The Stylistic Blind Spot: Uncovering the Hidden Implicit Bias of Coding Style on LLM Code Evaluation

Zhiyuan Liu
Nanjing University
Nanjing, China
zhiyuanliu@smail.nju.edu.cn

Yingying Jiang
Nanjing University
Nanjing, China
yyjiang1@smail.nju.edu.cn

Huiyan Wang*
Nanjing University
Nanjing, China
why@nju.edu.cn

Abstract

Rapid iteration of large language models (LLMs) improved their performance on software engineering tasks. While prevailing benchmarks effectively measure LLMs' functional correctness, they often overlook non-functional aspects, such as coding styles, which are crucial in real-world development and can introduce evaluation bias. To investigate such implicit bias, we propose BenchPrism that automatically disperses diverse coding styles and generates style-fulfilled benchmark variants for LLM evaluation, and thus explore how the bias behaves and is influenced across different benchmarks, LLMs, and code tasks. Experiments confirm the existence of bias, which indeed leads to unstable performance deviations in LLM evaluation (from a 14.92% drop to a 7.51% increase), even distorting model rankings. We also raise valuable takeaways and hope to inspire future researchers to be mindful of the robustness of LLM evaluation when considering the possibility of such hidden bias.

CCS Concepts

• **Software and its engineering → Automatic programming**.

Keywords

Large Language Models, Code Tasks, Benchmarking, Coding Style

ACM Reference Format:
Zhiyuan Liu, Yingying Jiang, and Huiyan Wang. 2026. The Stylistic Blind Spot: Uncovering the Hidden Implicit Bias of Coding Style on LLM Code Evaluation. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3805590

1 Introduction

The integration of large language models (LLMs) into software engineering has become a rapidly evolving field. Recent surveys demonstrate that LLMs exhibit unique superiority in code-related tasks [9, 12, 36, 39], like code translation [33], program repair [28], and test generation [25], which span the entire software lifecycle [12] and suggest the emergence of a forward-looking ecosystem [34]. Generally, code-related tasks usually require a syntactic and semantic understanding of programming languages, which presents a natural challenge given the inherently probabilistic and non-deterministic nature of LLMs. This mismatch underscores the need for robust evaluations to accurately assess LLMs' capabilities.

To advance LLM capability evaluation on code-related tasks, various techniques and benchmarks have been proposed, from early n-gram matching (e.g., BLEU [22] and CodeBLEU [23]) to rigorous execution-based frameworks [14, 31], high-level benchmarks [8, 35, 37] and, more recently, benchmarks specific to assessing reasoning capabilities of LLMs [11, 18]. Despite the availability of diverse benchmarks and model leaderboards, the story is far from over. Intuitively, an ideal benchmark for LLM code evaluation should simulate real-world scenarios for practical performance. However, existing benchmarks usually focus on functionality-related properties, e.g., correctness [2, 5] and code complexity [13], while overlooking non-functional properties [3] like typical coding styles. This can struggle to simulate realistic programs, casting doubt on the credibility of the evaluation results.

Recent studies have noted non-negligible style inconsistency in LLM-generated code [27] and varying fragility of LLMs against stylometric adversarial attacks [7, 15]. Nevertheless, they focus on the abilities of language models themselves, failing to uncover how the evaluation process influences experimental outcomes. Furthermore, code snippets in existing benchmarks are often sourced from diverse origins and thus exhibit uncontrolled coding styles, highlighting a divergence from real-world coding practices and introducing potential distortion. *This oversight constitutes a potential research gap with implicit bias introduced, and may unevenly affect the robustness of using these benchmarks for LLM evaluation.*

To bridge this gap, we propose BenchPrism to investigate the impact and reveal the implicit bias introduced by automatically simulating diverse coding styles. Specifically, BenchPrism leverages 41 fine-grained coding styles from the literature and systematically transforms given benchmarks with diverse, controlled stylistic variations that satisfy style-combinatorial adequacy. We conducted experiments in Java on eight popular code-based tasks (e.g., code translation), using seven widely-used benchmarks (e.g., xCodeEval [14]) and six popular LLMs (e.g., GPT-5 mini [26]). The experimental results confirmed the existence of an uncontrollable bias introduced by coding styles, as evidenced by observed deviations in LLM performance. Our findings also demonstrate that coding style may even challenge the reliability of current leaderboards and further underscore the need for style-aware benchmarking.

