CodeJudgeBench: Benchmarking LLM-as-a-Judge for Coding Tasks

Hongchao Jiang[1], Yiming Chen[1*], Yushi Cao[1], Hung-yi Lee[2], Robby T. Tan[1,3]
[1]ASUS Intelligent Cloud Services (AICS), Singapore
[2]NTU Artificial Intelligence Center of Research Excellence (NTU AI-CoRE), Taiwan
[3]National University of Singapore, Singapore
{hongchao_jiang, mattym_chen, yushi_cao}@asus.com hungyilee@ntu.edu.tw robby.tan@nus.edu.sg

Abstract

Large Language Models (LLMs) are increasingly used not only to generate code, but also to judge it: comparing, ranking, or scoring competing solutions. However, their reliability in this evaluative role remains poorly understood. Inconsistent or flawed judgments can undermine benchmarks and distort training signals. This paper investigates the performance and robustness of LLMs when used as code judges. We introduce CodeJudgeBench, a benchmark explicitly designed to evaluate LLM-as-a-Judge models across three critical coding tasks: code generation, code repair, and unit test generation. We comprehensively benchmark the performance of 26 LLM-as-a-Judge models, encompassing general-purpose, code-tuned, and reasoning models. Our empirical findings reveal that relatively small reasoning models (e.g., Qwen3-8B) can outperform much larger non-reasoning models up to 70B. We further stress-test robustness by applying both general and code-specific perturbations. All models show significant instability and are sensitive to changes such as response ordering, variable naming, and misleading comments. These findings highlight serious concerns about the consistency and robustness of LLM-based judges for coding tasks.[1]

	Eval	Task	Robust	Source	# Samples
JudgeBench (Tan et al., 2025)	Pairwise	CodeGen	-	LLM	73
RM-Bench (Liu et al., 2025)	Pairwise	CodeGen	-	LLM	684
CJ-Eval (Zhao et al., 2025)	Pointwise	CodeGen	-	LLM	1860
CODE-DITING (Yang et al., 2025b)	Pointwise	CodeGen	-	LLM	2952
Scoring-Verifiers (Ficek et al., 2025)	Ranking	CodeGen	-	LLM	1680
JETTS (Zhou et al., 2025c)	BoN	CodeGen	-	LLM	1682
(Moon et al., 2026)	Pointwise	CodeGen	Hand-craft	Human	2000
CodeJudgeBench	Pairwise, BoN	CodeGen, CodeRepair, TestGen	Hand-craft, LLM	LLM	5352

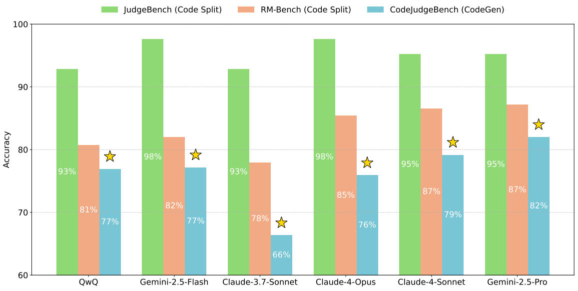


Figure 1: Overview of CodeJudgeBench (top) and comparison of LLM-as-a-Judge performance on CodeGen Task with previous benchmarks (bottom).

1 Introduction

Large Language Models (LLMs) (DeepMind, 2025; Anthropic, 2025b,a; Achiam et al., 2023; Grattafiori et al., 2024) have substantially advanced a wide range of software engineering tasks, including code generation (Li et al., 2022), code repair (Chen et al., 2024b), and unit test generation (Chen et al., 2023a; Mündler et al., 2024). Beyond generating code, LLMs are increasingly employed to *judge* code, performing tasks such as ranking, scoring, and comparing candidate solutions. This emerging *LLM-as-a-Judge* paradigm (Gu et al., 2024; He et al., 2025) offers a scalable alternative to human evaluation, and plays a central role in recent multi-agent workflows (Hong et al., 2024; Wu et al., 2024; Wang et al., 2025b), where LLMs not only generate code, but also act as automated judges; enabling robust, scalable, and fully automated software development pipelines.

Despite the growing adoption of the LLM-as-a-Judge paradigm, systematic studies on their effectiveness in coding tasks remain limited, which constrains progress in this area. Existing LLM-as-a-Judge benchmarks (Tan et al., 2025; Liu et al., 2025) are mostly designed for general domains and include only a small subset of relatively simple programming problems. Even benchmarks tailored to

---

[*] Corresponding author.

