The Fortieth AAAI Conference on Artificial Intelligence (AAAI-26)

DSCodeBench: A Realistic Benchmark for Data Science Code Generation

Shuyin Ouyang[1*], Dong Huang[2], Jingwen Guo[1], Zeyu Sun[3], Qihao Zhu[4], Jie M. Zhang[1]

[1]King's College London
[2]Institute of Data Science, National University of Singapore
[3]Institute of Software Chinese Academy of Sciences
[4]Peking University
{shuyin.ouyang, jingwen.1.guo, jie.zhang}@kcl.ac.uk, dhuang@nus.edu.sg, {szy_, zhuqh}@pku.edu.cn

Abstract

We introduce DSCodeBench, a new benchmark designed to evaluate large language models (LLMs) on complicated and realistic data science code generation tasks. DSCodeBench consists of 1,000 carefully constructed problems sourced from realistic problems from GitHub across ten widely used Python data science libraries. DSCodeBench offers a more challenging and representative testbed, more complex code solutions, more comprehensive data science libraries, clearer and better structured problem descriptions, and stronger test suites. To construct the DSCodeBench, we develop a robust pipeline that combines task scope selection, code construction, test case generation, and problem description synthesis. The process is paired with rigorous manual editing to ensure alignment and enhance the reliability of the evaluation. Experimental result shows that DSCodeBench exhibits robust scaling behavior, where larger models systematically outperform smaller ones, validating its ability to distinguish model capabilities. The best LLM we test, GPT-4o, has a pass@1 of 0.392, indicating that LLMs still have a large room to improve for realistic data science code generation tasks. We believe DSCodeBench will serve as a rigorous and trustworthy foundation for advancing LLM-based data science programming.

Introduction

Recent advances in large language models (LLMs) have significantly accelerated research in automated code generation, particularly for data science tasks that involve complex workflows and domain-specific libraries (Bolyen et al. 2019; Hassani and Silva 2023). As a result, a growing number of benchmarks have been proposed to evaluate LLMs in this setting, including DS-1000 (Lai et al. 2023), DA-Code (Huang et al. 2024f), DataSciBench (Zhang et al. 2025), and others. Among them, DS-1000 has emerged as a standard for evaluating LLMs on data science code generation. However, it still falls short in capturing the full complexity and realism of real-world data science coding scenarios. We identify three key limitations of DS-1000 that motivate the need for a more robust and realistic benchmark: (1) Lack of realistic reference code. Most code problems are from Stack Overflow. Their solutions are usually easy, one-off code snippets, which cannot represent the complex scenarios encountered in real-world programming. For example, the average length of reference code in DS-1000 is only 3.6 lines, which limits its ability to reflect realistic programming tasks. (2) Insufficient testing. DS-1000 includes, on average, fewer than 2.1 tests for each coding problem (Lai et al. 2023), which leads to insufficient coverage of program behavior. As a result, many test cases are relatively simple and fail to fully explore the code's functionality or corner cases. (3) Poorly structured problem description. Each code problem description in DS-1000 includes a natural language description and code context, but these elements are often inconsistently formatted, which can limit clarity regarding the expected program behaviors. These limitations highlight the need for a more comprehensive and realistic benchmark for data science code generation.

To fill this gap, we develop a pipeline to systematically construct realistic data science code generation tasks from GitHub. Using this pipeline, we build DSCodeBench, a benchmark with 1,000 problems covering ten widely-used Python data science libraries, namely, NumPy, Pandas, SciPy, Scikit-learn, TensorFlow, PyTorch, Matplotlib, Seaborn, Keras, and LightGBM. DSCodeBench offers a more challenging and representative testbed, featuring longer solution code (averaging 22.5 vs. 3.6 lines in DS-1000) and richer, better structured problem descriptions (averaging 474 vs. 140 words in DS-1000). It also provides much stronger tests (averaging 200 vs. 2.1 tests in DS-1000) with an evaluation framework supporting customizable test case configurations.

We evaluate ten state-of-the-art LLMs on DSCodeBench, and the results highlight DSCodeBench's challenging nature and evaluation reliability. In particular, the best performing LLM, GPT-4o, has pass@1 score of 0.392. Open-source coding-specific models, including DeepSeek-Coder and Qwen2.5-Coder variants, have a pass@1 score of 0.222 and 0.229, respectively. We also observe a clear trend of scaling behavior, where larger models yield higher performance, a trend not always evident on DS-1000. Together, these findings demonstrate that DSCodeBench serves as a more challenging, rigorous, realistic, and trustworthy benchmark for evaluating and advancing LLM-based data science code generation.

To conclude, this paper makes the following contributions.

- We introduce DSCodeBench, a realistic benchmark designed to evaluate LLM performance on complicated data science code generation tasks.

[*]Corresponding author

- We develop a pipeline that constructs the benchmark from GitHub repositories, including task scope determination, code construction, test case generation, problem description synthesis, and manual editing.
- We perform a comprehensive empirical evaluation of 10 state-of-the-art LLMs on DSCodeBench. The benchmark, code, and experiment results are available at https://github.com/ShuyinOuyang/DSCodeBench.

DSCodeBench Construction

To ensure both the scale and quality of DSCodeBench, we adopt a structured pipeline consisting of task scope determination, ground truth code selection, test case generation, problem description generation, and manual editing. We first determine the scope to specify the categories of programming tasks. Next, we construct ground truth code from GitHub, which serves as the basis for subsequent steps. We then generate a test case script from the ground truth code, which is used to automatically produce test cases for evaluation. Problem descriptions are subsequently created to describe each task. Finally, systematic manual editing is performed to ensure consistency and correctness across all components. Figure 1 provides an illustrative overview of our benchmark construction process. The following sections introduce the details of each step in this process.