2 BenchPrism

To reveal potential bias caused by coding styles in a benchmark, the main challenge is quantifying the degree of deviation relative to an ideal, unbiased style. However, since it is unlikely to determine a representative *default* coding style given the various real-world

*The corresponding author.

FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3805590

Category	Stylistic feature	Values	Source
Formatting	Indention Unit	`TAB`, `2_SPACES`, `4_SPACES`	[10, 21]
	Operator Spacing	`true`, `false`	[10, 21]
	Line Wrapping	`true`, `false`	[10, 21]
Structural	Modifier Order	`STANDARD`, `REVERSED`	[10]
	Declaration Merging	`true`, `false`	[10, 15, 16, 21]
	Assignment Statement	`REGULAR`, `COMPOUND`	[15]
Semantic	Case Format	`LOWER_CAMEL`, `UPPER_CAMEL`, `LOWER_UNDERSCORE`, …	[10, 15, 21]
	Conditional Return	`TERNARY`, `IF_ELSE_STMT`	[21]
	Loop Guard Clause	`GUARD`, `GUARD_ELSE`, `NO_GUARD`	[4]

Table 1: Selected Examples of Stylistic Features

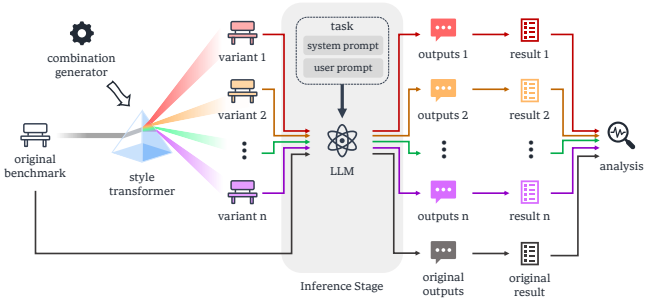

Figure 1: Overview of BenchPrism

programming conventions, our basic insight is to involve diverse coding styles as wide as possible instead of relying on a single one. We observe that most stylistic features indeed occur independently across different code structures because of their distinct target domains, e.g., variable naming versus code indentation. This enables us to treat stylistic features as *orthogonal* dimensions and incorporate diverse styles by combining feature dimensions.

To do so, we exhaustively collected 41 stylistic features from common coding conventions and prior literature [4, 10, 15, 16, 21], covering formatting, structural, and semantic, and diverse coding granularities (token-, statement-, and block-based levels), as listed in Table 1, with full lists on BenchPrism release [1]. Each stylistic feature corresponds to a discrete set of values to be assigned, which can be modeled as selection variables for later use.

The workflow of BenchPrism is shown in Figure 1. Ideally, to exhaustively cover coding styles, one usually needs to enumerate all possible variables and their combinations for each stylistic feature. Inspired by adequacy metrics in combinatorial testing [20], BenchPrism proposes a combination generator as shown in Figure 1, deploying a more practical adequacy criterion *style-combination coverage*. It takes the defined stylistic features and their discrete values as input and generates a compact set of value combinations achieving pairwise coverage, ensuring every possible value pair is covered. This is supported by Microsoft PICT [17] with seed 42 fixed and 26 combinations generated, and enables the output of transformation plans assigning specific values to each stylistic feature. After that, each transformation plan will be applied to the original benchmark to generate a benchmark variant that exhibits the targeted coding styles specified by the plan. To do so, BenchPrism designs a style transformer to parse and manipulate both token streams and ASTs of code snippets, and realizes automatic semantic-preserving transformations suiting each plan.

By doing so, BenchPrism indeed turns a benchmark into a series of variants by *dispersing* coding styles, ensuring pairwise coverage across stylistic combinations. By comparing performance between the original and transformed benchmark variants, we thus quantify LLMs' performance deviation and conduct bias analyses.

