Towards More Realistic Evaluation of Large Language Models for Code Generation

ANONYMOUS AUTHOR(S)

The rapid advancements in large language models (LLMs) have led to widespread interest in using these models as programming assistants for code generation. However, current benchmarks may not adequately reflect the tasks that developers actually use LLMs for. This paper investigates the gap between real-world developer needs and existing benchmarks by analyzing survey data, evaluating class-level code generation challenges, and proposing a more realistic benchmark for LLMs. We construct a new benchmark based on real-world usage of LLM-generated code from GitHub, focusing on simpler tasks such as data processing and frontend code. We validate our approach by involving experienced engineers in the construction process and conducting a large-scale survey among developers. Our findings suggest that while LLMs struggle with complex tasks like class-level code generation, they are more commonly used for simpler tasks in practice.

Additional Key Words and Phrases: Code Generation, Large Language Model, Benchmark, GitHub

1 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. Recent advances in Large Language Models (LLMs) have further accelerated progress in this field. Various LLMs, including GPT-4, DeepSeek, CodeLlama, and CodeGeeX, 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 [9] and MBPP [3]. Reporting performance on these benchmarks has seemingly become mandatory for a model to be considered competitive in code generation [34]. Indeed, nearly all new LLMs released in 2023-2024 highlight code generation results on one or both of these benchmarks. While these benchmarks have been widely used and provide valuable insights, the programming problems they contain are largely algorithmic and basic programming problems and do not fully reflect the challenges of real-world coding [54]. To address this, more complex benchmarks—such as CoderEval, EvoCodeBench, ComplexCodeEval, and ClassEval—have been developed to assess LLM performance on more challenging, practical coding tasks collected from real-world 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, it is important to note that developers currently do not typically rely on LLMs for complex coding tasks such as class-level code generation. One major reason for this is the relatively low success rate of LLMs on these more difficult benchmarks. For example, ChatGPT-3.5 achieves a pass@1 rate of only 30% on ClassEval, which can discourage developers from using LLMs for such sophisticated code generation tasks. A large-scale survey conducted by Liang et al. [30] 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 ClassEval and CoderEval, 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., ChatGPT) 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 on a daily basis and create benchmarks based on these practical use cases.

to better align benchmarks with developer practices, we propose a more realistic evaluation framework that focuses on these simpler tasks.

We propose NATURALCODEBENCH, a benchmark that aligns with real-world applications, comprising 402 problems in Python and Java across 6 domains. We open source 140 problems (70 Python, 70 Java) as the development set of NCB for research purposes, but keep the 262 problems of the test set closed to avoid contamination.

We introduce a semi-automated pipeline for the construction of code synthesis benchmarks, which significantly reduces time and manpower costs without compromising the quality of test cases. Comparative experiments reveal that our semi-automated pipeline can quadruple the construction speed of the evaluation framework

We systematically benchmark the code generation capabilities of 39 LLMs using NCB. Besides quantitative evaluation, we carry out a deep insight into the present stage of development in LLMs for code generation, and outline potential pathways for future progress.

2 BACKGROUND AND RELATED WORK

2.1 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 and GPT-3.5, both of which have demonstrated significant capabilities across a wide range of tasks. Additionally, other general-purpose models like Llama 3.1, Phi-3, Mistral, and ChatGLM have gained attention for their versatility. 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, StarCoder, CodeLlama, DeepSeek-Coder, and CodeGeeX. 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 [33]. 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.

2.2 Code Generation Benchmarks

Literature Search: To understand the progress of code generation benchmarks, we conduct a literature search covering publications from 2021 to 2024 using a forward snowballing approach [46] [*]. 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 [18], HumanEval [9], and MBPP [3]). Although earlier code generation benchmarks, such as Concode [21] and JuICe [1], were proposed before 2021, they mainly focused on evaluating smaller deep learning models, like LSTM and Transformer, rather than LLMs. Moreover, these earlier 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.

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

Therefore, our search process begins by gathering all papers that cite APPS [18], HumanEval [9], and MBPP [3] 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 59 code generation benchmark articles found in the literature.

The benchmarks identified can be broadly classified into two categories. The first category, comprising 22 papers [3, 5, 8, 15, 16, 20, 24, 25, 28, 29, 32, 32, 35, 38, 39, 41–43, 47, 49, 52, 53], focuses on domain-specific code generation abilities, such as generating security code [15, 39], VHDL code [43], bioinformatics code [42], Verilog code [32], data science code [3, 8, 20], AI code [49], object-oriented code in Java [5], parallel code [35], Infrastructure-as-Code (IaC) programs [24], 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 in the tables . Tables 1 and 2 provide an overview of 28 selected benchmarks, including details such as the year of introduction ("Time"), target programming language ("Language"), the source of programming problems ("Source"), target code granularity ("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 ChatGPT) in terms of "pass@1". Table 1 lists benchmarks where the best LLM's pass@1 exceeds 60%, while Table 2 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 1. The existing general code generation benchmarks (pass@1 > 60%)