Code Task Scope Determination

The construction of the DSCodeBench begins with the careful choosing of the code task scope. Our goal is to ensure comprehensive coverage of widely used data science libraries, thereby capturing the practical challenges commonly encountered by developers. We extend the foundation laid by the existing DS-1000 benchmark, which covers seven libraries, by expanding the set to ten. In particular, we add Seaborn, Keras, and LightGBM to better capture the evolving landscape of data science. These libraries have seen significant adoption in recent years: Seaborn for its advanced statistical visualization capabilities, Keras as a widely used high-level interface for deep learning, and LightGBM for its scalable implementation of gradient boosting algorithms. Their inclusion addresses the increased demand for code generation benchmarks involving complex APIs and diverse usage patterns that pose additional challenges for LLMs. This broader coverage increases the diversity and practical relevance of DSCodeBench, establishing it as a more representative and robust benchmark for assessing code generation models in real-world data science workflows.

Ground Truth Code Construction

The first stage focuses on constructing high-quality ground truth code to create more realistic code generation tasks, consisting of four sub-steps: collecting seed code, sourcing code from GitHub, reconstructing missing context, and filtering the code candidates.

Collecting seed code. We begin by collecting seed code that serves as the query for retrieving realistic data science code from GitHub. The code is obtained from two sources: the reference code in the DS-1000 benchmark, and the answer code extracted from Stack Overflow coding questions. Inspired by DS-1000, we use library names such as "numpy" and "pandas" as keywords to search for highly voted Stack Overflow questions, collecting up to 500 top-ranked questions per library. For each selected question, we extract the highest-voted answer's code as seed code.

Sourcing code from GitHub. Each snippet is then split at the line level, and each line is used as a query to retrieve code from GitHub via the GitHub REST API. We filter these results to retain only Python files and apply deduplication to remove redundant candidates, resulting in a pool of 807,198 code candidates, each stored with its associated repository metadata for subsequent processing.

Context reconstruction. We perform context reconstruction for two main reasons. First, the retrieved code candidates are often snippets that lack the surrounding context necessary for standalone execution. Second, using raw GitHub code directly poses a risk of data leakage, as large language models may have been pre-trained on the same public code, potentially leading to unfair evaluation. To address both challenges, we design an automated context reconstruction pipeline that transforms code candidates into standardized, self-contained units. For each candidate snippet, we locate its original source file within the corresponding repository and parse it into an Abstract Syntax Tree (AST) for structural analysis. We then perform dependency resolution, identifying and extracting all required contextual elements, such as import statements, variable initializations, and auxiliary function definitions. In cases where the candidate represents a method encapsulated within a class, we refactor it into a top-level function, transforming its object-oriented dependencies into explicit parameters as needed. To ensure functional completeness, if any referenced variables or functions remain unresolved after context reconstruction, we promote them to explicit function parameters, enabling the candidate to be executed in isolation as a self-contained unit. In addition, we apply code reformatting to further standardize the reconstructed candidates. For instance, if the main code lacks a return statement, we append a return for the last variable (excluding those defined as function parameters). We systematically remove file-related operations to avoid external dependencies, which can mitigate the risk of data leakage.

Code filtering. We apply a two-stage filtering process, property and functional filtering, to ensure that only high-quality candidates are retained. At the property level, we implement: (1) compilation filtering to eliminate candidates that fail to execute at the file-level independently; (2) star filtering to retain only candidates from repositories with at least 10 GitHub stars, thereby promoting code quality; and (3) API call filtering to ensure that each candidate contains at least three API calls, ensuring adequate task complexity. At the functional level, we evaluate candidates based on their ability to successfully execute generated test cases. For each candidate, we prompt the LLM to generate 10 diverse input test cases based on the code. A candidate is accepted if it executes successfully on at least one test case. If all initial