[1] `https://github.com/hongcha0/CodeJudgeBench`

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 19416–19448
July 2-7, 2026

coding scenarios (Zhao et al., 2025; Yang et al., 2025b; Ficek et al., 2025; Zhou et al., 2025c) primarily focus on code generation tasks, overlooking the potential for applying LLM-as-a-Judge to other types of coding tasks. Furthermore, the robustness and reliability of coding judges have received little attention.

To address this gap, we curate **CodeJudgeBench**, a benchmark designed to enable comprehensive analysis of LLM judge performance across diverse and challenging coding scenarios. CodeJudgeBench comprises three representative code judging tasks: code evaluation, code repair evaluation, and unit test evaluation. We primarily focus on assessing judge reliability when all candidate solutions originate from LLM rollouts within agentic or self-improving pipelines, rather than from human sources. The benchmark includes 5,352 solution pairs generated by various frontier LLMs, such as Gemini-2.5-Pro, on competitive programming problems, ensuring a high level of task difficulty and diversity.

As shown in Fig. 1, all evaluated LLM judges achieve substantially lower accuracy on CodeJudgeBench than on prior benchmarks. To probe robustness, we further stress-test judges using both general variations (e.g., response formatting and ordering) and code-specific perturbations (e.g., author attribution, misleading comments, version numbers, creation dates, variable renaming, statement reordering). We also test for model-dependent bias by comparing judge performance on code generated by different LLMs. Finally, we explore automated perturbation generation to enable scalable failure-case discovery and bias probing beyond hand-crafted perturbations.

Our empirical findings from this study reveal several important insights. First, smaller reasoning models such as Qwen3-8B (Yang et al., 2025a) outperform much larger non-reasoning models trained specifically for judge tasks, including those with up to 70B parameters (Wang et al., 2024; Pombal et al., 2025). Interestingly, recent efforts to fine-tune reasoning judges do not yield improvements over general-purpose reasoning models for coding tasks. Second, all models exhibit significant instability, with judgment accuracy sensitive to both general variations and coding-specific perturbations. Misleading comments, variable renaming, and statement reordering cause the largest degradations.

This paper makes the following contributions:

- **New Benchmark:** CodeJudgeBench is the first code benchmark explicitly designed to evaluate the performance and robustness of LLM-as-a-Judge models across a variety of coding tasks: code generation, code repair, and unit test generation.

- **Comprehensive Evaluation:** We evaluate 26 LLMs spanning general-purpose, code-tuned, and reasoning-enhanced architectures, revealing the capabilities of LLM-as-a-Judge on coding tasks more comprehensively.

- **Robustness Analysis:** Our study identifies key factors impacting evaluation reliability, including response order and formatting, generator bias, code comments, variable naming, and a range of code-specific perturbations. This provides the most extensive analysis to date of the fragile boundaries and biases inherent in LLM-as-a-Judge for coding tasks.

2 Related Work

Coding Judges While most coding judges rely on general-purpose LLMs, specialized models such as AceCodeRM (Zeng et al., 2025) and CriticGPT (McAleese et al., 2024) have been developed to enhance code evaluation accuracy. Existing works generally adopts two judging criteria: code functionality and code quality (He et al., 2025), with the latter emphasizing human-centric aspects like readability and style (Weyssow et al., 2024; Wang et al., 2025a). CodeJudgeBench advances execution-free assessment of code functionality, determining correctness of generated code or unit tests without execution and thereby overcoming scalability challenges associated with managing execution environments (Le et al., 2018; Zhou et al., 2024). General LLM-as-a-Judge benchmarks (Tan et al., 2025; Liu et al., 2025) include a coding split, but these are typically small in scale and focus primarily on code generation. In coding-specific evaluations, recent works (Yang et al., 2025b; Ficek et al., 2025; Zhou et al., 2025c) have mainly focused on judging basic code generation tasks sourced from MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021), which feature limited algorithmic complexity. To address these limitations, CodeJudgeBench advances the evaluation of LLM judges in coding by providing challenging data points across three critical tasks: code generation, code repair, and unit test generation.