Benchmark	Time	Language	Source	Granularity	#Tasks	#LOC	#Tokens	Pass@1
HumanEval [9]	2021	Python	Manual	Function	164	11.5	24.4	90.2 % (GPT-4o)
MBPP [3]	2021	Python	Manual	Function	974	6.8	24.2	83.5 % (GPT-4)
MultiPL-E [7]	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 [2]	2022	Multilingual	HumanEval	Function	164	11.5	24.4	90.2 % (GPT-4o)
MBXP [2]	2022	Multilingual	MBPP	Function	974	6.8	24.2	83.5 % (GPT-4)
HumanEval+ [31]	2023	Python	HumanEval	Function	164	11.5	24.4	76.2 % (GPT-4)
HumanEval-X [57]	2023	Multilingual	HumanEval	Function	820	11.5	24.4	90.2 % (GPT-4o)
StudentEval [4]	2023	Python	Manual	Function	48	-	-	63.6 % (StarChat-Alpha)
EvoEval [48]	2024	Python	HumanEval	Function	828	-	-	62.0 % (GPT-4)
ENAMEL [37]	2024	Python	HumanEval	Function	142	11.5	24.4	83.1 % (GPT-4)
ScenEval [36]	2024	Java	W3Resources, Stack Overflow, Textbooks	Class, Function, Statementn	12864	1-50	-	75.6 % (ChatGPT)
RACE [56]	2024	Python	HumanEval+, MBPP+, ClassEval, LeetCode	Class, Function	923	-	-	70.1 % (GPT-4-o1-mini)
LBPP [34]	2024	Python	Manual	Function	161	-	-	64.0 % (Claude-3.5-Sonnet)

Table 1 includes a total of **13 benchmarks where the pass@1 exceeds 60%** in LLM evaluations. Among them, 4 benchmarks [3, 4, 9, 34] were manually created, 2 [36, 56] were created by collecting open-source data, while 7 [2, 7, 31, 37, 48, 56, 57] are adapted or extended versions of HumanEval, and 3 [2, 7, 56] are adapted or extended versions of MBPP. Chen et al. [9] et al. first proposed the HumanEval dataset, which consists of 164 hand-written 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. [7], Athiwaratkun et al. [2], and Zheng et al. [57] 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) [34], with pass@1 reaching 90.2% for GPT-4o when employing a model debugger strategy [58], and 76.5% for GPT-4 in a zero-shot setting, as shown on the leaderboard [†]. Furthermore, Liu et al. [31] 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. [48] 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. [37] 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. [3], 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. [7] and Athiwaratkun et al. [2] 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 [19], as shown on the leaderboard [‡].

Among other manually created datasets, Babe et al. [4] 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. [34] 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. [36] 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. [56] 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.

Motivation 1. Table 1 highlights that many widely-used benchmarks 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 when using LLMs for code generation. Thus, while current benchmarks offer insights into a model's performance on fundamental programming skills, they lack the ability to evaluate LLMs on practical, day-to-day tasks encountered by developers in real-world applications.

From Table 2, there are a total of **14 benchmarks where the pass@1 is below 60%** in existing LLM evaluations, and 2 benchmarks [13, 44] without a pass@1 metric. Among them, APPS [18], AixBench [17], and ODEX [45] have a pass@1 of only about 50% because their evaluations are based on the best LLMs available in 2022, such as the Codex series. However, at that time, HumanEval

[†]https://paperswithcode.com/sota/code-generation-on-humaneval
[‡]https://paperswithcode.com/sota/code-generation-on-mbpp

had a pass@1 of only 28.81% with Codex-12B [9], suggesting that these three benchmarks would perform much better on today's SOTA models, likely far exceeding 60%. For the two benchmarks without a pass@1 metric, the MCoNaLa benchmark [44] only evaluates statement-level code generation scenarios collected from Stack Overflow. In contrast, ComplexCodeEval [13] includes function-level tasks from real and complex development environments collected from open-source repositories, with an average input length of 278.8 tokens per problem. However, it lacks test cases to accurately assess the accuracy of LLMs' code generation capabilities.