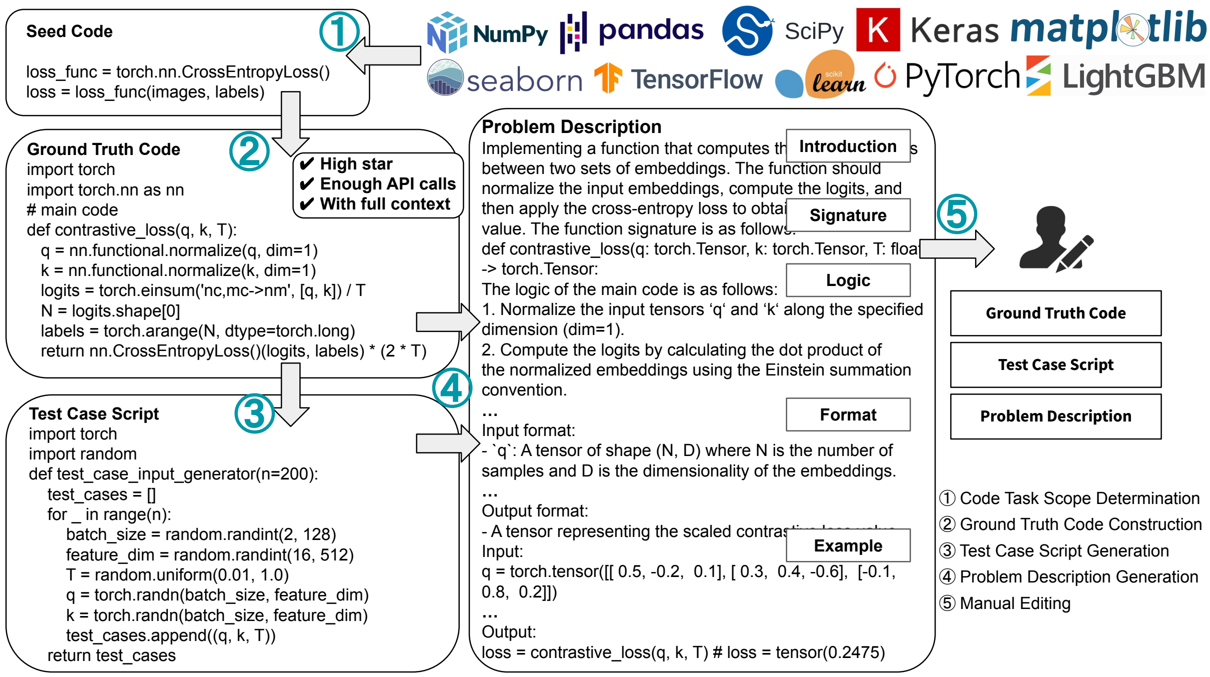

Figure 1: An example of DSCodeBench construction. The pipeline begins by determining the task scope, followed by collecting seed code and constructing ground truth code. This ground truth code is then used to generate a corresponding test case script that produces tailored input–output examples. Using both the ground truth and the generated test cases, the problem description is generated. Finally, all components are manually reviewed and aligned.

test cases fail, additional sets are iteratively generated until the candidate passes at least one or a predefined maximum of five attempts are reached. Candidates that fail to pass any test case within this limit are discarded. This process serves as a coarse-grained filter, designed to eliminate candidates with fundamental errors in the code logic. In the subsequent stage, we employ a more stringent procedure to further verify the quality of the ground truth code. Ultimately, this filtering pipeline yields 7,623 validated candidates, each paired with corresponding sample inputs and outputs, collectively constituting the ground truth code set for DSCodeBench.

Test Case Generation

Generating a large number of high-quality test cases for code problems is typically labor-intensive and costly. To address this challenge, we design an automatic approach that generates test-case input generation scripts rather than manually crafting individual test cases. We leverage LLMs to produce scripts that, when executed, are capable of generating diverse and valid test inputs for the corresponding ground truth code. However, not every generated script produces fully correct test cases. In particular, some generated test inputs may fail when passed to the ground truth code, raising runtime errors or exceptions. To mitigate this issue and ensure the reliability of our test case generation process, we incorporate a script repair mechanism based on self-repair (Xia and Zhang 2023).

Test Case Script Improvement. We employ a self-repair loop to iteratively enhance the quality of the generated test case scripts. Assuming the ground truth code obtained from the previous stage is correct, we enforce a strict criterion: all generated test case inputs must execute successfully without any errors, whether arising from the input data or the code logic. If a generated input triggers an error, we record both the input and the corresponding error message. This information is then used to guide the LLM in revising the test case script to eliminate the identified issues. The repair process is repeated for up to five iterations or until the script reliably generates valid inputs that pass all tests. This iterative refinement improves the robustness and reliability of the test case scripts, ensuring they satisfy the stringent standards of DSCodeBench. To further assess the effectiveness of the generated test cases, we perform a test coverage analysis by applying them to the ground truth code. Following this step, 2,407 code candidates are retained.

Problem Description Generation

Given the ground truth code, we leverage LLM to generate corresponding code problem descriptions, aiming to produce better-structured and well-formatted task descriptions. Each description is required to include at least five components: a general introduction, a function signature, logic, input and output formats, and illustrative input-output examples. However, we observe that LLMs often struggle to generate accurate and consistent input-output pairs. Although examples may appear plausible, they frequently fail to match the ground truth code's behavior, thereby compromising benchmark quality. To address this issue, we divide the description generation process into two separate stages.

First, we prompt the LLM to generate a problem description covering only the general introduction, the function signature, logic, and the input-output format, explicitly excluding examples. Second, we generate an input-output example independently by executing the test case generation scripts produced in the previous step. Finally, we concatenate the LLM-generated description and the independently generated example to form the final code problem description. This separation ensures that examples remain consistent with the underlying code logic. However, misalignments between the generated description and the ground truth code may still occur. To further ensure alignment quality, we perform manual editing in the next step.

Manual Editing

After completing the automated stages of construction, we obtain a preliminary pool of 1,000 code problem sets, each containing a ground truth code snippet, its corresponding problem description, and a test case generation script. To ensure the benchmark quality, we conduct a rigorous manual review and refinement process. Manual editing is collaboratively performed by four authors with expertise in programming and data science.

First, we double-check the test case generation scripts to ensure their robustness and variability. Specifically, we modify the random seeds used in the scripts and verify that they consistently generate valid and diverse test cases across different runs, reinforcing the reliability of the benchmark. Second, we validate the alignment between the ground truth code, the test case script, and the problem description. Specifically, we ensure that the ground-truth code aligns with the input-output examples in the problem description through execution-based validation. We also verify that the core logic outlined in the problem description accurately reflects the ground-truth implementation and provides sufficient information for a human to solve the data science task, using both human judgment and LLM-as-a-judge (Gu et al. 2024) for evaluation.