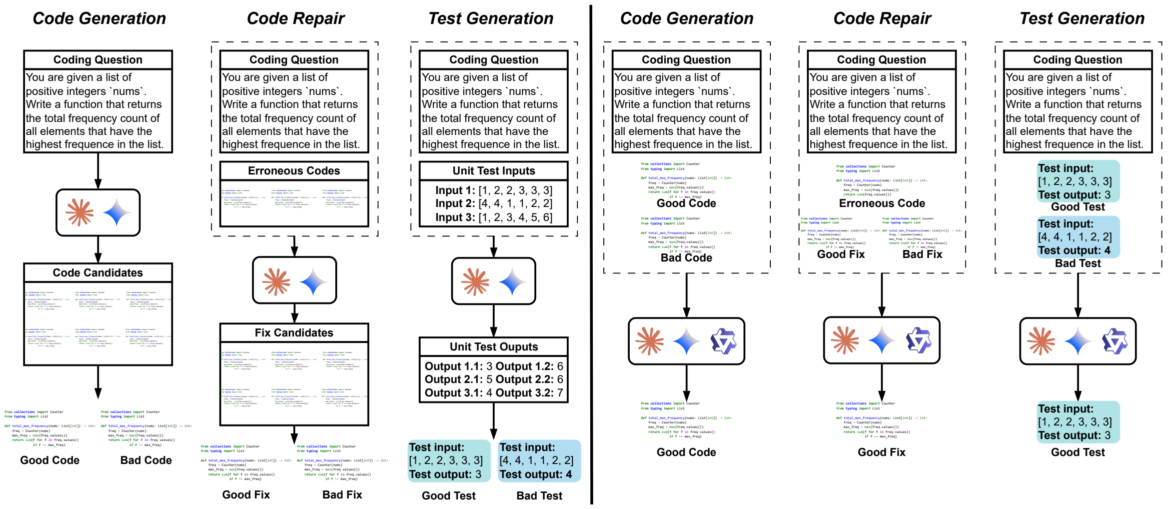


Figure 2: Overview of the proposed CodeJudgeBench. The left side illustrates the data curation process of CodeJudgeBench. The right side illustrates the evaluation process of CodeJudgeBench.

Judge Robustness LLMs such as GPT-4 demonstrate high correlation with human judgments when used as automated judges (Zheng et al., 2023). However, LLM-as-a-Judge systems exhibit biases such as position (Zheng et al., 2023), verbosity and formatting (Liu et al., 2025), where models tend to favor longer answers and those presented in markdown format. LLMs also display human-like biases (Ye et al., 2024), such as authority bias, where they prefer responses attributed to experts, and beauty bias, where they favor text with emoji over plain text. Notably, (Chen et al., 2024a) demonstrated that authority and beauty biases can successfully influence judgment of frontier models like GPT-4, achieving a 50% success rate. Recent work addresses these issues through supervised fine-tuning, chain-of-thought reasoning, and critique generation to enhance evaluation reliability (Zhu et al., 2025; Li et al., 2024; Ke et al., 2024). In parallel, a large body of research has examined the adversarial robustness of various neural models (Chen et al., 2023b; Gao et al., 2025), including adversarial attacks against general-domain judges (Chen et al., 2024c; Zhang et al., 2024). Despite these advances, the field still lacks systematic, large-scale benchmarks for assessing the robustness and reliability of coding judges. Importantly, bias specific to the coding domain remains underexplored. To address this gap, we introduce perturbation methods based on CodeJudgeBench, revealing key vulnerabilities of existing coding judges.

3 CodeJudgeBench

3.1 Overview

As shown in Fig. 2, CodeJudgeBench evaluates the capabilities of LLM-as-a-Judge across three crucial coding tasks: code generation (Jiang et al., 2024), code repair (Tang et al., 2024), and test generation (Alshahwan et al., 2024). Each data point in CodeJudgeBench consists of a triplet: Instruction, Good Response, Bad Response. The LLM-as-a-Judge is then tasked with evaluating both responses and choosing the one that better satisfies the instruction.

Formally, LLM-as-a-Judge is defined as:

$$J \leftarrow LLM(p \oplus r \oplus q),$$

where J is the final judgment or verdict produced by the LLM-as-a-Judge, p is the programming task, r is the response or set of responses to be evaluated, and q is the instruction prompting the LLM to act as a judge. The operator $\oplus$ specifies the method for concatenating or formatting p, r, and q into a single prompt for the LLM.

3.2 Task Definition

Code Generation (CodeGen): In this task, the LLM-as-a-Judge is presented with a coding problem statement and two candidate code snippets. The LLM-as-a-Judge has to accurately identify the correct code snippet.

Code Repair (CodeRepair): In the code repair scenario, the LLM-as-a-Judge is presented with the original coding problem statement, an erroneous code snippet along with its corresponding error message, and two candidate code repairs. The task of the LLM-as-a-Judge is to identify which candidate represents the correct fix.

Test Generation (TestGen): In the test generation setting (Chen et al., 2023a), the LLM-as-a-Judge is presented with a coding problem statement and two candidate unit test cases, each consisting of an input-output pair. The task of the LLM-as-a-Judge is to identify the correct unit test case for the given problem.

