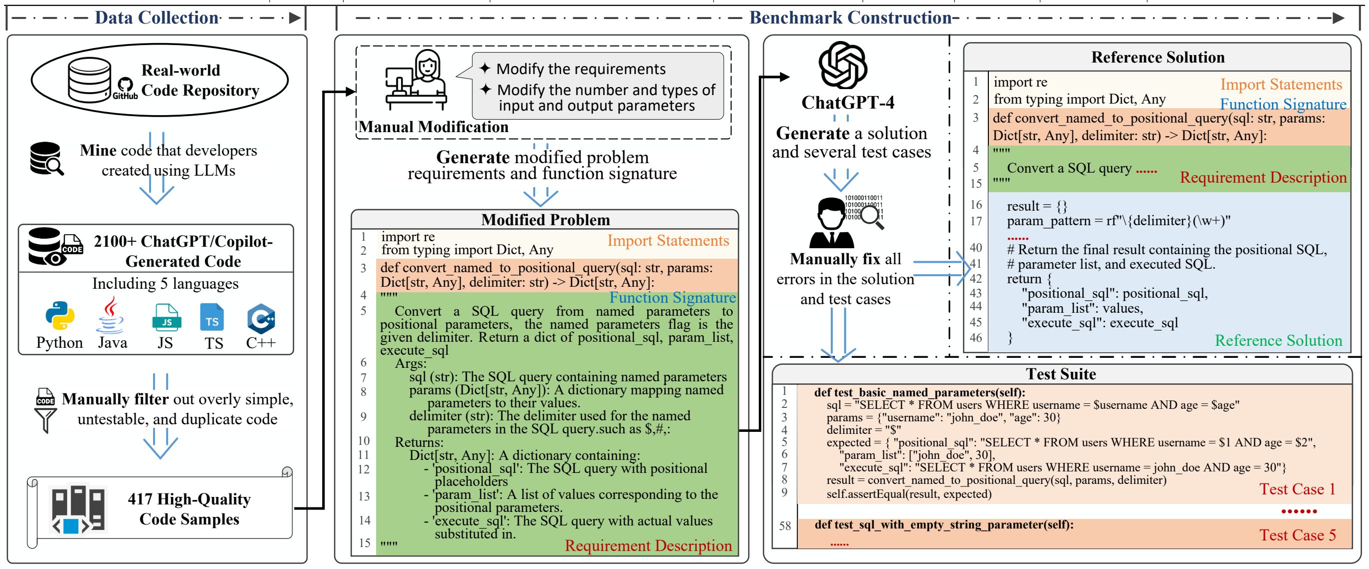
We first collect and filter high-quality code samples generated by ChatGPT/Copilot from GitHub.

ChatGPT/Copilot-Generated Code Collection. Yu et al. [19] 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 the approach of Yu et al. [19], 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 the collected samples, we prioritize repositories with high star ratings to source reputable code.

Suitable Programming Problems Filtering. Although we initially gather over 2,100 code samples generated by ChatGPT/Copilot from GitHub, not all of them are suitable for inclusion in our benchmark. We begin by manually filtering

TABLE II: The current 14 general code generation benchmarks with pass@1 scores below 60%, along with 2 benchmarks without pass@1 scores