3 Bias Analyses

3.1 Setup

To explore the influence of BenchPrism for our bias analyses, we selected seven widely-used benchmarks[2] covering eight popular code-based tasks. In detail, we chose two versatile function-level benchmarks, i.e., xCodeEval [14] (463/508/1,000 code snippets for translation/repair/tag classification) and CodeScope [31] (30/60/100 for translation/repair/test generation), and two reasoning-focused function-level benchmarks, i.e., CodeMMLU [18] (655 for multiple-choice question (MCQ) answering) and CRUXEval-X [29] (698 separately for input and output reasoning). To cover diverse benchmarks with scales, we also selected one class-level benchmark, i.e., ClassEval-T [30] (94 class-level snippets for code translation), and two repo-level benchmarks, i.e., CoderUJB [35] (470/940 repositories for code repair/defect detection) and TestBench (108 for test generation). To limit the analysis scope, we consider only Java as the original language for BenchPrism and only C++ and Python as destination languages for translation tasks. To enable BenchPrism's analyses, we retain only parsable code snippets, filtering out 4.23% on average across benchmarks, which we consider acceptable. We adopted widely accepted evaluation metrics consistent with benchmark guidelines or prior literature to enable cross-benchmark comparisons. For tasks such as code translation/repair, and input/output reasoning, we use pass rates (method- and class-level pass rates for ClassEval-T, following its original work [30]). For test generation, we calculate line coverage, and for classification tasks, MCQ answering, and defect detection, we utilize F1 score and accuracy.

After that, we selected six state-of-the-art LLMs varying in parameter scale, access, and domain. Our proprietary, general-purpose selection includes GPT-5 mini [26] and Gemini 2.5 Flash [6] (hereafter Gemini 2.5). For open-access models, we selected two general-purpose 14B parameter models, Qwen2.5 14B Instruct [32] (hereafter Qwen2.5) and Phi-4 [1], alongside two code-specific models, the 9B parameter CodeGeeX4 [38] and CodeLlama 7B Instruct HF [24] (hereafter CodeLlama). To explore the implicit bias on LLM evaluation introduced by diverse code styles, we explicitly analyze how each LLM performs between the original benchmarks and their style-transformed variants generated by BenchPrism.

Experiments were conducted on a 12th Gen Intel(R) Core(TM) i7-12700H CPU @ 2.30GHz and 16GB RAM, with WSL Ubuntu 22.04 and MSYS2 UCRT 3.6.4, as required for Windows-exclusive libraries. Models were accessed via official APIs or widely used API platforms with default hyperparameters, while CodeLlama was run on an Nvidia A800 with greedy decoding for reproductivity.

[1] https://github.com/bakafradow/BenchPrism

[2] To control time and resource expenditure, for large benchmarks with over 1,000 snippets, we sampled 25% of code snippets (up to 1,000). We confirmed that the distribution of the sampled snippets is generally consistent with that of the entire.