3.3 Data Curation

We source challenging coding questions from LiveCodeBench (Jain et al., 2025). Candidate responses are generated by a coding model and automatically labeled as *good* or *bad* based on objective correctness criteria, such as passing unit tests or matching ground-truth outputs. We exclude cases where only good or only bad responses are available, and randomly pair one good and one bad response to form each evaluation instance. Further task-specific details are provided in Appendix D.

While LiveCodeBench's unit tests have coverage and diversity limitations (Wang et al., 2025c; Zhou et al., 2025b), our pairwise framework is minimally affected, as it only requires the good response in each pair to pass more tests than the bad one. The judge model is thus evaluated on distinguishing between pairs, not on absolute correctness.

3.4 Data Statistics

To ensure the high quality and diversity of the generated responses, we utilize three state-of-the-art LLMs—Claude-3.7-Sonnet, Gemini-2.5-Flash, and Gemini-2.5-Pro, all of which demonstrate strong performance on coding benchmarks. For the CodeGen task, we further include models such as Qwen3-235B, Claude-4-Sonnet, Claude-4-Opus, Gemini-2.5-Flash-Lite.

Following (Zhang et al., 2025; Frick et al., 2025), we categorize the samples in each task into three difficulty levels: easy, medium, and hard based on the proportion of LLMs that correctly judge each sample. As pairwise judging is a binary task susceptible to random guessing, we only use top performing LLMs from both open-source and close-source for the assessment. A breakdown of the dataset can be found in Tab. 1.

Task	Easy	Medium	Hard	Overall
CodeGen	694	580	829	2103
CodeRepair	738	703	968	2409
TestGen	184	162	494	840

Table 1: Data statistics of CodeJudgeBench tasks.

4 Experiment

4.1 Experiment Setup

Evaluated models To evaluate the capabilities of LLM-as-a-Judge, we choose multiple representative LLMs, as shown in Tab. 2. We classify these models based on whether they are capable of thinking, open-source, and trained on domain-specific (i.e., code datasets) or task-specific data (i.e., LLM-as-a-Judge datasets). The details of these models can be found in Appendix C.

Implementation Details For the Judge-Tuned LLMs, we follow the prompts and sampling parameters provided in their official implementations. In the case of general LLMs, we use the pairwise prompt from (Tan et al., 2025), which instructs the LLM to first generate its own reference answer, which is then used to compare and evaluate candidate responses. The LLM is instructed to choose the better response without allowing for ties. To mitigate the risk of random guessing, each sample pair is evaluated twice. The good response is alternately placed as the first (i.e., position A) and second (i.e., position B) candidate, and the results are averaged.

Model	Thinking	Open-Source	Code-Tuned	Judge-Tuned
Claude-3.5				
Claude-3.7	✓			
Claude-4	✓			
Gemini-2.0				
Gemini-2.5	✓			
R1-Distill	✓	✓		
RM-R1	✓	✓		✓
QwQ	✓	✓		
Qwen3	✓	✓		
AceReason-Nemotron	✓	✓	✓	
DeepCoder	✓	✓	✓	
Qwen2.5-Coder		✓	✓	
Phi-4	✓	✓		
AceCodeRM		✓	✓	✓
Self-Taught		✓		✓
Skywork-Critic		✓		✓
Prometheus		✓		✓