To mitigate the risk of data leakage, we do not directly reuse code extracted from GitHub repositories. Instead, we apply systematic perturbations to the extracted code to produce semantically similar solutions. These transformations include modifying function signatures (e.g., changing parameter names and return types), adding or removing lines of code while preserving the overall functionality, and restructuring control flows based on the code's context. While it is possible that similar code may have been seen during pretraining, our benchmark provides human-written problem descriptions that differ significantly from any original code comments or documentation, reducing the likelihood of overlap in both code and natural language. We conduct a similarity analysis between LLM-generated code and ground truth code. The text similarity (less than 0.4) and AST similarity (less than 0.5) confirm the effectiveness.

Furthermore, DSCodeBench is designed to support flexible evaluation settings. The benchmark provides users with the ability to customize the test cases by adjusting the random seed and specifying the number of test cases required, enabling more adaptable and robust evaluation protocols.

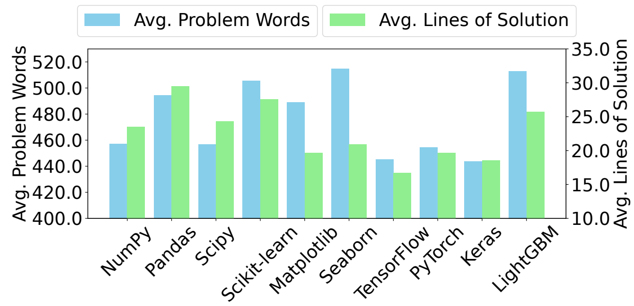

Figure 2: Distribution of tasks in DSCodeBench by library.

Benchmark Statistics

DSCodeBench focuses on ten data science libraries, including 131 problems for NumPy, 92 for Pandas, 112 for SciPy, 108 for Scikit-learn, 105 for Matplotlib, 83 for Seaborn, 110 for TensorFlow, 101 for PyTorch, 104 for Keras, and 54 for LightGBM. As shown in Figure 2, the average number of problem words for different libraries stays between 443.8 and 514.9, and the average line of solutions stays between 16.7 and 29.5, which demonstrate that tasks are not limited to short or simplified descriptions but instead reflect rich problem statements that mirror real-world project specifications.

We compared DSCodeBench with other code generation benchmarks. Table 1 offers a comprehensive comparison between DSCodeBench and several widely used benchmarks for code generation, including HumanEval (Chen et al. 2021), MBPP (Austin et al. 2021), APPS (Hendrycks et al. 2021), BigCodeBench (Zhuo et al. 2024), LiveCodeBench (Jain et al. 2024), DSP (Chandel et al. 2022), DA-Code (Huang et al. 2024f), DataSciBench (Zhang et al. 2025), and DS-1000 (Lai et al. 2023). When compared to general code generation benchmarks such as HumanEval, MBPP, APPS, BigCodeBench, and LiveCodeBench, DSCodeBench demonstrates significantly greater complexity and evaluation rigor, featuring more test cases per task and more intricate problem descriptions. HumanEval and MBPP, while foundational, are limited by their concise problem descriptions (averaging 23.0 and 15.7 words, respectively) and short canonical solutions (6.3 and 6.7 lines on average), reflecting small, well-scoped programming tasks. Even BigCodeBench and LiveCodeBench, which feature more challenging problems, offer a considerably lower evaluation depth with only 5.6 and 17.0 test cases per problem. In contrast, DSCodeBench is detailed in problem description, averaging 474.0 words, and requires more extensive solutions (22.5 lines on average), while each task is evaluated with 200 test cases by default.

Compared to existing data science code generation benchmarks, DSCodeBench further raises the standard across many dimensions. Although DSP and DS-1000 are early efforts to adapt benchmarks toward the data science domain, they still maintain short problem descriptions (71.9 and 140.0 words on average) and short solutions (4.5 and 3.6 lines on average), and often focus on isolated code snippets extracted from notebooks or Q&A forums. Moreover, evaluation in these datasets remains insufficient, with only 2.1 and 1.6 test

Benchmark	No. of Problems	No. of Libraries	Avg. Test Cases	Avg. Problem Words	Avg. Lines of Solution	Data Source
HumanEval	164	-	7.1	23.0	6.3	Hand-Written
MBPP	974	-	3.0	15.7	6.7	Hand-Written
APPS	10,000	-	13.2	293.2	18.0	Hand-Written
BigCodeBench	1,140	62	5.6	147.8	10.0	StackOverflow
LiveCodeBench	511	-	17.0	278.0	13.6	Online Coding Platforms
DSP	1119	8	2.1	71.9	4.5	Notebooks
DA-Code	500	-	-	40.2	85.0*	Annotation
DataSciBench	222	6	2.3	130.5	24.7	Multi-Sourced
DS-1000	1000	7	1.6	140.0	3.6	StackOverflow
DSCodeBench	1000	10	200.0 (default)	474.0	22.5	GitHub

Table 1: Comparison of DSCodeBench to other benchmarks. * refers to multiple solutions needed in one task. The top five are general code generation benchmarks, while the bottom four are data science–specific code generation benchmarks.

cases per problem on average. DA-Code and DataSciBench primarily focus on the diversity of data science tasks (e.g., data processing, visualization, and etc.), with tasks curated from real-world datasets. However, they provide insufficient test cases, particularly lacking comprehensive testing for intermediate generated code. In contrast, DSCodeBench is built from GitHub repositories and emphasizes robust evaluation, ensuring that each problem represents a substantial, end-to-end data science task with robust evaluation test suites rather than an isolated code snippet. It places particular focus on assessing whether the complex logic of the code is correctly implemented within the context of real-world coding tasks.