Benchmark	Time	Language	Source	Granularity	#Tasks	#LOC	#Tokens	Pass@1
APPS [29]	2021	Python	Contest Sites	Function	5000	21.4	58	47.3 % (Code-davinci-002)
AixBench [62]	2022	Java	Open-sourced data	Function	175	-	-	49.1 % (aiXcoder XL)
MCoNaLa [63]	2023	Python	Conala	Statement	896	1	27.6	-
ODEX [64]	2023	Python	CoNaLa, mCoNaLa	Function	945	-	21.1	47.0 % (Code-davinci-002)
CodeApex [65]	2023	C++	Online OA platform	Function	476	˜12	˜50	58.4 % (GPT-4)
HumanEvo [66]	2024	Python, Java	PyPI, GitHub	Function, Repository	400	-	-	27.0 % (GPT-4)
CoderEval [14]	2024	Python, Java	GitHub	Function	230	30	108.2	21.0 % (GPT-3.5)
EvoCodeBench [15]	2024	Python	GitHub	Function, Repository	275	-	-	20.7 % (GPT-4)
ClassEval [17]	2024	Python	Manual	Class	100	45.7	123.7	37.0 % (GPT-4)
DevEval [67]	2024	Python	PyPI	Function, Repository	1874	-	-	53.0 % (GPT-4)
NCB [13]	2024	Python, Java	Online Services	Function	402	-	-	52.8 % (GPT-4)
MHPP [68]	2024	Python	Manual	Function	140	12.2	˜150.2	53.6 % (GPT-4)
BigCodeBench [69]	2024	Python	GitHub, Huagging face, Croissant	Function	1140	10	-	51.1 % (GPT-4o)
LiveCodeBench [70]	2024	Python	LeetCode, AtCoder, and CodeForces	Function	511	-	-	40.3 % (GPT-4)
XCodeEval [71]	2024	Multilingual	Open-sourced data	Function	20M	-	-	30.5 % (GPT-3.5)
ComplexCodeEval [16]	2024	Python, Java	PyPI, GitHub	Function	11081	35.9	278.8	-
ComplexCodeEval [16]	2024	Python, Java	PyPI, GitHub	Function	11081	35.9	278.8	-
ComplexCodeEval [16]	2024	Python, Java	PyPI, GitHub	Function	11081	35.9	278.8	-


Data Collection
Benchmark Construction
Real-world Code Repository
GitHub
Mine code that developers created using LLMs
2100+ ChatGPT/Copilot-Generated Code
Including 5 languages
Python
Java
JS
TS
C++
Manually filter out overly simple, untestable, and duplicate code
417 High-Quality Code Samples
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-4
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

out overly simplistic code (e.g., code that merely calculates the Euclidean distance between two points). Additionally, we exclude samples with solutions that are difficult to test. Finally, we review the remaining samples to eliminate tasks that are overly similar (e.g., multiple samples that validate whether a string is a valid email address), ensuring the benchmark includes a diverse and varied set of programming problems. After this filtering process, we are left with 207 refined Python code samples, 33 Java samples, 83 JavaScript samples, 51 TypeScript samples, and 47 C++ samples.

B. Benchmark Construction

Once we have collected code samples generated by ChatGPT and Copilot 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 GPT-4's advanced capabilities in code comment generation [72] to produce concise summaries for each code sample. This allows us to clearly understand the core functionality of the code, making it easier to modify the programming problems and prevent data leakage. Data leakage is a concern because many LLMs are pre-trained on code from GitHub, which can lead to inadvertent memorization of specific content [73], [74]. Consequently, these models may solve programming tasks by recalling solutions they encountered during pre-training. To mitigate this risk, we apply slight modifications to the requirements of the original code samples, similar to existing benchmark practices [32], ensuring that the code's intent and task complexity remain largely unchanged. Additionally, we modify the number and types of input and output parameters where feasible. In the adapted function signatures, we explicitly outline the implementation requirements for LLMs, specifying the function's objectives, input parameters, and return value constraints for each programming language. For instance, one GitHub project with 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 [4]. 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.* Here, the input parameters increase to three, and the output changes to a dictionary format.

Reference Solution Generation. We then use GPT-4 to generate solutions for each adapted programming problem by providing the problem description (including the function signature and requirement description) as prompts. Although GPT-4 is a highly capable tool, it can still produce incorrect code during generation. Therefore, each solution is meticulously reviewed by three expert 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 experts 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 GPT-4 to generate high-quality test cases for each adapted programming problem. The prompt provided to GPT-4 begins 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, we include the description of the programming problem and its reference solution in the prompt. After the test cases are generated, the same three expert programmers review and correct any issues related to formatting or outputs. If one of the programmers identifies any errors, they revise the test cases accordingly. The updated test cases are then reviewed by the other two experts to verify the accuracy of the corrections. Once this process is complete, the line and branch coverage for each function is reassessed. If coverage is still below 100%, one of the programmers manually writes additional test cases to strive for complete coverage, whenever feasible. These additional test cases are also reviewed by the other two experts to ensure their correctness.