Task	Metric	Model	Orig.	Var. (±SD)	Diff.	Trend
xCodeEval						
Code Translation (Java → C++)	Pass Rate	GPT-5 mini	94.55	95.14 (±0.68)	+0.59	
		Gemini 2.5	88.09	85.45 (±1.21)	-2.64	
		Phi-4	45.05	39.93 (±1.82)	-5.12	
		Qwen2.5	31.34	27.81 (±1.62)	-3.53	
		CodeGeeX4	37.75	34.98 (±1.85)	-2.77	
		CodeLlama	28.41	26.04 (±1.39)	-2.37	
Code Translation (Java → Python)	Pass Rate	GPT-5 mini	97.28	96.92 (±0.68)	-0.36	
		Gemini 2.5	73.27	74.73 (±2.59)	+1.47	
		Phi-4	44.24	35.13 (±3.36)	-9.11	
		Qwen2.5	27.48	23.81 (±1.93)	-3.67	
		CodeGeeX4	30.50	27.84 (±2.90)	-2.66	
		CodeLlama	14.57	6.37 (±2.90)	-8.20	
Code Repair	Pass Rate	GPT-5 mini	80.34	81.70 (±1.09)	+1.36	
		Gemini 2.5	74.21	72.29 (±1.47)	-1.92	
		Phi-4	35.74	33.50 (±1.47)	-2.24	
		Qwen2.5	15.64	15.09 (±1.23)	-0.55	
		CodeGeeX4	21.23	18.64 (±1.23)	-2.59	
		CodeLlama	10.78	11.06 (±0.99)	+0.28	
Code2Tag	Macro F1	GPT-5 mini	46.11	45.19 (±1.62)	-0.92	
		Gemini 2.5	38.44	35.38 (±1.69)	-3.06	
		Phi-4	4.91	5.91 (±0.38)	+1.00	
		Qwen2.5	14.19	15.79 (±1.12)	+1.60	
		CodeGeeX4	12.82	13.26 (±1.43)	+0.44	
		CodeLlama	6.84	6.60 (±0.56)	-0.25	
CodeScope						
Code Translation (Java → C++)	Pass Rate	GPT-5 mini	96.00	92.15 (±3.59)	-3.85	
		Gemini 2.5	80.00	72.92 (±7.55)	-7.08	
		Phi-4	56.00	46.00 (±7.30)	-10.00	
		Qwen2.5	36.00	30.77 (±6.15)	-5.23	
		CodeGeeX4	39.13	41.64 (±6.05)	+2.51	
		CodeLlama	24.00	19.08 (±5.90)	-4.92	
Code Translation (Java → Python)	Pass Rate	GPT-5 mini	100.00	96.00 (±2.72)	-4.00	
		Gemini 2.5	80.00	73.85 (±7.12)	-6.15	
		Phi-4	48.00	33.08 (±6.23)	-14.92	
		Qwen2.5	20.00	21.69 (±7.29)	+1.69	
		CodeGeeX4	43.48	35.62 (±7.03)	-7.86	
		CodeLlama	4.00	1.54 (±2.73)	-2.46	
Code Repair	Pass Rate	GPT-5 mini	69.64	73.08 (±2.42)	+3.43	
		Gemini 2.5	69.49	64.54 (±3.55)	-4.95	
		Phi-4	28.81	27.84 (±4.26)	-0.98	
		Qwen2.5	20.34	12.45 (±3.22)	-7.89	
		CodeGeeX4	18.00	16.08 (±3.01)	-1.92	
		CodeLlama	8.47	6.98 (±2.46)	-1.50	
Test Generation	Pass Rate	GPT-5 mini	90.60	88.91 (±2.88)	-1.69	
		Gemini 2.5	93.60	94.68 (±1.02)	+1.08	
		Phi-4	71.80	71.08 (±1.68)	-0.72	
		Qwen2.5	70.82	72.56 (±1.33)	+1.74	
		CodeGeeX4	83.58	83.46 (±2.06)	-0.12	
		CodeLlama	61.25	58.54 (±1.34)	-2.71	
	Line Coverage	GPT-5 mini	85.99	84.82 (±2.49)	-1.17	
		Gemini 2.5	92.72	93.49 (±0.59)	+0.77	
		Phi-4	88.55	89.21 (±1.76)	+0.66	
		Qwen2.5	91.19	90.20 (±1.33)	-0.99	
		CodeGeeX4	89.57	88.14 (±1.34)	-1.42	
		CodeLlama	89.88	88.84 (±0.95)	-1.03	
CodeMMLU						
MCQ Answering	Accuracy	GPT-5 mini	71.99	68.43 (±1.04)	-3.56	
		Gemini 2.5	58.79	59.45 (±1.39)	+0.65	
		Phi-4	47.39	48.33 (±1.17)	+0.93	
		Qwen2.5	48.12	45.48 (±1.21)	-2.64	
		CodeGeeX4	41.37	39.43 (±1.42)	-1.94	
		CodeLlama	29.56	30.23 (±0.63)	+0.68	