Table 2: LLM-as-a-Judge models evaluated on CodeJudgeBench

LLM-as-a-Judge	Code Generation				Code Repair				Test Generation				Avg.
	Easy	Medium	Hard	Overall	Easy	Medium	Hard	Overall	Easy	Medium	Hard	Overall	
Claude-3.5-Sonnet-v2	72.62	61.81	43.91	58.32	81.50	71.19	50.15	65.90	66.85	60.49	45.65	53.15	59.12
Gemini-2.0-Flash	62.25	56.47	43.37	53.21	70.19	59.96	46.95	57.87	65.22	52.16	49.29	53.33	54.80
Gemini-2.0-Flash-Lite	64.63	54.14	43.00	53.21	65.11	56.33	46.38	55.02	61.14	54.32	46.86	51.43	53.22
Claude-3.7-Sonnet	89.91	71.21	43.18	66.33	94.31	77.95	48.04	70.94	94.02	77.47	51.21	65.65	67.64
Claude-4-Sonnet	**98.63**	88.53	56.27	79.15	99.12	**93.39**	64.00	83.33	97.01	**92.90**	64.88	77.32	79.93
Claude-4-Opus	97.12	84.91	51.81	75.89	**99.66**	91.68	60.02	81.40	97.01	84.88	58.10	71.79	76.36
Gemini-2.5-Flash	96.90	85.09	55.07	77.15	98.17	87.55	53.20	77.00	91.03	80.56	58.40	69.82	74.66
Gemini-2.5-Pro	98.85	**90.00**	**62.24**	**81.98**	99.32	**93.39**	**67.82**	**84.93**	**97.83**	89.81	**69.23**	**79.46**	**82.12**
AceCodeRM-7B	59.51	52.07	50.66	53.97	37.40	46.37	51.34	45.62	70.11	55.56	56.28	59.17	52.92
AceCodeRM-32B	70.32	55.17	49.22	57.82	50.41	54.48	50.21	51.52	69.57	54.94	52.23	56.55	55.30
Qwen2.5-Coder-32B	76.37	62.33	41.80	58.87	77.57	64.79	46.23	61.25	61.14	54.01	47.06	51.49	57.20
Skywork-Critic-70B	72.77	63.28	45.66	59.46	71.21	61.52	48.61	59.30	61.14	45.37	45.95	49.17	55.98
Prometheus-14B	75.65	61.64	43.43	59.08	78.79	66.43	47.88	62.76	59.51	50.62	45.04	49.29	57.04
Self-Taught-70B	72.12	60.60	42.88	57.42	76.36	62.38	45.61	59.92	65.76	53.70	45.85	51.73	56.36
R1-0528-Distill-Qwen3-8B	94.52	77.84	48.07	71.61	94.44	77.10	50.62	71.77	86.68	75.31	50.61	63.27	68.88
R1-Distill-Qwen-14B	86.96	71.03	40.23	64.15	92.68	78.17	44.37	69.03	89.40	70.06	47.37	60.95	64.71
R1-Distill-Qwen-32B	95.24	76.90	36.55	67.05	97.29	80.94	40.34	69.63	95.11	80.86	51.01	66.43	67.70
DeepCoder-14B	92.44	72.24	39.32	65.93	95.80	76.96	41.48	68.47	93.75	76.85	47.87	63.51	65.97
Qwen3-8B	93.52	78.10	47.77	71.23	94.44	81.37	46.90	71.52	79.62	63.27	41.50	54.05	65.60
Qwen3-14B	98.05	83.02	43.49	72.40	98.78	88.55	43.65	73.64	95.92	79.94	52.83	67.50	71.18
Qwen3-32B	97.69	86.98	47.35	74.89	98.92	88.90	46.80	75.05	95.92	78.40	53.04	67.32	72.42
QwQ-32B	99.06	89.74	49.34	76.89	99.53	90.68	52.89	78.21	97.83	92.28	59.31	74.11	76.40
Phi4-Reasoning-Plus-14B	92.44	74.57	46.56	69.42	82.18	66.86	44.68	62.64	89.40	79.01	54.15	66.67	66.24
AceReason-Nemotron-14B	97.12	82.50	45.90	72.90	98.10	86.06	47.26	74.16	98.91	89.20	61.03	74.76	73.94
RM-R1-14B	90.49	73.36	39.02	65.48	91.80	74.04	44.94	67.79	88.59	68.83	39.57	55.95	63.07
RM-R1-32B	92.07	74.05	37.94	65.76	89.84	71.19	38.79	63.89	84.24	73.77	43.32	58.15	62.60

Table 3: Performance of LLM-as-a-Judge models on CodeJudgeBench. Accuracy is reported across Easy, Medium, and Hard difficulty splits, as well as the overall average across all tasks. Models are grouped into four categories: (1) proprietary non-thinking, (2) proprietary thinking, (3) open-source non-thinking, and (4) open-source thinking judges. We highlight the best performance with **bold**, and the best open-source performance with underline.

4.2 Main Results

Tab. 3 presents the performance of various LLM-as-a-Judge models on CodeJudgeBench tasks. In general, reasoning models such as DeepCoder-14B, AceReason-14B, Qwen3, QwQ, RM-R1, Claude 3.7/4, and Gemini-2.5—Pro/Flash consistently outperform others. These models allocate more tokens for code analysis, which enhances their ability to understand and accurately judge code responses. Notably, smaller thinking models, such as Qwen3-8B, surpass CoT models like Prometheus-14B and Self-Taught 70B in overall accuracy. In contrast, non-thinking models—including proprietary models like Claude-3.5 and models specifically fine-tuned for LLM-as-a-Judge tasks such as Prometheus-14B—achieve accuracies below 60%, approaching the random guess baseline of 50%.

Interestingly, fine-tuning thinking models specifically for LLM-as-a-Judge tasks does not always yield improved performance. For example, RM-R1 underperforms relative to similarly sized models such as Qwen3-32B and QwQ, likely due to insufficient code-related training data in LLM-as-a-Judge datasets, which often focus on modeling general human preferences. Among all evaluated models, closed-source models such as Gemini-2.5-Pro and Claude-4-Sonnet achieve the highest scores on CodeJudgeBench.