Multi-Language Version Creation. To create multi-language versions of each programming problem in RealisticCodeBench, we leverage GPT-4's translation and adaptation capabilities to generate code in Python, Java, JavaScript, TypeScript, and C++. The translation process begins with a structured prompt that includes the original problem, its reference solution, and specific instructions to adapt the code while following 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++. Following initial translations, the same three expert programmers conduct a thorough review of each translated version and its associated test cases, making any necessary corrections or adjustments

[4]https://GitHub.com/jerber/fastgql/blob/4c308e742685e0a1cf4dc6d05f29cfbaea2d059a/fastgql/query_builders/sql/query_builder.py\#L464

to align with standard coding practices and language-specific nuances. If any language version lacks full line and branch coverage, additional test cases are created and reviewed to close the coverage gap.

Expert Review. We engage 13 experienced engineers to assess if the programming problems collected from GitHub could also represent coding tasks proprietary developers might address using LLMs (the three expert 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 insight 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 ask these engineers to assess whether the programming problems, including their multi-language versions, represent realistic development scenarios and whether developers would likely use LLMs to solve such problems. 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 involved reading a JSON file and converting it into a Python data type—a task achievable in a single line of Python code (*json.loads*()), making LLM assistance unnecessary. Another excluded task is 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 417 programming problems translated across multiple languages: 392 in Python, 376 in JavaScript, 372 in TypeScript, 339 in Java, and 353 in C++, each accompanied by reference solutions and test cases. These 417 problems include 401 function-based tasks and 16 class-based code generation tasks. As indicated in Table I, the average token length for problem input information, which includes the requirements and function signature, is 95.2 across the five languages. The average LOC in the reference solutions for these languages are 30.6. 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.

Based on functionality, these 417 problems cover nine distinct domains, as shown in Table III: 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. This diversity also introduces a range of complex input data types, classified into eight categories, including strings, sequences, numbers, matrices, dictionaries, functions, complex data, and files. The first six are common basic data types, where sequences represent ordered data structures like arrays and tuples, numbers include integers, floating points, and boolean numbers represented by 0 or 1, and complex data covers unique data types specific to different languages, such as DataFrame in Python, Objects in JavaScript/TypeScript, and Structs in C++. Files include data files used in development in various formats (e.g., csv, xlsx, json, jsonl, xml, and yaml), as well as image files and office files like pdf, docx, and doc. This diverse input not only reflects the complexity found in real-world development but also enhances the broad applicability of the benchmark test.

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 [17]. Table IV 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-V2.5) 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 models like GPT-4 and GPT-3.5-turbo, we use the OpenAI API interface (accessed in September 2024). For DeepSeek-V2.5 [5], we rely on the DeepSeek API interface (also accessed in September 2024), as this model, while open-sourced, requires eight 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 selecting the token with

[5]DeepSeek-V2.5 is an upgraded version that combines DeepSeek-V2-Chat and DeepSeek-Coder-V2-Instruct, integrating the general and coding abilities. However, the official website has not disclosed the parameter count for DeepSeek-V2.5. Since the parameter count for DeepSeek-V2 is 236B, we infer that DeepSeek-V2.5 likely also has 236 billion parameters.

TABLE III: The types of the 417 programming problems

Topic	Description	Examples	#T
Data Structures and Algorithms	Utilizing common data structures for processing non-text data, along with fundamental algorithms	Implementation of linked lists, queues and other data structures to process non-text data, sorting algorithms, search algorithms, dictionary lists into list dictionaries	106
Text Processing	Perform formatting conversions on strings, regular expression matching, content extraction, and other text operations.	Regularly match specific strings, convert string types to other types, extract content between specified characters	100
File Handling	Read and write files, modify content, convert encodings, change file types, and handle file retrieval in directories.	Convert JSON files to YAML, modify CSV file content based on specific rules, filter files in directories	67
Mathematical Problems and Scientific Computing	Programming for mathematical problems, matrix and vector operations, data statistics, and related scientific computing.	Matrix multiplication, BMI calculation, deflection angle conversion, calculus computations	67
Date and Time Processing	Perform date format conversions, time calculations, and time unit conversions.	Convert timestamps to specific timezone formats, convert dates to milliseconds	32
Data Visualization and Graphic Applications	Involve data display and image file processing	Convert images to grayscale, color statistics of images, generate gradient colors	20
Network Programming	Involve IP address and network interfaces, URL requests, and domain name resolutions	Obtain local IP address and port, parse URLs to extract specified parameters, extract domain levels	12
Frontend Development	Parse and process web content HTML, CSS, modify styles, compress HTML	Remove page ads, implement specific element highlighting, switch page themes	8
Security	Involve data encryption and decryption, complexity checks of data, etc	Encrypt user passwords, check if input data meets security requirements, decrypt specific data	5