Task	Metric	Model	Orig.	Var. (±SD)	Diff.	Trend
CRUXEval-X						
Input Reasoning	Pass Rate	GPT-5 mini	99.85	99.75 (±0.19)	-0.10	
		Gemini 2.5	99.56	99.46 (±0.27)	-0.10	
		Phi-4	55.64	58.70 (±3.61)	+3.06	
		Qwen2.5	68.23	68.66 (±2.22)	+0.43	
		CodeGeeX4	44.93	44.72 (±1.80)	-0.21	
		CodeLlama	59.82	67.34 (±2.48)	+7.51	
Output Reasoning	Pass Rate	GPT-5 mini	99.85	99.98 (±0.05)	+0.13	
		Gemini 2.5	99.85	99.90 (±0.10)	+0.05	
		Phi-4	82.31	82.04 (±1.78)	-0.27	
		Qwen2.5	98.25	98.28 (±0.44)	+0.03	
		CodeGeeX4	94.25	93.02 (±2.28)	-1.23	
		CodeLlama	70.02	75.61 (±2.11)	+5.59	
ClassEval-T						
Code Translation (Java → C++)	Pass Rate (Method)	GPT-5 mini	9.52	10.47 (±1.36)	+0.95	
		Gemini 2.5	11.27	12.48 (±1.30)	+1.22	
		Phi-4	9.93	8.09 (±1.56)	-1.84	
		Qwen2.5	5.45	5.21 (±1.91)	-0.24	
		CodeGeeX4	8.18	9.66 (±1.02)	+1.48	
		CodeLlama	3.08	5.16 (±1.35)	+2.08	
	Pass Rate (Class)	GPT-5 mini	7.87	9.12 (±1.44)	+1.25	
		Gemini 2.5	6.74	10.07 (±1.26)	+3.33	
		Phi-4	8.99	6.66 (±1.71)	-2.33	
		Qwen2.5	1.12	4.15 (±1.61)	+3.03	
		CodeGeeX4	6.74	6.44 (±1.01)	-0.30	
		CodeLlama	3.57	4.67 (±1.04)	+1.10	
Code Translation (Java → Python)	Pass Rate (Method)	GPT-5 mini	78.83	84.23 (±1.98)	+5.40	
		Gemini 2.5	78.99	84.87 (±4.12)	+5.87	
		Phi-4	79.98	76.49 (±5.43)	-3.49	
		Qwen2.5	70.17	70.56 (±4.57)	+0.39	
		CodeGeeX4	74.94	72.91 (±4.26)	-2.03	
		CodeLlama	72.51	70.65 (±4.32)	-1.86	
	Pass Rate (Class)	GPT-5 mini	28.74	31.79 (±1.42)	+3.05	
		Gemini 2.5	26.44	26.97 (±2.59)	+0.53	
		Phi-4	31.03	27.94 (±2.13)	-3.09	
		Qwen2.5	26.44	25.86 (±2.47)	-0.57	
		CodeGeeX4	26.44	23.83 (±2.72)	-2.61	
		CodeLlama	19.54	14.41 (±2.69)	-5.13	
CoderUJB						
Code Repair	Pass Rate	GPT-5 mini	41.24	38.40 (±1.22)	-2.84	
		Gemini 2.5	35.12	33.82 (±1.60)	-1.29	
		Phi-4	13.68	11.28 (±1.10)	-2.40	
		Qwen2.5	13.49	10.19 (±0.89)	-3.30	
		CodeGeeX4	8.35	6.69 (±0.98)	-1.66	
		CodeLlama	5.35	3.94 (±0.56)	-1.42	
Defect Detection	Accuracy	GPT-5 mini	61.90	58.68 (±0.78)	-3.23	
		Gemini 2.5	56.00	54.36 (±1.08)	-1.64	
		Phi-4	50.10	49.84 (±0.36)	-0.27	
		Qwen2.5	50.62	49.73 (±0.30)	-0.89	
		CodeGeeX4	49.90	50.33 (±1.14)	+0.43	
		CodeLlama	50.21	50.81 (±0.73)	+0.60	
TestBench						
Test Generation	Pass Rate	GPT-5 mini	71.96	72.65 (±1.84)	+0.68	
		Gemini 2.5	29.91	31.16 (±3.22)	+1.26	
		Phi-4	11.21	13.55 (±2.16)	+2.34	
		Qwen2.5	14.95	12.19 (±2.33)	-2.77	
		CodeGeeX4	23.58	26.16 (±2.90)	+2.58	
		CodeLlama	19.42	16.65 (±2.15)	-2.76	
	Line Coverage	GPT-5 mini	51.09	51.89 (±2.01)	+0.80	
		Gemini 2.5	24.60	22.95 (±2.93)	-1.64	
		Phi-4	7.94	9.84 (±1.97)	+1.90	
		Qwen2.5	11.06	9.98 (±2.12)	-1.08	
		CodeGeeX4	16.21	17.13 (±2.18)	+0.92	
		CodeLlama	14.15	12.28 (±1.91)	-1.87	

Table 2: Overall Results (Values in %; Var. for Averages and SDs across Style Combinations, Visualized in Trend)

3.2 Overall Results

The overall experimental results are exhibited in Table 2. Compared with the evaluated metric results for each benchmark, the results for the transformed variants after applying BenchPrism's stylistic dispersions vary across benchmarks, with rather different mean and standard deviation values. Interestingly, these performance shifts vary in both direction and magnitude, as reflected in the clustered red and green blocks in column **Diff.**, with brightness proportional to the performance gaps, and in the diversely stretched curves of different heights and shapes in column **Trend**, which represent the contrasting fluctuation of values across style combinations held by the series of style-fulfilled benchmarks. Take CodeScope as an example. Results for translation and repair tasks are more likely to yield unstable pass rates, while results for the test generation task seem more robust. Moreover, *red zones* are mainly located in the results for xCodeEval and CodeScope, whereas *green zones* more frequently occur in CRUXEval-X and ClassEval-T.