Excluding the benchmarks mentioned above, the remaining benchmarks can be categorized into three types: one [11, 26, 27, 55, 59] focuses on generating complex non-standalone functions or classes, another [10, 14, 23] targets algorithmic tasks, and the third [50, 54] is more aligned with real-world development tasks faced by actual developers. For benchmarks involving generating complex non-standalone functions or classes, the input information to LLM is often quite large (for example, the average input for classEval [11] is 123.7 tokens). This places significant demands on the LLM's comprehension and reasoning abilities, resulting in pass@1 performance still falling below 60% even with current SOTA models. Moreover, developers typically do not prefer to input such extensive prompts into an LLM for everyday tasks. Zheng et al. [55] et al. constructed the repository-level dataset HumanEvo by extracting pull requests from GitHub and selecting methods suitable for generation tasks based on changes in the repository. Due to the complex dependencies of most selected methods, only 27.0% achieved pass@1 on GPT-4. Similarly, Li et al. [26, 27] et al. introduced the repository-level datasets EvoCodeBench and DevEval. These datasets are derived from GitHub and include methods with real and complex dependency scenarios, which are then further modified to create evaluation problems. Compared to EvoCodeBench, DevEval has lower dependency complexity, with only about 70% of tasks involving dependencies. As a result, DevEval achieves a higher pass@1 score of 53.0% on GPT-4, while EvoCodeBench only reaches 20.7%. Zhuo et al. [59] et al. developed BigCodeBench, a function-level Python benchmark consisting of 1,140 complex instructions that involve multiple method calls, targeting non-standalone function scenarios. Due to its intricate instructions and dependencies, it achieves only a 51.1% pass@1 on GPT-4. In class-level generation tasks, Du et al. [11] et al. introduced ClassEval, which comprises 100 manually created Python problems simulating real-world class generation scenarios. With an average of 123.7 input tokens per task and complex dependencies among classes, methods, and fields, ClassEval challenges the reasoning ability of LLMs, resulting in a mere 37.0% pass@1 on GPT-4. Additionally, complex algorithm tasks are constructed to directly assess the LLMs' problem-solving capabilities, although developers often prefer more general problems in real-world settings. CodeApex [14], LiveCodeBench [22], and XCoderEval [23] gather algorithm problems from platforms like LeetCode or open-source data, yielding diverse difficulty levels and pass@1 scores of 58.4%, 40.3%, and 30.5%, respectively, on their respective models. For MHPP [10], a set of 140 handwritten problems more challenging than HumanEval and MBPP, focuses on algorithm tasks and achieves a pass@1 of 53.6% on GPT-4 due to its difficulty. Lastly, in benchmarks targeting real-world development scenarios, CoderEval [50] created 230 real development tasks in Java and Python by collecting high-star projects from GitHub. Each task includes some context dependencies, yet excessive context dependencies lead to excessively long token inputs (108.2) during generation, and the complexity of the problems results in a pass@1 of only 21.0 % on GPT-3.5.

NCB [54] aligns more closely with our goals, gathering real developer requests through online services to generate the corresponding functions and assess whether LLMs can effectively solve practical problems. NCB involves Python and Java, and the problems span multiple scenarios with a certain level of complexity, resulting in an average pass@1 of 52.8% on GPT-4.

We see that many benchmarks are focused on complex and challenging tasks, such as non-standalone function generation and class-level code generation, where the pass@1 performance of even the best models is relatively low. This suggests that developers may be hesitant to rely on LLMs for generating such intricate code, as the accuracy is insufficient to make these tools practical for more complex programming problems. Furthermore, even though some benchmarks collect real developer content from GitHub or other online services, this doesn't necessarily reflect what developers would generate when using LLMs. We should explore more deeply what problems developers truly want to achieve with LLMs and use these as the basis for our benchmark tasks [51].

The motivation of our paper is to address the mismatch between current code generation benchmarks and real-world coding tasks. The limitations include benchmarks in Table 1 focusing on overly simplistic, algorithmic problems that do not capture the complexity of actual development, while those in Table 2 contain highly complex tasks with low model success rates, discouraging practical use of LLMs. Your work aims to bridge this gap by creating benchmarks that better reflect the everyday coding tasks developers face, providing a more realistic evaluation of LLM capabilities.

Our Motivation. The primary limitation of existing code generation benchmarks is that they either focus on algorithmic and basic programming problems tasks (as seen in Table 1) that do not reflect real-world coding challenges, or they present highly complex problems (as in Table 2) where LLMs perform poorly, making them impractical for developers to currently use for code generation. Our paper is to create benchmarks that bridge this gap, focusing on practical, real-world code generation tasks that developers encounter daily, thereby providing more meaningful assessments of LLM performance.

Table 2. The existing general code generation benchmarks (pass@1 < 60%)

Benchmark	Time	Language	Source	Granularity	#Tasks	#LOC	#Tokens	Pass@1
APPS [18]	2021	Python	Contest Sites	Function	500	21.4	58	47.3 % (Code-davinci-002)
AixBench [17]	2022	Java	Open-sourced data	Function	175	-	-	49.1 % (aiXcoder XL)
MCoNaLa [44]	2023	Python	Conala	Statement	896	1	27.6	-
ODEX [45]	2023	Python	CoNaLa, mCoNaLa	Function	945	-	21.1	47.0 % (Code-davinci-002)
CodeApex [14]	2023	C++	Online OA platform	Function	476	~12	~50	58.4 % (GPT-4)
HumanEvo [55]	2024	Python, Java	PyPI, Github	Function, Repository	400	-	-	27.0 % (GPT-4)
CoderEval [50]	2024	Python, Java	Github	Function	230	30	108.2	21.0 % (GPT-3.5)
EvoCodeBench [26]	2024	Python	Github	Function, Repository	275	-	-	20.7 % (GPT-4)
ClassEval [11]	2024	Python	Manual	Class	100	45.7	123.7	37.0 % (GPT-4)
DevEval [27]	2024	Python	PyPI	Function, Repository	1874	-	-	53.0 % (GPT-4)
NCB [54]	2024	Python, Java	Online Services	Function	402	-	-	52.8 % (GPT-4)
MHPP [10]	2024	Python	Manual	Function	140	12.2	~150.2	53.6 % (GPT-4)
BigCodeBench [59]	2024	Python	Github, Huagging face, Croissant	Function	1140	10	-	51.1 % (GPT-4o)
LiveCodeBench [22]	2024	Python	LeetCode, AtCoder, and CodeForces	Function	511	-	-	40.3 % (GPT-4)
XCodeEval [23]	2024	Multilingual	Open-sourced data	Function	20M	-	-	30.5 % (GPT-3.5)
ComplexCodeEval [13]	2024	Python, Java	PyPI, Github	Function	11081	35.9	278.8	-

3 REALISTICCODEBENCH

The overview of construing RealisticCodeBench is shown in Figure 2. The pipeline of constructing RealisticCodeBench consists of four steps: 1) collecting and filtering high-quality code generated by ChatGPT/Copilot from GitHub (Section 2.1) 2) constructing a complete evaluation framework (adapt problem requirements, writing reference solutions and code test cases) through a semi-automated pipeline (Section 2.2) 3) translating all problems and instructions to produce multilingual versions.