Benchmark State-of-the-art LLMs

Experiment Setup

Models. We evaluate both open-source and closed-source LLMs in code generation. For the closed-source model, we evaluate DSCodeBench with OpenAI's GPT models. Specifically, we experiment with GPT-3.5-turbo, GPT-4o-mini, and GPT-4o. For the open-source model, the LLMs we used include DeepSeek-Coder-Instruct (1.3B, 6.7B, and 33B), DeepSeek-Coder-V2-Lite-Instruct (15.7B), and Qwen2.5-Coder-Instruct (7B, 14B, and 32B).

Randomness control. LLMs are non-deterministic in nature (Ouyang et al. 2025a). We set the temperature of all models to 0.2, while keeping all other parameters at their default values. We also run all the models 3 times identically to mitigate the randomness that affects our experiment results. For each run, we collect the code snippets from the LLM's response, evaluate them, and record their test case passing status, whether they fully pass the test cases, partially pass the test cases, or fail all the test cases.

Compute resource. The experiments are conducted in an edge server with an Intel Xeon Platinum 8336C CPU with 128 cores, 8 * NVIDIA A100-SXM GPUs, and a total memory capacity of 2.0TiB.

Metrics

Pass@k We measure the performance with pass@k (Chen et al. 2021), where k code samples are generated per problem. A problem is considered solved if any sample passes the unit tests. The fraction of problems solved is reported. We pick k as 1 and 3 in our experiments. We run the experiments 3 times, and report the mean pass@1 and pass@3 in our results.

Test case passing status In this paper, we classify the passing status into three categories, namely correct, partially correct, and wrong. Correct means that the generated code can pass all the test cases. Partially correct means that the code can only pass part of all the test cases. Wrong means that the code fails to pass any test cases. We record the average numbers for each category and use them to reflect the code generation performance.

Experiment Results

Closed-source models As shown in Table 2, closed-source models such as GPT-3.5-turbo, GPT-4o-mini, and GPT-4o achieve the highest overall performance on DSCodeBench. Among them, GPT-4o achieves the highest pass@1 score of 0.392, the highest pass@3 score of 0.438, and the largest average number of correct outputs (391.7) across all evaluated models. Notably, the smaller GPT-4o-mini also surpasses GPT-3.5-turbo in all key metrics, indicating that architectural and training optimizations play a critical role beyond model scaling alone. Despite these advances, **the relatively low pass@1 scores of even the best models reflect the challenge posed by DSCodeBench in realistic and diverse data science code generation tasks**.

Open-source models In contrast, open-source code generation models, including DeepSeek and Qwen variants, lag significantly behind closed-source models on DSCodeBench. The strongest open-source model, DeepSeek-Coder-33B-Instruct, reaches pass@1 of only 0.222, which remains substantially below GPT-4o's 0.392. Similarly, the average numbers of fully correct and partially correct outputs are consistently lower, while the number of wrong outputs remains higher. However, the clear and consistent performance gains observed with increasing model sizes and improved versions indicate that **our chosen models on DSCodeBench follow the scaling law**.

Comparison with DS-1000 Compared to DS-1000, models consistently achieve lower pass@1 and pass@3 scores on DSCodeBench. For example, GPT-4o obtains pass@1 and pass@3 of 0.451 and 0.545 on DS-1000 but only 0.392 and 0.438 on DSCodeBench. This decline highlights that (1)

Benchmark	Model	Pass@1	Pass@3	Avg. Correct	Avg. Partially Correct	Avg. Wrong
DS-1000	GPT-3.5-turbo	0.374	0.442	373.7±6.2	0.0±0.0	626.3±6.2
	GPT-4o-mini	0.422	0.485	422.0±8.6	0.0±0.0	578.0±8.6
	GPT-4o	0.451	0.545	450.7±2.9	0.0±0.0	549.3±2.9
	DeepSeek-Coder-1.3B-Instruct	0.122	0.230	121.7±4.1	0.0±0.0	878.3±4.1
	DeepSeek-Coder-6.7B-Instruct	0.172	0.301	172.0±4.2	0.0±0.0	828.0±4.2
	DeepSeek-Coder-V2-Lite-Instruct(15.7B)	0.228	0.337	228.0±8.5	0.0±0.0	772.0±8.5
	DeepSeek-Coder-33B-Instruct	0.136	0.241	136.0±3.7	0.0±0.0	864.0±3.7
	Qwen2.5-Coder-7B-Instruct	0.005	0.012	4.7±2.5	0.0±0.0	995.3±2.5
	Qwen2.5-Coder-14B-Instruct	0.323	0.476	323.0±10.2	0.0±0.0	677.0±10.2
	Qwen2.5-Coder-32B-Instruct	0.419	0.508	419.0±3.7	0.0±0.0	581.0±3.7
DSCodeBench	GPT-3.5-turbo	0.307	0.346	307.3±5.6	82.3±3.4	610.3±6.3
	GPT-4o-mini	0.342	0.374	342.3±1.2	77.3±3.3	580.3±4.5
	GPT-4o	0.392	0.438	391.7±4.6	105.0±4.3	503.3±0.5
	DeepSeek-Coder-1.3B-Instruct	0.076	0.103	76.3±3.3	27.0±2.2	896.7±5.4
	DeepSeek-Coder-6.7B-Instruct	0.163	0.195	162.7±2.1	47.0±1.6	790.3±2.1
	DeepSeek-Coder-V2-Lite-Instruct(15.7B)	0.205	0.234	205.0±1.4	48.0±0.8	747.0±0.8
	DeepSeek-Coder-33B-Instruct	0.222	0.258	222.3±2.6	51.0±5.4	726.7±4.0
	Qwen2.5-Coder-7B-Instruct	0.116	0.164	116.3±7.1	28.7±0.5	855.0±7.5
	Qwen2.5-Coder-14B-Instruct	0.213	0.251	212.7±5.6	49.0±1.4	738.3±5.3
	Qwen2.5-Coder-32B-Instruct	0.229	0.260	228.7±2.6	45.7±4.2	725.7±2.5

Table 2: Experiment results on DSCodeBench and DS-1000.