TABLE IV: The overview of the 12 evaluated LLMs

	Model Name	Organization	Sizes	Release Time	Open-Source
General	GPT-4 [6]	OpenAI	-	2023	
	GPT-3.5 [20]	OpenAI	-	2022	
	DeepSeek-V2.5 [7]	DeepSeek	236B	2024	✓
	Llama 3.1 [8]	Meta	8B	2024	✓
	Phi-3 [21]	Microsoft	7B	2024	✓
	Mistral [22]	Mistral AI	7B	2024	✓
	ChatGLM [23]	THUDM	6B	2024	✓
Coding	CodeGeex4 [9]	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	✓

the highest probability at each step, providing a deterministic evaluation of the models' performance. Additionally, we use nucleus sampling to generate 10 code solutions (n=10) per task, with a top-p value of 0.95 and a temperature of 0.8, to explore the models' ability to produce diverse outputs.

Following established practices in code generation evaluation, we employ the pass@k metric to assess the functional correctness of generated code. For each programming problem, n code solutions are generated by LLMs, and k solutions are randomly selected for testing against reference test cases. The pass@k score measures the percentage of programming problems, among the problems in RealisticCodeBench, for which 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 the unbiased estimator of pass@3 and pass@5 implemented in HumanEval [10], ensuring reliable and consistent evaluations of LLM performance across our benchmark.

V. Experimental Results

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

Tables V present 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 achieves the highest average pass@1 across the five languages, with an average pass@1 score of 67.27%, followed by DeepSeek-V2.5 and GPT-3.5, which achieves an average pass@1 of 66.08% and 58.83%, respectively. GPT-4's average pass@1 surpasses that of DeepSeek-V2.5 by 1.19%. In Python, GPT-4 leads by a margin of 4.84%, yet the gap is much smaller in Java, JavaScript, and C++ (from 0.47% to 2.37%), with DeepSeek-V2.5 even outperforming GPT-4 in TypeScript by 3.22%. Compared to GPT-3.5, DeepSeek-V2.5 achieves a 7.25% higher average pass@1. This performance trend remains consistent for pass@3, while for average pass@5, DeepSeek-V2.5 slightly surpasses GPT-4. Overall, these results highlight the superior code generation capabilities of GPT-4, DeepSeek-V2.5, and GPT-3.5. As an open-source model, DeepSeek-V2.5 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 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 48.14%, with DeepSeek-Coder following closely at 40.61%. Notably, the difference between CodeGeeX4 and GPT-3.5 in the programming languages is not large, ranging from 4.81% in JavaScript to 14.83% in Python.