3.1 Data Collection

We first adopt a multi-step process to collect and filter high-quality code samples generated by ChatGPT and Copilot from GitHub.

ChatGPT/Copilot-Generated Code Collection: Yu et al. [51] 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. [51], 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/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 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 392 refined Python code samples, 339 Java samples, 376 JavaScript samples, 99 372 samples, and 353 C/C++ samples.

3.2 Benchmark Construction

Once we have collected code samples generated by ChatGPT and Copilot from GitHub, we move forward with constructing our benchmark.

Each coding task in RealisitcCodeBench comprises an input description for the target class (i.e., the class to be generated), a test suite for verifying the correctness of the generated code, and a canonical solution that acts as a reference implementation of the target class. Typically, LLMs generate code snippets based on input descrip- tions and the correctness is verified with the provided test suite. The generated code must conform to a consistent interface (e.g., the types of input parameters and return values) specified in the test suite for valid execution.

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-4's advanced capabilities in code comment generation [40] to produce concise summaries for each code sample. This allows us to clearly understand the core functionality of the code, making it easier to rewrite 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 [6, 12]. 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 [25], ensuring that the code's intent and task complexity remain largely unchanged. Additionally, we often modify the number and types of input and output parameters where possible.

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 (as shown in Figure 1). 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 modified SQL string along with an ordered list of values [§]. 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, with the named parameters flag being the given delimiter. Here, the input parameters increase to three, and the output changes to a dictionary format, including keys such as positional_sql, param_list, and execute_sql.

Reference Solution Generation: Next, we use ChatGPT-4 to generate Python, Java, JavaScript, TypeScript, and C/C++ solutions for each adapted programming problem, regardless of the original code's language, by providing the adapted function signatures as prompts. Although ChatGPT-4 is a highly capable tool, it can still produce incorrect code during the generation process. 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 utilize ChatGPT-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.

Expert Review: To ensure that our benchmark accurately reflects the types of code developers currently generate using LLMs, we engage 13 experienced engineers in the filtering process (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, Meituan), while the rest are from smaller IT companies. They have an average of 7.7 years of software development experience, ranging from 4 to 11 years, with a median of 6 years. Additionally, over the past one to two years, they have used either their company's internal LLM tools or external tools like ChatGPT and Copilot to assist in their daily coding tasks. We ask these engineers to evaluate whether the programming problems in our benchmark represent realistic development scenarios and whether developers would practically use LLMs to generate code to solve these problems. Only programming problems approved by a majority (at least 10 out of 13

[§]https://github.com/jerber/fastgql/blob/4c308e742685e0a1cf4dc6d05f29cfbaea2d039a/fastgql/query_builders/sql/query_builder.py#L464 engineers) are retained, ensuring that the benchmark reflects tasks developers are likely to use LLMs for in real-world development. Ultimately, 13 programming problems are excluded, leaving a total of 423 in our final benchmark.

3.3 Benchmark Characteristics

We provide more detailed statistics in Table 2. NCB comprises a total of 402 problems col- lected from online services, with 201 problems in Python and 201 in Java, spanning across 6 domains: Database, Artificial Intelligence, Data Science, Al- gorithm and Data Structure, Front-End, Software Engineering, and System Administration. This di- versity also leads to complex input data types in NCB, which are classified into 9 categories: num- ber (int/float/boolean), string, list (array), dict, ten- sor (matrix), data frame (table), plain text file, im- age, and special format file. The first four are the most common and simplest data types. Since a boolean can be represented by 1 and 0, we consider it as a type of number. Matrix and list are two simi- lar types of data, but they are categorized separately due to differences in their usage scenarios. Due to the current popularity of deep learning, tensor has become a very common data format. Therefore, we have designated a separate category for tensor and have included matrix within this category. The last three are all file types, differentiated by their processing methods. The content of a plain text file is text and can be directly read. Figures require processing of each pixel value. A special format file refers to files that require specific methods for processing, such as PDF and DOCX. Each problem within the dataset has been care- fully curated with a set of test cases to assess the correctness of solutions. On average, there are 9.3 test cases associated with each problem. These cases are strategically designed, with about 60% fo- cused on enhancing statement and branch coverage, and the remaining 40% dedicated to evaluating the robustness of solutions against corner values. The average word count for each problem in the NCB is 78.3.