Bias is also observed in model rankings, which are commonly-used when directly comparing the capabilities of multiple LLMs. Figure 2 illustrates results for ranking changes for selected benchmarks and tasks. Each plot links each model's original performance score (left) to its mean score after applying stylistic variations via BenchPrism (right). From the figure, we can observe quite a few cases that involve rank switching, such as rank change (e.g., Qwen2.5 in the first two Code-Choice reasoning tasks and Phi-4 in ClassEval-T), performance convergence (e.g., Qwen2.5 and CodeLlama in ClassEval-T except for the last plot) and divergence (e.g., Gemini 2.5, Qwen2.5, and CodeGeex4 in the last plot), etc.

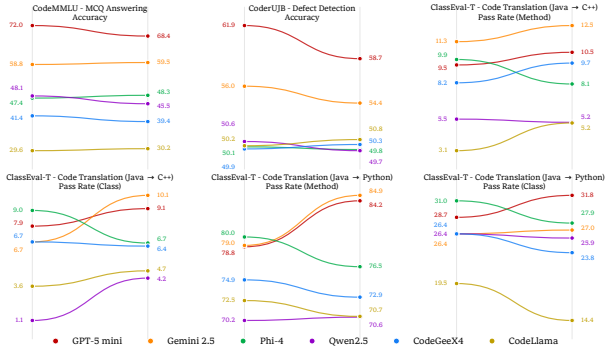

Figure 2: Diverse Fluctuations in Model Ranking

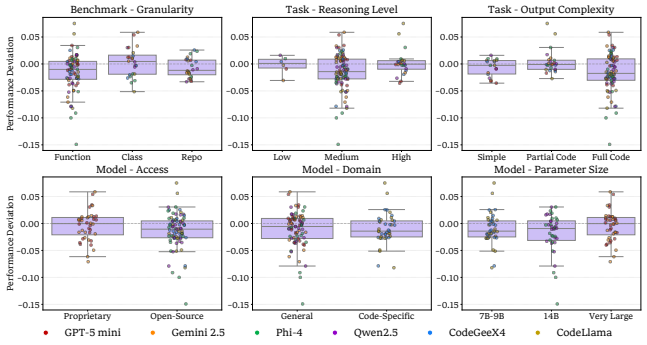

Figure 3: Several Factors Influenced by Stylistic Dispersions

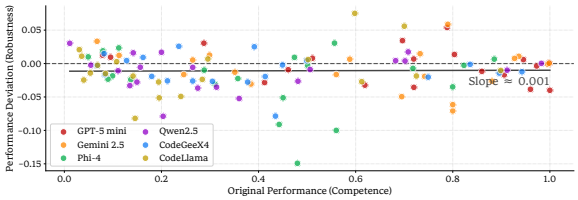

Figure 4: A Scatter Plot of Competence and Robustness

Results confirm that bias introduced by coding styles exists and demonstrate an uncontrolled influence on LLM evaluation across varying benchmarks and tasks. Moreover, such bias can even lead to biased LLM capability evaluation like distorting model rankings.

3.3 Factor Investigation

We also investigate the possible factors behind such LLM performance deviation, as in Figure 3. We observe that: (1) repo-level and class-level benchmarks are more stable than function-level ones, probably because of their more abundant context; (2) low-reasoning tasks (e.g., tag classification) and high-reasoning tasks (e.g., MCQ answering and defect detection) are more robust than other average tasks like translation, repair, and test generation; (3) tasks with complicated outputs like full code output typically suffer more; and (4) while code-specific models and larger-parameter models resist instability relatively well, which also aligns with widespread expectations, they are still subject to the performance bias.

When aggregating the performance deviations across all LLMs, Figure 4 shows that such bias does commonly exist and does not exhibit a clear correlation with the model's original capability. This further underscores a widespread need to examine possible bias control in the future, as it may impact all models.