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,

TABLE V: 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	**83.46**	**60.64**	**69.62**	60.21	**62.46**	**67.27**
	GPT-3.5	70.30	52.69	60.33	54.59	56.24	58.83
	DeepSeek-V2.5	78.62	60.17	67.25	**63.43**	60.96	66.08
	Llama 3.1	51.83	25.14	41.54	34.05	22.50	35.01
	Phi-3	45.15	24.52	46.20	38.42	26.08	36.07
	Mistral	33.57	24.01	32.88	20.81	22.97	26.84
	ChatGLM	26.50	12.23	23.45	18.25	9.02	17.89
Coding	CodeGeex4	**55.47**	**40.47**	**55.52**	**45.94**	**43.34**	**48.14**
	DeepSeek-Coder	47.73	34.41	42.62	40.77	37.53	40.61
	StarCoder2	45.92	31.17	40.48	36.20	35.53	37.86
	CodeGen2.5	41.76	27.64	40.26	36.42	20.31	33.27
	CodeLlama	44.30	27.97	38.60	36.08	32.24	35.83
				Pass@3			
General	GPT-4	85.80	65.31	**78.90**	69.08	66.31	**75.02**
	GPT-3.5	74.43	56.40	68.93	57.99	60.16	63.58
	DeepSeek-V2.5	**89.94**	**66.72**	77.86	**72.45**	**66.79**	74.75
	Llama 3.1	54.06	28.32	47.17	37.37	25.32	38.44
	Phi-3	47.20	26.08	47.12	40.56	29.15	38.02
	Mistral	36.40	25.78	36.99	22.19	24.02	29.07
	ChatGLM	28.72	13.64	25.56	20.12	10.26	19.66
Coding	CodeGeex4	**58.06**	**43.20**	**56.66**	**47.29**	**47.20**	**52.30**
	DeepSeek-Coder	50.12	36.24	45.93	43.44	39.12	44.65
	StarCoder2	50.97	33.24	45.06	40.24	38.66	41.63
	CodeGen2.5	43.42	28.46	41.34	38.08	23.89	35.03
	CodeLlama	46.24	29.04	40.12	38.26	35.18	38.41
				Pass@5			
General	GPT-4	86.62	**70.54**	**84.53**	73.25	**70.02**	76.99
	GPT-3.5	76.02	61.18	74.02	62.25	64.44	67.58
	DeepSeek-V2.5	**90.79**	69.28	80.19	**76.33**	69.23	**77.16**
	Llama 3.1	55.50	32.12	48.32	38.04	27.43	40.28
	Phi-3	48.76	27.26	49.06	42.18	33.24	40.10
	Mistral	37.62	27.32	38.75	25.31	27.21	31.24
	ChatGLM	29.35	15.16	28.32	23.48	12.60	21.78
Coding	CodeGeex4	**61.64**	**45.62**	**63.33**	**50.35**	**49.12**	**54.01**
	DeepSeek-Coder	51.90	40.18	47.28	45.92	41.48	45.35
	StarCoder2	51.56	35.73	49.10	43.60	40.57	44.11
	CodeGen2.5	43.66	30.27	43.22	41.64	25.46	36.85
	CodeLlama	48.25	31.19	42.37	40.09	36.20	39.62

the distinct segments unique to each model highlight their specific strengths. GPT-4 and DeepSeek-V2.5 stand out with the largest unique areas, demonstrating their superior performance in solving programming problems that other models cannot, which underscores their stronger performance in code generation across the five languages.

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 achieving an 83.46% 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's pass rate rises from 67.27% at pass@1 to 76.99% at pass@5, DeepSeek-V2.5 improves from 66.08% to 77.16%, and GPT-3.5 from 58.83% to 67.58%. 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 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 achieves the highest average pass@1 across five languages, closely followed by the open-source model DeepSeek-V2.5. Among the smaller-parameter open-source models, CodeGeeX4 stands out with strong performance, showing a small gap compared to GPT-3.5.

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

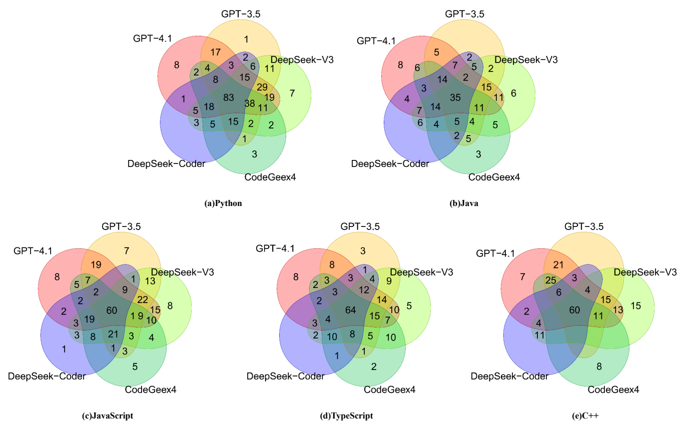


Fig. 2: The number of problems solved by GPT-4, GPT-3.5, DeepSeek-V2.5, CodeGeex4, and DeepSeek-Coder