Compared with Other Benchmark. Table 1 com- pares NCB to other benchmarks. It is noteworthy that our benchmark offers a substantial supplement to current benchmarks in terms of both problem and data types. Unlike HumanEval and MBPP, which consist of 96.9% and 89.5% algorithmic and basic programming problems respectively, our bench- mark features a more balanced distribution across each domain. In addition, NCB includes more data types. Fur- thermore, NCB focuses on assessing the model's ability to handle multiple file formats, a type of data that is both very commonly used in daily life and relatively challenging to process. We note that the problems involving files have fewer test cases, since GPT-4 still struggles to fully generate various types of file . This is also more challenging for human annotators to design compared to simpler data types. On the other hand, NCB is also limited by its size due to the high costs of problems collection and the construction of the evaluation framework. We are continuously working on expanding our benchmark.

4 EXPERIMENTAL SETUP

In this paper, 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 [11]. Table 3 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 twelve 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 or less, 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 [¶], we rely on the DeepSeek API interface (also accessed in September 2024), 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 selecting the token with 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, allowing us 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 [9], ensuring reliable and consistent evaluations of LLM performance across our benchmark.

Table 3. The overview of the twelve evaluated LLMs.

	Model Name	**Organization**	**Sizes**	**Release Time**	**Open-Source**
General	GPT-4	OpenAI	-	2023	
	GPT-3.5	OpenAI	-	2022	
	DeepSeek-V2.5	DeepSeek	236B	2024	✓
	Llama 3.1	Meta	8B	2024	✓
	Phi-3	Microsoft	7B	2024	✓
	Mistral	Mistral AI	7B	2024	✓
	ChatGLM	THUDM	6B	2024	✓
Coding	CodeGeex4	THUDM	9B	2023	✓
	DeepSeek-Coder	DeepSeek	6.7B	2024	✓
	StarCoder2	BigCode	7B	2024	✓
	CodeGen2.5	Salesforce	7B	2023	✓
	CodeLlama	Meta	7B	2023	✓

[¶]DeepSeek-V2.5 is an upgraded version that combines DeepSeek-V2-Chat and DeepSeek-Coder-V2-Instruct, integrating the general and coding abilities of both previous versions. However, the official website (https://huggingface.co/deepseek-ai/DeepSeek-V2.5) 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.

5 EXPERIMENTAL RESULTS

5.1 Overall Correctness

Based on the experimental results presented in Tables 4, 5, and 6, we can draw several key observations regarding the performance of the evaluated LLMs on our RealisticCodeBench benchmark.

GPT-4 consistently achieves the highest performance across all metrics (pass@1, pass@3, and pass@5), with an average pass@1 score of 67.39%, followed by DeepSeek-V2.5 and GPT-3.5, which achieves an average pass@1 of 66.08% and 59.61%, respectively. This trend remains consistent across different pass metrics, demonstrating the superior code generation capabilities of GPT-4, DeepSeek-V2.5, and GPT-3.5. 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. 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 14.56% in Python.

There are notable differences in pass@1 scores across 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 contribute to the elevated 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 increases only slightly, from 67.39% at pass@1 to 76.52% at pass@5. Similarly, GPT-3.5 shows limited improvement, rising from 59.61% at pass@1 to 68.20% at pass@5, while CodeGeeX4 progresses from 44.09% at pass@1 to 51.07% at pass@5. 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.

5.2 Performance With Different Benchmarks

We show the rank orders of all tested LLMs in Ta- ble 3 with regard to HumanEval and NCB, as well as the difference of rank orders. We also plot the corresponding performances on two benchmarks to scatter diagram in Figure 1. Based on the table and figure, we have some interesting findings. Performances of most LLMs on two benchmarks grow linearly proportional, and the differences of scores' rank order are around 0. It demonstrates that NCB can indeed reflect the coding abilities of LLMs as HumanEval does in most cases. However, we observe that some model series, notably the Phi, Deepseek-Chat, and WizardCoder, consistently exhibit a propensity to achieve supe- rior rankings on the HumanEval dataset as opposed to the NCB across various scales, as shown in the Table 3. Additional model families, including CodeGen and Llama-3-Instruct, similarly display the trend, though to a reduced degree. There might be a few potential hypotheses for the observation. First, as problems in NCB are more difficult and derived from natural user prompts, compared to those in HumanEval, LLMs with poorer generalization and instruction- following capabilities tend to perform worse. We find in preliminary experiments that problems in NCB cannot be properly

Table 4. The pass@1 of the 11 LLMs on our RealisticCodeBench benchmark.

	Model	**Python**	**Java**	**JavaScript**	**TypeScript**	**C++**	**Average**
General	GPT-4	83.46	60.64	69.62	60.21	62.46	67.27
	GPT-3.5	70.03	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	32.12	35.83

Table 5. The pass@3 of the 11 LLMs on our RealisticCodeBench benchmark.