In terms of task difficulty, judging the correctness of unit test generation is the most challenging for LLM-as-a-Judge models, followed by code generation, with code repair being the easiest. The greater difficulty of test generation likely arises because it provides only the problem statement, while code generation and repair have richer context, such as code snippets and error messages, which support more accurate judgment. While larger model sizes often correlate with improved performance, this trend is not as apparent in CodeJudgeBench. Several 14B models perform comparably to their larger counterparts; for example, RM-R1-14B achieves results similar to RM-R1-32B, and Qwen3-14B is on par with Qwen3-32B.

Pointwise vs. Pairwise: In addition to pairwise evaluation, pointwise evaluation is another commonly used schema. In the pointwise approach, the LLM-as-a-Judge evaluates each candidate response independently, assigning a score on a Likert scale (e.g., 1 to 5). The response with the highest score among the candidate pair is then selected

LLM-as-a-Judge	Ice-Score		CodeJudge		Pairwise
	Accuracy	Tie	Accuracy	Tie	
DeepCoder-14B	58.01	64.10	57.25	58.06	**65.93**
Phi4-Reasoning-Plus-14B	45.32	24.49	54.54	47.46	**69.42**
Qwen3-8B	57.68	63.43	62.05	63.72	**71.23**
Qwen3-14B	62.67	60.63	63.24	59.72	**72.40**
Qwen3-32B	63.72	61.34	65.38	60.29	**74.89**
QwQ-32B	65.00	62.67	64.05	62.20	**76.89**
R1-Distill-Qwen-14B	54.92	63.34	56.87	57.96	**64.15**
R1-Distill-Qwen-32B	59.11	61.10	59.63	60.82	**67.05**
R1-0528-Distill-Qwen3-8B	62.53	48.45	61.53	56.82	**71.61**
AceReason-Nemotron-14B	63.29	57.58	63.81	54.21	**72.90**

Table 4: Comparison of pointwise (Ice-Score, CodeJudge) and pairwise prompting on the CodeGen task. For each model, the best performance is in **bold** and the second-best is underlined. The Tie column reports the percentage of instances where both candidates receive the same score.

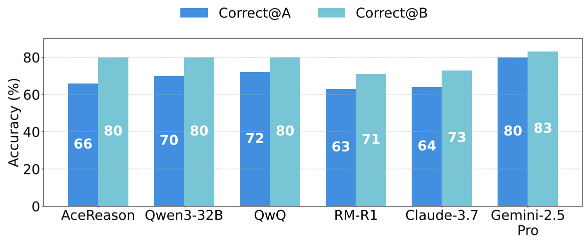
as the preferred answer. Alternatively, the LLM-as-a-Judge can provide a binary judgment indicating whether the code is good or bad. We use the Ice-Score (Zhuo, 2024) prompt for the scoring approach and the CodeJudge (Tong and Zhang, 2024) prompt for the binary approach.

Our experiments on the CodeGen task show that the pointwise approach significantly underperforms compared to the pairwise approach. Further analysis reveals that this discrepancy is primarily due to the frequent occurrence of tied scores or identical binary judgment between candidates. In such cases, we randomly select a candidate when calculating the accuracy. As shown in Tab. 4, approximately 50% of pointwise judgments result in ties for various models. We attribute this to the absence of direct comparison in the pointwise setting, making it difficult for the model to distinguish between highly similar candidates and resulting in arbitrary or indistinguishable scoring.

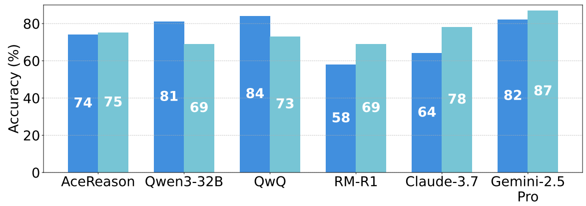
4.3 Robustness of Coding Judges

Ideally, LLM-as-a-Judge models should be capable of evaluating a wide range of outputs, with their assessments remaining unaffected by superficial factors. Motivated by this, we study the robustness and generalization abilities of LLM judges by assessing their sensitivity to response order, outputs from different coding models, and a range of functional and non-functional perturbations specific to coding.