3.4 Threat Analyses

BenchPrism's AST-based style transformations occasionally spawn malformed or functionality-changed code. To alleviate this threat, such snippets, detected by compilers or benchmark tests, fall back to their origins, ensuring semantic equivalence. The final fallback rate is low (3.24%), indicating minimal impact on results. Also, to address common issues in LLM-based evaluation, such as inconsistent output formats and occasionally repetitive explanations, that may obscure true capabilities, we employ designed prompts and post-processing scripts, ensuring each LLM's original performance is consistent with the results reported in existing literature.

Finally, considering data leakage, a long-standing problem in the LLM area [19], we investigated the knowledge cutoffs of the analyzed models and the open times of benchmarks officially released. By treating benchmarks after each LLM cutoff as clean and those before it as suspicious, we compared performance bias analyses across the two and conducted t-tests accordingly. Since the calculated p-value (0.78) is well above the acceptance threshold ($\gg$0.05), the difference is not statistically significant, suggesting that the bias persists regardless of whether data leakage occurs.

4 Discussion

Based on these analyses, we offer three valuable takeaways.

Takeaway 1: LLM evaluation should be done in a style-aware manner. Our analyses show that the unresolved instability in benchmarks interacts with multiple factors, leading to fluctuating performance and distorted model rankings. To prevent the disruption of such fragility, we advise researchers to enhance stylometric diversity during benchmarking and leverage style perturbation to quantify the stylistic robustness.

Takeaway 2: scaling is not a master key for robustness. The competence and robustness of models are counterintuitively orthogonal, as is shown by Figure 4. This reveals that current training paradigms overfit to canonical coding styles, e.g., acknowledged programming conventions. While current work on adversarial training of LLMs is still rare [34], we suggest future work to curate stylistically diverse training corpora, leverage models' promising reasoning capabilities to truly capture the essential semantics behind practical styles, and thus better suit in practice.

Takeaway 3: reexamining non-functional properties is also crucial for LLM. While stylometry is usually treated aesthetically, we observe that it can impact functional correctness, blurring the boundary between functional and non-functional properties. Nevertheless, compared with other non-functional properties that lack standard benchmarks [3], our work is merely a piece of the puzzle in this direction. We encourage the community to further explore non-functional properties and reexamine their relationships with LLM evaluation to mitigate potential biases.

5 Conclusion

Leveraging BenchPrism to disperse coding styles, we observe performance deviations from representative LLMs across typical benchmarks and code-based tasks, and confirm the presence of the uncontrollable bias. By revealing such implicit bias in LLM evaluation and further exploring related factors, we hope to raise concerns about robust benchmarking and evaluation in academia and industry. We also hope for deeper research in the future, as this work does not involve languages other than Java, investigation of concrete style combinations, and repeated experiments to eliminate randomness.

Acknowledgments

This work was supported by the Natural Science Foundation of China (No. 62302209), and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, et al. 2024. Phi-4 Technical Report. arXiv:2412.08905 [cs.CL]

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]

[3] Aymeric Blot and Justyna Petke. 2025. A Comprehensive Survey of Benchmarks for Improvement of Software's Non-Functional Properties. *ACM Comput. Surv.* 57, 7, Article 168 (Feb. 2025), 36 pages. doi:10.1145/3711119

[4] Binger Chen and Ziawasch Abedjan. 2023. DUETCS: Code Style Transfer through Generation and Retrieval. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2362–2373. doi:10.1109/ICSE48619.2023.00198

[5] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[6] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, et al. 2025. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. arXiv:2507.06261 [cs.CL]

[7] Atish Dipongkor. 2024. Can Large Language Models Comprehend Code Stylometry?. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 2429–2431. doi:10.1145/3691620.3695370

[8] 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 81, 13 pages. doi:10.1145/3597503.3639219

[9] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[10] Google. 2025. Google Java style guide. Retrieved 2025-09-28 from https://google.github.io/styleguide/javaguide.html

[11] Alex Gu, Baptiste Rozière, Hugh Leather, et al. 2024. CRUXEval: a benchmark for code reasoning, understanding and execution. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 659, 54 pages.