	Model	**Python**	**Java**	**JavaScript**	**TypeScript**	**C++**	**Average**
General	GPT-4	85.80	65.31	78.90	69.08	66.31	76.99
	GPT-3.5	74.43	56.40	68.93	57.99	60.16	67.58
	DeepSeek-V2.5						
	Llama 3.1	54.06	28.32	47.17	37.37	25.32	40.28
	Phi-3	47.20	26.08	47.12	40.56	29.15	40.10
	Mistral	36.40	25.78	36.99	22.19	24.02	31.24
	ChatGLM	28.72	13.64	25.56	20.12	10.26	21.78
Coding	CodeGeex4	58.06	43.20	56.66	47.29	47.20	54.01
	DeepSeek-Coder	50.12	36.24	45.93	43.44	39.12	45.35
	StarCoder2	50.97	33.24	45.06	40.24	38.66	44.11
	CodeGen2.5	43.42	28.46	41.34	38.08	23.89	36.85
	CodeLlama						

solved by pre-trained base LLMs via mere in-context learning as HumanEval does, which indicates that to solve NCB problems requires stronger alignment and generalizability than HumanEval needs. Second, it is possible that training sets of some LLMs are over-specifiedly optimized for HumanEval-style problems. On one hand, pre- training data of certain LLMs may be contaminated. As GPT-4 (OpenAI et al., 2023) reported, 25% of HumanEval has been contaminated in their pre-training corpus. On the other hand, instruc- tion fine-tuning dataset may also be polluted. For example, Phi (Li et al., 2023b) reports a consid- erable amount of synthetic prompts resonating to some test samples in HumanEval. In (Yang et al., 2023b), the authors report leakage unidentifiable by n-gram overlap when using popular rephrasing techniques to create training sets. The performance discrepancy between HumanEval and NCB in our experiments is also an evidence of the potential contamination.

Table 6. The pass@5 of the 11 LLMs on our RealisticCodeBench benchmark.

	Model	Python	Java	JavaScript	TypeScript	C++	Average
General	GPT-4	86.62	68.20	84.53	73.25	70.02	76.52
	GPT-3.5	76.02	64.28	74.02	62.25	64.44	68.20
	DeepSeek-V2.5						
	Llama 3.1	55.50	23.29	48.32	38.04	27.43	38.51
	Phi-3	48.76	22.18	49.06	42.18	33.24	39.08
	Mistral	37.62	21.32	38.75	25.31	27.21	30.04
	ChatGLM	29.35	8.89	28.32	23.48	12.60	20.52
Coding	CodeGeex4	61.64	30.91	63.33	50.35	49.12	51.07
	DeepSeek-Coder	51.90	24.90	47.28	45.92	41.48	42.29
	StarCoder2	51.56	26.34	49.10	43.60	40.57	42.23
	CodeGen2.5	43.66	18.23	43.22	41.64	25.46	34.42
	CodeLlama						

6 DISCUSSION

6.1 Implications

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. Consequently, our benchmark is better suited for assessing LLM performance on practical coding tasks encountered by developers. Compared to other GitHub-derived benchmarks like CodeEval 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 are capable of addressing today.

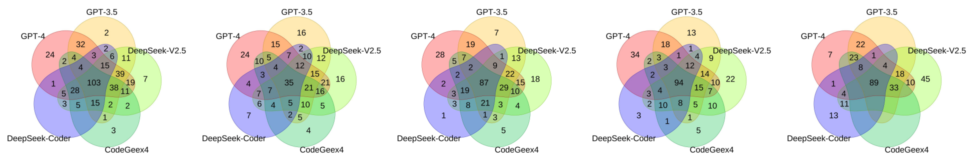

Fig. 1. python

Given data privacy concerns, as noted by Liang et al. [30], 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 gap between DeepSeek-V2.5 and GPT-4 in terms of average pass@1 is minimal (1.19%), but deploying DeepSeek-V2.5 requires 8 GPUs with 80GB of memory, which makes it

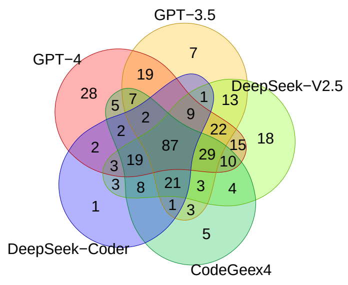

Fig. 2. javascript

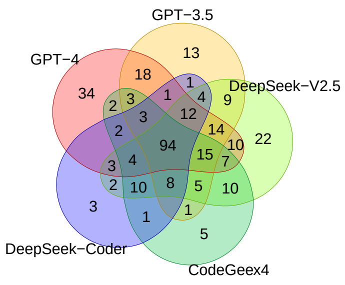

Fig. 3. typescript

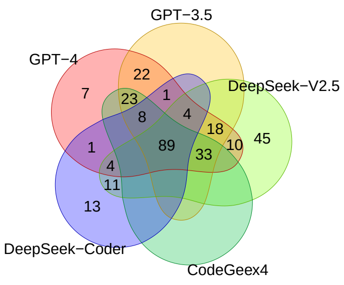

Fig. 4. c&cpp

a suitable option for well-funded enterprises prioritizing data privacy. CodeGeeX4-9B shows competitive performance relative to the proprietary model GPT-3.5 across five programming languages, achieving a pass@1 rate of 55.47% and a pass@5 rate of 61.6%. This narrows the accuracy gap with GPT-3.5 (pass@1 of 70.03%) to only 8.43% when multiple solutions are generated, demonstrating that developers can attain high-quality results from open-source models. Moreover, our experiments reveal that CodeGeeX4-9B runs on servers with dual NVIDIA GeForce RTX 3090 (24GB) GPUs, a setup costing approximately $3,000. This setup yields an average code generation time of around 8 seconds per programming question, making it an affordable solution for individual developers and small companies. 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, we recommend DeepSeek-V2.5 for well-funded companies focused on data privacy and CodeGeeX4-9B for privacy-conscious individual developers or small companies with limited budgets for everyday coding tasks.