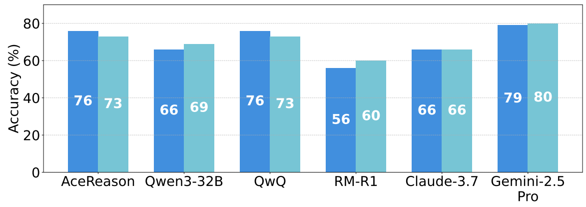
Response Ordering: We first investigate whether LLM-as-a-Judge models produce consistent evaluations under trivial changes in response order, specifically by swapping the position of the correct response within the pair. Surprisingly, as shown in Fig. 3, model performance varies substantially depending on the order, with discrepancies reaching up to 14%. For certain models, this positional bias persists across all tasks: for example, RM-R1-32B and Claude-3.7-Sonnet consistently exhibit recency bias, tending to prefer the response presented in the second position across CodeGen, CodeRepair, and TestGen tasks. In contrast, Qwen3-32B displays a task-dependent position bias, performing better when the correct response is first for CodeGen, but preferring the second position for CodeRepair. Gemini-2.5-Pro demonstrates the least position bias among the evaluated models.



(a) Comparison of CodeGen Response Ordering



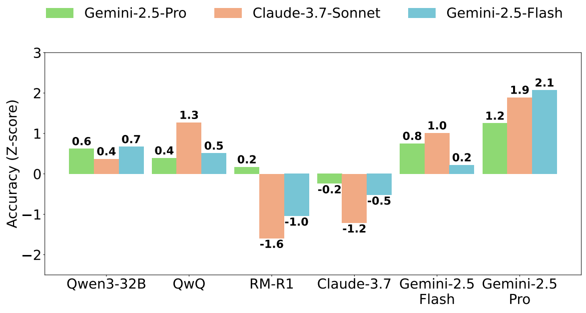
(b) Comparison of CodeRepair Response Ordering



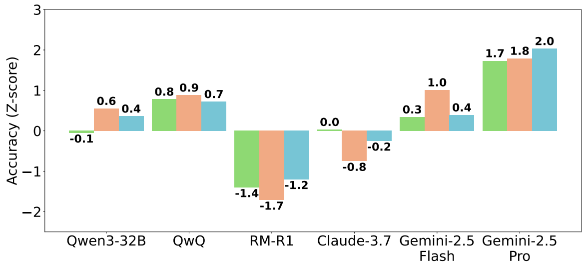
(c) Comparison of TestGen Response Ordering

Figure 3: Performance under different response orderings. Correct@A and Correct@B indicate that the correct answer appears in the first or second position, respectively

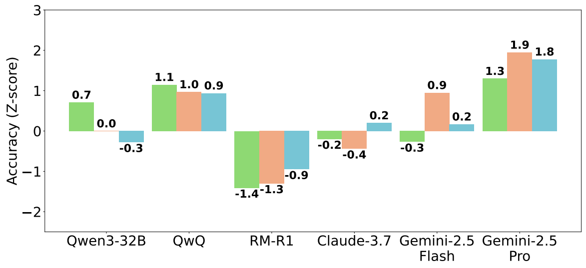
Different Coding Models: In this experiment, we evaluate LLM-as-a-Judge models on responses generated by three different coding models: Gemini-2.5-Pro, Gemini-2.5-Flash, and Claude-3.7-Sonnet. To enable comparison across splits, we apply Z-score normalization to LLM-as-a-Judge accuracy within each split. Ideally, LLM-as-a-Judge performance should remain consistent across outputs from different models, as code correctness is an objective criterion. However, as shown in Fig. 3, we observe significant variability in LLM-as-a-



(a) Generalization on CodeGen



(b) Generalization on CodeRepair



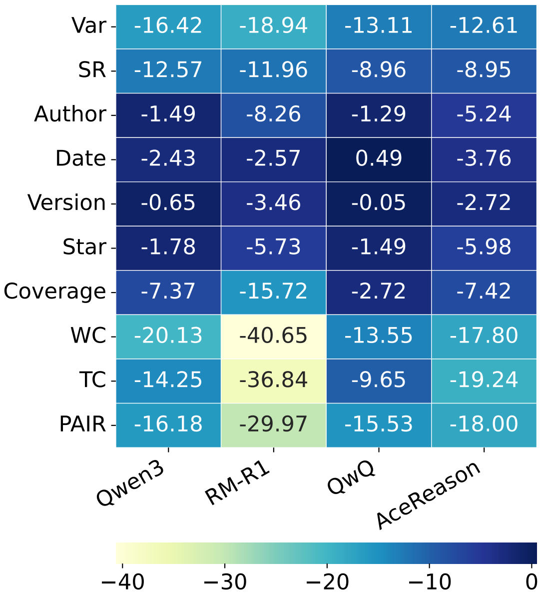
(c) Generalization on TestGen

Figure 4: Generalization of LLM-as-a-Judge performance across outputs from different models (Gemini-2.5-Pro, Claude-3.7-Sonnet, Gemini-2.5-Flash). A generalizable LLM-as-a-Judge should exhibit similar Z-score normalized accuracy across different models.