DSCodeBench presents a more challenging task for LLMs. In addition, the variance of the evaluation metrics, such as the average number of correct outputs, is relatively smaller in DSCodeBench than in DS-1000. This lower variance reflects that (2) DSCodeBench provides a more consistent and reliable test suite, where evaluation outcomes are less affected by randomness and better reflect the capabilities of models. Furthermore, (3) the results on DSCodeBench exhibit clear adherence to the scaling law, with larger models systematically outperforming smaller ones. Within both DeepSeek and Qwen model families, increasing the size of parameters consistently leads to higher pass@1 and pass@3 scores and better correctness. For example, in the Qwen2.5-Coder series, the pass@1 steadily increases from 0.116 (7B) to 0.213 (14B) and 0.229 (32B), reflecting the critical role of scaling even within the same architectural family. In contrast, DS-1000 shows irregular scaling behavior. **The results show that DSCodeBench not only increases task difficulty but also offers a more rigorous, stable, and trustworthy benchmark to assess real-world data science code generation.**

Looking deeper into the benchmark, we observe that DS-1000 exhibits several issues that undermine its reliability as a benchmark, particularly in cases where it fails to reflect expected scaling behavior. First, the code context in DS-1000's problem descriptions makes it difficult for models to generate valid solutions that conform to its expected format, which is designed for code completion and insertion. As a result, models often produce either empty outputs or solutions that fail to meet basic structural requirements. Second, the reference solutions in DS-1000 are typically very short, and models tend to mimic this brevity, often resulting in repeated or incomplete outputs that encounter the response length constraint. Third, the evaluation is often based on a small set of test cases, which contributes to higher variance and unstable model performance. In contrast, DSCodeBench addresses these challenges in three aspects. First, its problem descriptions are designed to be aligned with ground truth code and standardized, reducing ambiguity and helping models generate structurally valid solutions. Second, the solutions are of realistic length, avoiding unintended biases toward brevity and minimizing the risk of incomplete or constrained outputs. Third, DSCodeBench employs a larger set of test cases, which leads to more reliable model performance. Together, these improvements make DSCodeBench a more robust and reliable benchmark than DS-1000 for evaluating LLMs on data science code generation tasks.

Library-level analysis Table 3 presents a detailed breakdown of the mean pass@1 scores on DSCodeBench across ten libraries. Closed-source models, particularly GPT-4o, dominate performance in nearly all libraries. GPT-4o consistently achieves the highest pass@1 scores (e.g., 0.405 for NumPy, 0.482 for SciPy, and 0.591 for PyTorch). In contrast, open-source models exhibit lower overall performance but still show reasonable scaling trends. Within both the DeepSeek and Qwen families, larger models outperform smaller ones across most libraries. For instance, DeepSeek-Coder-33B-Instruct shows clear improvements over DeepSeek-Coder-1.3B-Instruct, especially in libraries like scikit-learn (0.432) and PyTorch (0.479). Similarly, Qwen2.5-Coder-32B-Instruct shows notable gains compared to its smaller variants. Beyond cases where the generated logic fails to satisfy the task requirements, common errors include data structure mismatches, such as incompatible shapes or types during transformation steps, particularly in NumPy, TensorFlow, and PyTorch tasks. Visualization libraries also

Model	NumPy	Pandas	Scipy	Scikit-learn	Matplotlib	Seaborn	TensorFlow	PyTorch	Keras	LightGBM
GPT-3.5-turbo	0.226	0.239	0.315	0.330	0.311	0.149	0.400	0.498	0.324	0.216
GPT-4o-mini	0.280	0.348	0.432	0.435	0.238	0.124	0.373	0.488	0.356	0.290
GPT-4o	0.405	0.366	0.482	0.485	0.210	0.141	0.452	0.591	0.385	0.290
DeepSeek-Coder-1.3B-Instruct	0.097	0.036	0.000	0.176	0.022	0.060	0.115	0.142	0.042	0.049
DeepSeek-Coder-6.7B-Instruct	0.193	0.083	0.027	0.340	0.029	0.076	0.300	0.330	0.096	0.080
DeepSeek-Coder-V2-Lite-Instruct(15.7B)	0.193	0.149	0.027	0.401	0.013	0.129	0.333	0.465	0.179	0.099
DeepSeek-Coder-33B-Instruct	0.282	0.149	0.027	0.432	0.029	0.096	0.373	0.479	0.141	0.130
Qwen2.5-Coder-7B-Instruct	0.158	0.029	0.018	0.201	0.010	0.052	0.239	0.244	0.106	0.037
Qwen2.5-Coder-14B-Instruct	0.275	0.156	0.036	0.327	0.029	0.177	0.367	0.396	0.196	0.086
Qwen2.5-Coder-32B-Instruct	0.336	0.174	0.036	0.299	0.022	0.149	0.430	0.449	0.173	0.130

Table 3: Experiment result on DSCodeBench for each library (pass@1).

exhibit frequent issues, with models often mis-specifying figures or plotting arguments.