In this section, we compare the pass@1 performance of 12 LLMs on RealisticCodeBench and HumanEval, excluding MBPP due to limited data availability (with only four models reporting pass@1 results on MBPP). We also 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 pass@1 results for GPT-4, GPT-3.5, DeepSeek-Coder, and CodeLlama are sourced from the DeepSeek-Coder technical report published in January 2024 [27]. Results for Llama3.1 [8], Phi-3 [21], Mistral [75], ChatGLM [23], StarCoder2 [25], and CodeGen2.5 [24] are drawn from their respective technical reports, while results for DeepSeek-V2.5 and CodeGeeX4 are based on their official evaluations on Huggingface and GitHub [6]. The scatter plot includes a green dashed line representing a linear fit and a light

[6]https://huggingface.co/deepseek-ai/DeepSeek-V2.5, https://GitHub.com/THUDM/CodeGeeX4

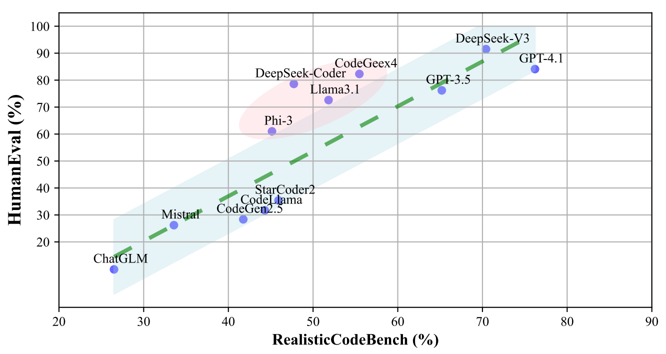


Fig. 3: The performance comparison of pass@1 for 12 LLMs between HumanEval and RealisticCodeBench

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.

However, despite DeepSeek-V2.5 outperforming GPT-4 on HumanEval, GPT-4 achieves higher pass@1 results on RealisticCodeBench. Additionally, 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 55.47% on RealisticCodeBench; Llama 3.1 decreases from 72.6% to 51.83%; DeepSeek-Coder falls from 78.6% to 47.73%; and Phi-3 declines from 61.0% to 45.15%.

Several factors may explain this phenomenon. First, some LLMs' training sets might be overly optimized for HumanEval-style problems. Previous studies [70], [13], [66], [12] 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, GPT-4 [6] reported that 25% of HumanEval had been contaminated in their pre-training corpus. Additionally, contamination may arise from instruction fine-tuning datasets [12], as noted by Phi [76], [77], 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 generates incorrect code within five attempts. In instances where GPT-4 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 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 Parameter Handling and Edge Case Coverage:** A common issue in GPT-4's generated code involves inadequate handling of parameter types, missing boundary conditions, or incomplete exception handling. For instance, in Example 1, GPT-4 generates code for Dijkstra's algorithm without accounting for negative-weight edges, which are incompatible with this algorithm. Dijkstra's algorithm is fundamentally unsuitable for graphs with negative weights, yet the generated code lacks a mechanism to detect and handle this limitation. By incorporating one or two specific test cases in the prompt, GPT-4 generates a corrected version of the code, highlighting its dependency on explicit prompt guidance for handling such constraints.

(2) **API Misuse and Incorrect Imports:** Another recurring issue involves incorrect API usage or errors in package imports. In Example 2, GPT-4 attempts to use the *datetime.datetime.now*() function without properly importing the necessary class from the datetime module. Instead of using *from datetime import datetime* to correctly import the required functionality, GPT-4 incorrectly accesses *datetime.datetime*, resulting in an API error. By providing GPT-4 with the specific error message returned during code execution, we prompt the model to correct the import statement, demonstrating that runtime error feedback can improve the accuracy of generated code.

(3) **Incorrect Mathematical Formula Application:** For problems involving mathematical calculations, GPT-4 occasionally misapplies formulas when the prompt does not explicitly provide the correct one. For example, in Example 3, GPT-4 is tasked with calculating an integral using Simpson's Rule over the interval [a, b]. However, in the absence of an explicit formula in the prompt, GPT-4 incorrectly uses another integration method. When we provide the Simpson's Rule formula directly in the prompt, GPT-4 generates the correct solution, emphasizing the need for precise mathematical instructions when dealing with formula-based computations.