[12] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 220 (Dec. 2024), 79 pages. doi:10.1145/3695988

[13] Wenhao Hu, Jinhao Duan, Chunchen Wei, et al. 2025. DynaCode: A Dynamic Complexity-Aware Code Benchmark for Evaluating Large Language Models in Code Generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 21980–21997. doi:10.18653/v1/2025.findings-acl.1133

[14] Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, et al. 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)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 6766–6805. doi:10.18653/v1/2024.acl-long.367

[15] Zhen Li, Guenevere (Qian) Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. 2022. RoPGen: towards robust code authorship attribution via automatic coding style transformation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1906–1918. doi:10.1145/3510003.3510181

[16] Qianjun Liu, Shouling Ji, Changchang Liu, and Chunming Wu. 2021. A Practical Black-Box Attack on Source Code Authorship Identification Classifiers. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3620–3633. doi:10.1109/TIFS.2021.3080507

[17] Microsoft. 2025. PICT: Pairwise Independent Combinatorial Testing. Retrieved 2025-05-01 from https://github.com/microsoft/pict

[18] Dung Manh Nguyen, Thang Chau Phan, Nam Le Hai, Tien-Thong Doan, Nam V. Nguyen, Quang Pham, and Nghi D. Q. Bui. 2025. CodeMMLU: A Multi-Task Benchmark for Assessing Code Understanding & Reasoning Capabilities of CodeLLMs. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=CahIEKCu5Q

[19] Shiwen Ni, Xiangtao Kong, Chengming Li, Xiping Hu, Ruifeng Xu, Jia Zhu, and Min Yang. 2025. Training on the Benchmark Is Not All You Need. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 24948–24956. doi:10.1609/aaai.v39i23.34678

[20] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Comput. Surv.* 43, 2, Article 11 (Feb. 2011), 29 pages. doi:10.1145/1883612.1883618

[21] Oracle. 2025. Code Conventions for the Java Programming Language. Retrieved 2025-09-28 from https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

[22] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, USA, 311–318. doi:10.3115/1073083.1073135

[23] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE]

[24] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, et al. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[25] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[26] Aaditya Singh, Adam Fry, Adam Perelman, et al. 2025. OpenAI GPT-5 System Card. arXiv:2601.03267 [cs.CL]

[27] Yanlin Wang, Tianyue Jiang, Mingwei Liu, et al. 2025. Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE032 (June 2025), 23 pages. doi:10.1145/3715749

[28] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494. doi:10.1109/ICSE48619.2023.00129

[29] Ruiyang Xu, Jialun Cao, Yaojie Lu, et al. 2025. CRUXEVAL-X: A Benchmark for Multilingual Code Reasoning, Understanding and Execution. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 23762–23779. doi:10.18653/v1/2025.acl-long.1158

[30] Pengyu Xue, Linhao Wu, Zhen Yang, et al. 2025. ClassEval-T: Evaluating Large Language Models in Class-Level Code Translation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA063 (June 2025), 24 pages. doi:10.1145/3728940

[31] Weixiang Yan, Haitian Liu, Yunkun Wang, et al. 2024. CodeScope: An Execution-based Multilingual Multitask Multidimensional Benchmark for Evaluating LLMs on Code Understanding and Generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 5511–5558. doi:10.18653/v1/2024.acl-long.301

[32] An Yang, Baosong Yang, Beichen Zhang, et al. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL]

[33] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. *Proc. ACM Softw. Eng.* 1, FSE, Article 71 (July 2024), 24 pages. doi:10.1145/3660778

[34] Zhou Yang, Jieke Shi, Premkumar Devanbu, and David Lo. 2025. Ecosystem of Large Language Models for Code. *ACM Trans. Softw. Eng. Methodol.* 35, 1, Article 12 (Dec. 2025), 30 pages. doi:10.1145/3731753

[35] Zhengran Zeng, Yidong Wang, Rui Xie, Wei Ye, and Shikun Zhang. 2024. CoderUJB: An Executable and Unified Java Benchmark for Practical Programming Scenarios. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 124–136. doi:10.1145/3650212.3652115

[36] Quanjun Zhang, Chunrong Fang, Yang Xie, et al. 2024. A Survey on Large Language Models for Software Engineering. arXiv:2312.15223 [cs.SE]

[37] Quanjun Zhang, Ye Shang, Chunrong Fang, et al. 2024. TestBench: Evaluating Class-Level Test Case Generation Capability of Large Language Models. arXiv:2409.17561 [cs.SE]

[38] Qinkai Zheng, Xiao Xia, Xu Zou, 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* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 5673–5684. doi:10.1145/3580305.3599790

[39] Zibin Zheng, Kaiwen Ning, Yanlin Wang, et al. 2024. A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. arXiv:2311.10372 [cs.SE]