However, certain libraries, particularly Matplotlib and Seaborn, remain challenging across all models. The pass@1 scores for these libraries are consistently lower than for libraries like Scikit-learn, Keras, or PyTorch. This suggests that generating data visualization remains difficult for current LLMs, particularly under our evaluation criterion requiring a similarity score exceeding 50%. Even DeepSeek-Coder-33B-Instruct only achieves 0.029 and 0.096 pass@1 on Matplotlib and Seaborn tasks, respectively. The results reveal that while current models are increasingly capable of handling core data science libraries like NumPy and PyTorch, significant challenges remain for specialized domains such as visualization. **The detailed breakdown shows that DSCodeBench successfully covers a broad and challenging spectrum of real-world data science coding tasks.**

Related Work

LLM4Code

LLMs and their agent frameworks have advanced code intelligence demonstrating strong capabilities in code generation, completion, translation, repair, and summarization (Chen et al. 2021; Macedo et al. 2024; Yang et al. 2024; Ahmed et al. 2024; Huang et al. 2023b,a). Recent studies explore scaling laws and pretraining objectives for code-specific LLMs (Hui et al. 2024; Guo et al. 2024), semantic integration for improved understanding (Macedo et al. 2024), and retrieval-augmented or prompt-based methods to tackle complex coding tasks (Ouyang et al. 2025a; Tan et al. 2024; Tao et al. 2024; Huang et al. 2024c,a). Applications include test generation (Ryan et al. 2024; Huang et al. 2024e) and program repair (Chen et al. 2024a). Despite this progress, challenges remain in addressing hallucination, and the need for domain-specific tuning (Gu et al. 2025; Ouyang et al. 2025b), underscoring the importance of task-oriented benchmarks with robust evaluation.

Code Generation Benchmark

With growing model capabilities, benchmarks for code generation (Chen et al. 2021; Huang et al. 2024d,b; Qing et al. 2025; Hu et al. 2025; Zhuo et al. 2024; Jain et al. 2024; Huang et al. 2024f; Zhang et al. 2025; Chen, Pusarla, and Ray 2025; Chen et al. 2024b; Xiang et al. 2025) have evolved in difficulty and scope. HumanEval (Chen et al. 2021) set a standard for evaluating functional correctness from natural language. Subsequent datasets expanded this, including APPS (Hendrycks et al. 2021) for diverse coding tasks, MBPP (Austin et al. 2021) for beginner Python problems, and CodeContests (Li et al. 2022) for competitive programming. Domain-specific benchmarks emerged with DS-1000 (Lai et al. 2023), focusing on real-world data science tasks, and SWE-bench (Jimenez et al. 2023), emphasizing realistic software engineering workflows. However, data science code benchmarks remain limited: many rely on synthetic code, vague task descriptions, and insufficient test cases (e.g., fewer than 2.3 tests per task (Lai et al. 2023; Zhang et al. 2025)). To address this, we introduce DSCodeBench, a benchmark grounded in realistic data workflows, with well-formatted descriptions and comprehensive test suites, offering a more rigorous evaluation framework for LLM-based data science code generation.

Conclusion and Future Work

In this work, we present DSCodeBench, a benchmark dataset designed to evaluate the performance of LLMs on realistic data science code generation tasks. DSCodeBench consists of 1,000 problems constructed from real-world use cases across ten widely used Python data science libraries. In contrast to prior benchmarks that often focus on simplified or artificial tasks with limited evaluation rigor, DSCodeBench provides more challenging, library-specific problems alongside a robust evaluation framework with comprehensive test cases. By targeting library-aware code generation, DSCodeBench aims to bridge the gap between benchmark tasks and real-world programming scenarios.

Building on this foundation, several directions remain for future development. First, extending beyond Python to include languages like R would enable broader evaluation across diverse data science ecosystems. Second, incorporating more complex code structures, such as error handling, multi-function logic, and project-level tasks, would better reflect real-world workflows. Third, future versions could go beyond functional correctness by evaluating runtime performance, security, and adherence to coding best practices. Lastly, integrating human or LLM-based judges to assess readability and style could complement automated testing for a more holistic evaluation.

Acknowledgements

This work was supported by ITEA Genius and ITEA Green-Code projects (funded by InnovateUK), the UKRI Centre for Doctoral Training in Safe and Trusted Artificial Intelligence (EP/S023356/1), and the National Natural Science Foundation of China (62402482).

References

Ahmed, T.; Pai, K. S.; Devanbu, P.; and Barr, E. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th international conference on software engineering*, 1–13.

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

Bolyen, E.; Rideout, J. R.; Dillon, M. R.; Bokulich, N. A.; Abnet, C. C.; Al-Ghalith, G. A.; Alexander, H.; Alm, E. J.; Arumugam, M.; Asnicar, F.; et al. 2019. Reproducible, interactive, scalable and extensible microbiome data science using QIIME 2. *Nature biotechnology*, 37(8): 852–857.

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

Chen, J.; Hu, X.; Li, Z.; Gao, C.; Xia, X.; and Lo, D. 2024a. Code Search Is All You Need? Improving Code Suggestions with Code Search. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

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

Chen, S.; Feng, X.; Han, X.; Liu, C.; and Yang, W. 2024b. Ppm: Automated generation of diverse programming problems for benchmarking code generation models. *Proceedings of the ACM on Software Engineering*, 1(FSE): 1194–1215.