Answer to RQ3: Common errors in GPT-4's code generation on RealisticCodeBench include insufficient handling of edge cases and parameter robustness, API misuse or incorrect imports, and the misapplication of mathematical formulas.

VI. Discussion

A. Implications

Our results highlight several implications for LLM researchers and practitioners.

Unlike widely-used benchmarks such as HumanEval and MBPP, which focus primarily 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 ClassEval—which are designed to test the upper bounds of LLM capabilities—our benchmark offers a complementary perspective. While we recognize the value of these other benchmarks, ours serves as a practical addition, providing insights from a real-world LLM usage perspective. 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. [18], where 41% of developers express fears about LLMs accessing private codebases, our findings indicate that open-source models like DeepSeek-V2.5 and CodeGeeX4-9B offer a viable and privacy-conscious alternative. The performance differential between DeepSeek-V2.5 and GPT-4, with an average pass@1 gap of only 1.19%, suggests that DeepSeek-V2.5, 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-3.5 on some programming languages; for instance, in Python, it achieves a pass@1 rate of 55.47% and a pass@5 rate of 61.64%, narrowing the accuracy gap with GPT-3.5 (pass@1 of 70.30%) to only 8.66% 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-V2.5 is recommended, while CodeGeeX4-9B is advised for privacy-conscious developers or smaller companies operating within tighter budget constraints**.

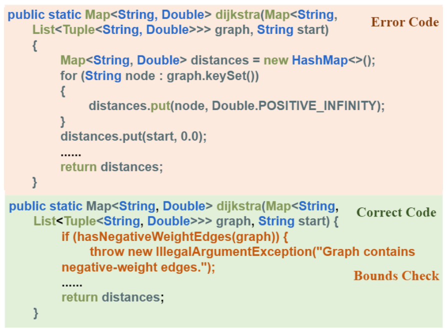
The error case analysis in Section V-C underscores **the need for research focused on enhancing LLM robustness in handling boundary conditions, domain-specific formulas, and resolving issues related to API misuse and incorrect imports**. For tasks requiring robust parameter handling and comprehensive edge case coverage, developers are advised to include specific test cases within the prompt to highlight these aspects effectively. Incorporating iterative feedback loops, such as feeding runtime error messages back to the model, can further improve accuracy in subsequent attempts and help avoid API misuse and incorrect import issues. 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. [18] 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 study focuses exclusively on LLM-generated code from high-star GitHub repositories, aiming to reflect open-source developers' use of LLMs. We lack access to proprietary code generated by developers in private companies, which limits the generalizability of our findings to broader industry applications. To address this, we invite 13 developers from major tech companies (e.g., Microsoft, Huawei, ByteDance, Tencent, Alibaba, Bilibili, Meituan) as well as smaller IT companies to assess our benchmark's relevance. These developers evaluate whether the programming problems in our benchmark represent realistic development scenarios and if LLMs would practically be used to generate solutions for such problems. However, the limited number of programming problems (417) may not fully capture the diversity of real-world coding tasks. Additionally, due to the extensive manual effort involved (about 700 person-hours), we currently limit our benchmark to this scale. 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.

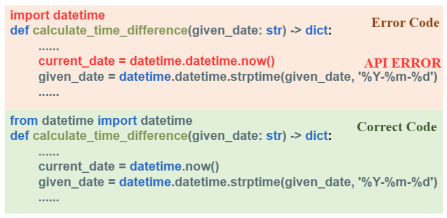
To mitigate potential risks of data leakage, we adapt the programming problems derived from GitHub code, altering the types and quantities of input/output parameters. We calculate the Levenshtein distance between the original GitHub code and the LLM-generated code, finding substantial differences. For example, the Levenshtein distance between the original



```
public static Map<String, Double> dijkstra(Map<String,
    List<Tuple<String, Double>>> graph, String start)
    {
        Map<String, Double> distances = new HashMap<>();
        for (String node : graph.keySet())
        {
            distances.put(node, Double.POSITIVE_INFINITY);
        }
        distances.put(start, 0.0);
        ......
        return distances;
    }
```

```
public static Map<String, Double> dijkstra(Map<String,
    List<Tuple<String, Double>>> graph, String start) {
        if (hasNegativeWeightEdges(graph)) {
            throw new IllegalArgumentException("Graph contains
        negative-weight edges.");
        ......
        return distances;
    }
```