6.2 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. [30] 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 solely on LLM-generated code from GitHub, reflecting open-source developers' use of LLMs. We do not have access to proprietary code generated by developers working in private companies, which limits the generalizability of our findings to broader industry applications. To mitigate this, we invite 13 developers from major tech companies (e.g., Microsoft, Huawei, ByteDance, Tencent, Alibaba, Bilibili, Meituan) and smaller IT companies to assess the relevance of our benchmark. These developers evaluate whether the programming problems in our benchmark represent realistic development scenarios and whether LLMs would be practically used to generate solutions for such problems. However, the limited number of programming problems (approximately 420 per language) may not fully capture the diversity of real-world coding tasks. As the use of LLMs in open-source development grows, we plan to expand our benchmark by incorporating more programming problems from GitHub and other repositories.

To mitigate potential risks of data leakage, we adapted the programming problems derived from GitHub code, altering the input/output parameters' types and quantities. We also calculate the Levenshtein distance between the original GitHub code and the LLM-generated code, finding significant differences. Manual inspection further confirms these differences, suggesting minimal risk of data leakage.

7 CONCLUSION

This paper presents a new, more realistic benchmark for evaluating LLMs in code generation tasks. By focusing on simpler, real-world tasks such as data processing, transformation, and frontend code generation, our benchmark better reflects how developers use LLMs for code generation in practice. Our approach, validated by experienced engineers and a large-scale developer survey, demonstrates that .

We recommend that small companies and individual developers concerned about code privacy consider using open-source models like CodeGeeX4 and YiCoder, which offer competitive performance without the risk of data leakage associated with cloud-based LLMs.

Our findings underscore the need for future research to develop LLMs capable of handling more complex tasks, but also to recognize the value of improving LLM performance on simpler, everyday tasks. This dual approach will allow developers to leverage LLMs more effectively in their daily workflows, gradually building trust and expanding the scope of tasks for which LLMs can be reliably used.

REFERENCES

[1] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. *arXiv preprint arXiv:1910.02216* (2019).

[2] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868* (2022).

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

[4] Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q Feldman, and Carolyn Jane Anderson. 2023. StudentEval: a benchmark of student-written prompts for large language models of code. *arXiv preprint arXiv:2306.04556* (2023).

[5] Jialun Cao, Zhiyong Chen, Jiarong Wu, Shing-Chi Cheung, and Chang Xu. 2024. 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*. 870–882.

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

[7] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227* (2022).

[8] Shubham Chandel, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. 2022. Training and evaluating a jupyter notebook data science assistant. *arXiv preprint arXiv:2201.12901* (2022).

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[10] Jianbo Dai, Jianqiao Lu, Yunlong Feng, Rongju Ruan, Ming Cheng, Haochen Tan, and Zhijiang Guo. 2024. MHPP: Exploring the Capabilities and Limitations of Language Models Beyond Basic Code Generation. *arXiv preprint arXiv:2405.11430* (2024).

[11] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[12] Aparna Elangovan, Jiayuan He, and Karin Verspoor. 2021. Memorization vs. generalization: Quantifying data leakage in NLP performance evaluation. *arXiv preprint arXiv:2102.01818* (2021).

[13] Jia Feng, Jiachen Liu, Cuiyun Gao, Chun Yong Chong, Chaozheng Wang, Shan Gao, and Xin Xia. 2024. Complex-CodeEval: A Benchmark for Evaluating Large Code Models on More Complex Code. *arXiv preprint arXiv:2409.10280* (2024).

[14] Lingyue Fu, Huacan Chai, Shuang Luo, Kounianhua Du, Weiming Zhang, Longteng Fan, Jiayi Lei, Renting Rui, Jianghao Lin, Yuchen Fang, et al. 2023. Codeapex: A bilingual programming evaluation benchmark for large language models. *arXiv preprint arXiv:2309.01940* (2023).

[15] Yanjun Fu, Ethan Baker, and Yizheng Chen. 2024. Constrained Decoding for Secure Code Generation. *arXiv preprint arXiv:2405.00218* (2024).

[16] Patrick Haller, Jonas Golde, and Alan Akbik. 2024. Pecc: Problem extraction and coding challenges. *arXiv preprint arXiv:2404.18766* (2024).

[17] Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. Aixbench: A code generation benchmark dataset. *arXiv preprint arXiv:2206.13179* (2022).

[18] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).

[19] Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010* (2023).

[20] Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin Clement, Nan Duan, and Jianfeng Gao. 2022. Execution-based evaluation for data science code generation models. *arXiv preprint arXiv:2211.09374* (2022).

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* (2018).

[22] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974* (2024).

[23] Mohammad Abdullah Matin Khan, M Saiful Bari, Do Long, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. 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)*. 6766–6805.