Chen, S.; Pusarla, P.; and Ray, B. 2025. DyCodeEval: Dynamic Benchmarking of Reasoning Capabilities in Code Large Language Models Under Data Contamination. In *Forty-second International Conference on Machine Learning*.

Gu, J.; Jiang, X.; Shi, Z.; Tan, H.; Zhai, X.; Xu, C.; Li, W.; Shen, Y.; Ma, S.; Liu, H.; et al. 2024. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594*.

Gu, X.; Chen, M.; Lin, Y.; Hu, Y.; Zhang, H.; Wan, C.; Wei, Z.; Xu, Y.; and Wang, J. 2025. On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and Methodology*, 34(3): 1–22.

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*.

Hassani, H.; and Silva, E. S. 2023. The role of ChatGPT in data science: how ai-assisted conversational interfaces are revolutionizing the field. *Big data and cognitive computing*, 7(2): 62.

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

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

Huang, D.; Bu, Q.; Qing, Y.; and Cui, H. 2023a. Codecot: Tackling code syntax errors in cot reasoning for code generation. *arXiv preprint arXiv:2308.08784*.

Huang, D.; Dai, J.; Weng, H.; Wu, P.; Qing, Y.; Cui, H.; Guo, Z.; and Zhang, J. 2024a. Effilearner: Enhancing efficiency of generated code via self-optimization. *Advances in Neural Information Processing Systems*, 37: 84482–84522.

Huang, D.; Qing, Y.; Shang, W.; Cui, H.; and Zhang, J. 2024b. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37: 11506–11544.

Huang, D.; Zeng, G.; Dai, J.; Luo, M.; Weng, H.; Qing, Y.; Cui, H.; Guo, Z.; and Zhang, J. M. 2024c. Effi-code: Unleashing code efficiency in language models. *arXiv preprint arXiv:2410.10209*.

Huang, D.; Zhang, J. M.; Bu, Q.; Xie, X.; Chen, J.; and Cui, H. 2024d. Bias testing and mitigation in llm-based code generation. *ACM Transactions on Software Engineering and Methodology*.

Huang, D.; Zhang, J. M.; Du, M.; Harman, M.; and Cui, H. 2024e. Rethinking the influence of source code on test case generation. *arXiv preprint arXiv:2409.09464*.

Huang, D.; Zhang, J. M.; Luck, M.; Bu, Q.; Qing, Y.; and Cui, H. 2023b. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Huang, Y.; Luo, J.; Yu, Y.; Zhang, Y.; Lei, F.; Wei, Y.; He, S.; Huang, L.; Liu, X.; Zhao, J.; et al. 2024f. Da-code: Agent data science code generation benchmark for large language models. *arXiv preprint arXiv:2410.07331*.

Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

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

Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; and Narasimhan, K. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Lai, Y.; Li, C.; Wang, Y.; Zhang, T.; Zhong, R.; Zettlemoyer, L.; Yih, W.-t.; Fried, D.; Wang, S.; and Yu, T. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, 18319–18345. PMLR.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Macedo, M.; Tian, Y.; Nie, P.; Cogo, F. R.; and Adams, B. 2024. InterTrans: Leveraging Transitive Intermediate Translations to Enhance LLM-based Code Translation. *arXiv preprint arXiv:2411.01063*.

Ouyang, S.; Zhang, J. M.; Harman, M.; and Wang, M. 2025a. An empirical study of the non-determinism of chatgpt in code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2): 1–28.

Ouyang, S.; Zhang, J. M.; Sun, Z.; and Penuela, A. M. 2025b. Knowledge-Enhanced Program Repair for Data Science Code. *arXiv preprint arXiv:2502.09771*.

Qing, Y.; Zhu, B.; Du, M.; Guo, Z.; Zhuo, T. Y.; Zhang, Q.; Zhang, J. M.; Cui, H.; Yiu, S.-M.; Huang, D.; Ng, S.-K.; and Tuan, L. A. 2025. EffiBench-X: A Multi-Language Benchmark for Measuring Efficiency of LLM-Generated Code. arXiv:2505.13004.

Ryan, G.; Jain, S.; Shang, M.; Wang, S.; Ma, X.; Ramanathan, M. K.; and Ray, B. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE): 951–971.

Tan, H.; Luo, Q.; Jiang, L.; Zhan, Z.; Li, J.; Zhang, H.; and Zhang, Y. 2024. Prompt-based code completion via multi-retrieval augmented generation. *ACM Transactions on Software Engineering and Methodology*.

Tao, W.; Zhou, Y.; Wang, Y.; Zhang, W.; Zhang, H.; and Cheng, Y. 2024. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems*, 37: 51963–51993.

Xia, C. S.; and Zhang, L. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385*.

Xiang, Y.; Yan, H.; Ouyang, S.; Gui, L.; and He, Y. 2025. Scireplicate-bench: Benchmarking llms in agent-driven algorithmic reproduction from research papers. *arXiv preprint arXiv:2504.00255*.

Yang, A. Z.; Kolak, S.; Hellendoorn, V. J.; Martins, R.; and Goues, C. L. 2024. Revisiting unnaturalness for automated program repair in the era of large language models. *arXiv preprint arXiv:2404.15236*.

Zhang, D.; Zhoubian, S.; Cai, M.; Li, F.; Yang, L.; Wang, W.; Dong, T.; Hu, Z.; Tang, J.; and Yue, Y. 2025. Datascibench: An llm agent benchmark for data science. *arXiv preprint arXiv:2502.13897*.

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