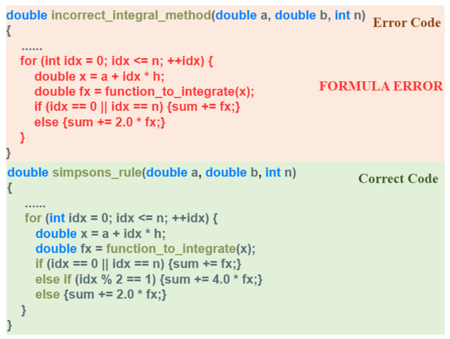
(a) Example 1



```
import datetime
def calculate_time_difference(given_date: str) -> dict:
        ......
        current_date = datetime.datetime.now()
        given_date = datetime.datetime.strptime(given_date, '%Y-%m-%d')
        ......
```

```
from datetime import datetime
def calculate_time_difference(given_date: str) -> dict:
        ......
        current_date = datetime.now()
        given_date = datetime.datetime.strptime(given_date, '%Y-%m-%d')
        ......
```

(b) Example 2



```
double incorrect_integral_method(double a, double b, int n)
{
    ......
   for (int idx = 0; idx <= n; ++idx) {
      double x = a + idx * h;
      double fx = function_to_integrate(x);
      if (idx == 0 || idx == n) {sum += fx;}
      else {sum += 2.0 * fx;}
   }
}
```

```
double simpsons_rule(double a, double b, int n)
{
    ......
    for (int idx = 0; idx <= n; ++idx) {
      double x = a + idx * h;
      double fx = function_to_integrate(x);
      if (idx == 0 || idx == n) {sum += fx;}
      else if (idx % 2 == 1) {sum += 4.0 * fx;}
      else {sum += 2.0 * fx;}
    }
}
```

(c) Example 3

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

GitHub Python code and the GPT-4-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, comprising 392 Python problems, 376 JavaScript problems, 372 TypeScript problems, 339 Java problems, and 353 C++ problems, represents a wide array of coding challenges sourced from high-star GitHub repositories, closely reflecting developers' everyday coding needs. Experimental evaluations of 12 general-purpose and code-specific LLMs reveal that, while GPT-4 achieves the highest average pass@1, open-source models like DeepSeek-V2.5 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.

VIII. Data Availability

Our RealisticCodeBench benchmark, along with the code used to evaluate the performance of the 12 LLMs discussed in this paper, is available at [78].

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, G. Li, and Z. Jin, "Codep: grammatical seq2seq model for general-purpose code generation," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 188–198.

[3] L. Guo, Y. Wang, E. Shi, W. Zhong, H. Zhang, J. Chen, R. Zhang, Y. Ma, and Z. Zheng, "When to stop? towards efficient code generation in llms with excess token prevention," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1073–1085.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] J. 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] X. Yu, L. Liu, X. Hu, J. W. Keung, J. Liu, and X. Xia, "Where are

large language models for code generation on github?" *arXiv preprint arXiv:2406.19544*, 2024.

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

[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] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

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

[49] 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."

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

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

[59] 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.

[60] L. Zhong, Z. Wang, and J. Shang, "Ldb: A large language model debugger via verifying runtime execution step-by-step," *arXiv preprint arXiv:2402.16906*, 2024.

[61] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," *arXiv preprint arXiv:2312.13010*, 2023.

[62] 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.

[63] 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.

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

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

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

[74] 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.

[75] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, “Mixtral of experts,” *arXiv preprint arXiv:2401.04088*, 2024.

[76] 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.

[77] 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.

[78] Anonymity, “Supplemental materials,” https://figshare.com/s/835f0ada67b8623c64d4, 2025.