[24] Patrick Tser Jern Kon, Jiachen Liu, Yiming Qiu, Weijun Fan, Ting He, Lei Lin, Haoran Zhang, Owen M Park, George S Elengikal, Yuxin Kang, et al. [n. d.]. IaC-Eval: A code generation benchmark for Infrastructure-as-Code programs. ([n. d.]).

[25] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*. PMLR, 18319–18345.

[26] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories. *arXiv preprint arXiv:2404.00599* (2024).

[27] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. 2024. DevEval: A Manually-Annotated Code Generation Benchmark Aligned with Real-World Code Repositories. *arXiv preprint arXiv:2405.19856* (2024).

[28] Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852* (2023).

[29] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[30] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2024. 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*. 1–13.

[31] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[32] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–8.

[33] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).

[34] Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. 2024. On leakage of code generation evaluation datasets. *arXiv preprint arXiv:2407.07565* (2024).

[35] Daniel Nichols, Joshua H Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. 2024. Can large language models write parallel code?. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 281–294.

[36] Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. 2024. ScenEval: A Benchmark for Scenario-Based Evaluation of Code Generation. *arXiv preprint arXiv:2406.12635* (2024).

[37] Ruizhong Qiu, Weiliang Will Zeng, Hanghang Tong, James Ezick, and Christopher Lott. 2024. How Efficient is LLM-Generated Code? A Rigorous & High-Standard Benchmark. *arXiv preprint arXiv:2406.06647* (2024).

[38] Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. 2024. Can Language Models Solve Olympiad Programming? *arXiv preprint arXiv:2404.10952* (2024).

[39] Mohammed Latif Siddiq and Joanna CS Santos. 2022. 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*. 29–33.

[40] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, et al. 2023. Automatic code summarization via chatgpt: How far are we? *arXiv preprint arXiv:2305.12865* (2023).

[41] Xiangru Tang, Yuliang Liu, Zefan Cai, Yanjun Shao, Junjie Lu, Yichi Zhang, Zexuan Deng, Helan Hu, Kaikai An, Ruijun Huang, et al. 2023. ML-Bench: Evaluating Large Language Models and Agents for Machine Learning Tasks on Repository-Level Code. *arXiv e-prints* (2023), arXiv–2311.

[42] Xiangru Tang, Bill Qian, Rick Gao, Jiakang Chen, Xinyun Chen, and Mark B Gerstein. 2024. BioCoder: a benchmark for bioinformatics code generation with large language models. *Bioinformatics* 40, Supplement_1 (2024), i266–i276.

[43] Prashanth Vijayaraghavan, Luyao Shi, Stefano Ambrogio, Charles Mackin, Apoorva Nitsure, David Beymer, and Ehsan Degan. 2024. VHDL-Eval: A Framework for Evaluating Large Language Models in VHDL Code Generation. *arXiv preprint arXiv:2406.04379* (2024).

[44] Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F Xu, and Graham Neubig. 2022. Mconala: a benchmark for code generation from multiple natural languages. *arXiv preprint arXiv:2203.08388* (2022).

[45] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481* (2022).

[46] Claes Wohlin. 2014. 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*. 1–10.

[47] Tongtong Wu, Weigang Wu, Xingyu Wang, Kang Xu, Suyu Ma, Bo Jiang, Ping Yang, Zhenchang Xing, Yuan-Fang Li, and Gholamreza Haffari. 2024. VersiCode: Towards Version-controllable Code Generation. *arXiv preprint arXiv:2406.07411* (2024).

[48] Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. Top Leaderboard Ranking= Top Coding Proficiency, Always? EvoEval: Evolving Coding Benchmarks via LLM. *arXiv preprint arXiv:2403.19114* (2024).

[49] Yinghui Xia, Yuyan Chen, Tianyu Shi, Jun Wang, and Jinsong Yang. 2024. AICoderEval: Improving AI Domain Code Generation of Large Language Models. *arXiv preprint arXiv:2406.04712* (2024).

[50] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[51] Xiao Yu, Lei Liu, Xing Hu, Jacky Wai Keung, Jin Liu, and Xin Xia. 2024. Where Are Large Language Models for Code Generation on GitHub? *arXiv preprint arXiv:2406.19544* (2024).

[52] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. When language model meets private library. *arXiv preprint arXiv:2210.17236* (2022).

[53] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: continual pre-training on sketches for library-oriented code generation. *arXiv preprint arXiv:2206.06888*

(2022).

[54] Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Yuxiao Dong, and Jie Tang. 2024. NaturalCodeBench: Examining Coding Performance Mismatch on HumanEval and Natural User Queries. In *Findings of the Association for Computational Linguistics ACL 2024*. 7907–7928.

[55] Dewu Zheng, Yanlin Wang, Ensheng Shi, Ruikai Zhang, Yuchi Ma, Hongyu Zhang, and Zibin Zheng. 2024. Towards more realistic evaluation of LLM-based code generation: an experimental study and beyond. *arXiv preprint arXiv:2406.06918* (2024).

[56] Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2024. Beyond Correctness: Benchmarking Multi-dimensional Code Generation for Large Language Models. *arXiv preprint arXiv:2407.11470* (2024).

[57] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. 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*. 5673–5684.

[58] Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906* (2024).

[59] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877* (2024).