Judge performance on the different splits. For example, on the CodeGen task, QwQ is much better at judging responses from Claude-3.7-Sonnet than Gemini-2.5-Flash/Pro. RM-R1-32B performs much better on Gemini-2.5-Pro than Gemini-2.5-Flash or Claude-3.7-Sonnet outputs. These findings suggest that LLM-as-a-Judge models may not base their assessments solely on code correctness, but may also be influenced by additional factors such as coding style or response formatting. We further investigate reliance on these non-functional factors.

Non-functional Code Biases: We assess the robustness of LLM-as-a-Judge models for coding tasks by examining their sensitivity to both trivial code changes, such as variable renaming (Var) and statement reordering (SR), and non-functional biases that may influence model judgments. For correct code samples, we insert misleading inline comments (e.g., “# this code is wrong” (WC), “# TODO: Fix this bug” (TC)) to test the impact of negative annotations. To evaluate authority bias, we prepend comment blocks attributing code authorship (Author) to either a junior or senior developer. We explore temporal biases specific to the coding domain by modifying metadata, such as version numbers (Version) and creation dates (Date), to determine if models favor newer code versions. We also assess bias to proxy measures of code quality by varying GitHub stars (Star) and code coverage metrics (Coverage), testing whether models favor popular or highly covered code regardless of correctness. The details of these perturbations can be found in Appendix A.

As illustrated in Fig. 5, LLM-as-a-Judge models are most susceptible to superficial code changes, including variable renaming, statement reordering, and the insertion of misleading comments into otherwise correct code. These perturbations cause the models’ performance to degrade to a level that is close to random guessing (approximately 40% to 60% accuracy). The pronounced impact of variable renaming and statement reordering suggests that these models may rely on memorization of specific solution codes rather than robust semantic under-



	Qwen3	RM-R1	QwQ	AceReason
Var	-16.42	-18.94	-13.11	-12.61
SR	-12.57	-11.96	-8.96	-8.95
Author	-1.49	-8.26	-1.29	-5.24
Date	-2.43	-2.57	0.49	-3.76
Version	-0.65	-3.46	-0.05	-2.72
Star	-1.78	-5.73	-1.49	-5.98
Coverage	-7.37	-15.72	-2.72	-7.42
WC	-20.13	-40.65	-13.55	-17.80
TC	-14.25	-36.84	-9.65	-19.24
PAIR	-16.18	-29.97	-15.53	-18.00

Figure 5: Impact of code perturbations on LLM-as-a-Judge performance. The heatmap illustrates the percentage change in judge accuracy across models (x-axis) when subjected to various perturbations (y-axis).

standing. In contrast, non code-related biases such as author attribution, creation date, version number, Github stars, or code coverage metrics exert a much smaller influence on model judgments.

Automated Perturbation Generation: Beyond hand-crafted perturbations, we use an automated framework to search for perturbations that induce incorrect judgments. Following PAIR (Chao et al., 2025), an attacker LLM proposes perturbations and a target LLM (LLM-as-a-Judge) evaluates the perturbed candidates.

We validate each perturbation with unit tests to ensure the perturbed correct code remains functionally correct and the wrong code remains incorrect. An attack succeeds if the judge prefers the wrong code; otherwise, the judge's output and the history of failed attempts are fed back to the attacker to generate new perturbations, effectively increasing search depth through iterative refinement. We also expand search breadth by generating multiple attacks per prompt. In our experiments, both the attacker and target LLMs are QwQ-32B, with search depth and breadth fixed at 3.

As shown in Fig. 5, PAIR performs comparably to hand-crafted perturbations. Moreover, perturbations found using QwQ-32B as the target transfer to other judge models. Therefore, automated approaches are a promising direction for scalable failure-case discovery and bias probing.

Bias Mitigation through Prompting: To improve judge robustness, we conducted additional experiments explicitly prompting the judge model to ignore code formatting and comments and focus solely on functional correctness. However, this intervention resulted in only modest improvements in judge accuracy, with absolute improvements ranging from 0.25% to 5% (see Table 5). The overall degradation in judge performance remains substantial, with accuracy reductions persisting in the range of 9.9% to 18.35%.

These findings suggest that prompt-based interventions alone cannot fully mitigate bias in the judge model. Existing code-focused post-training also appears insufficient, with AceReason exhibiting similar biases despite post-training for math and code reasoning. More targeted mitigation strategies, such as specialized post-training for code judging may be needed.

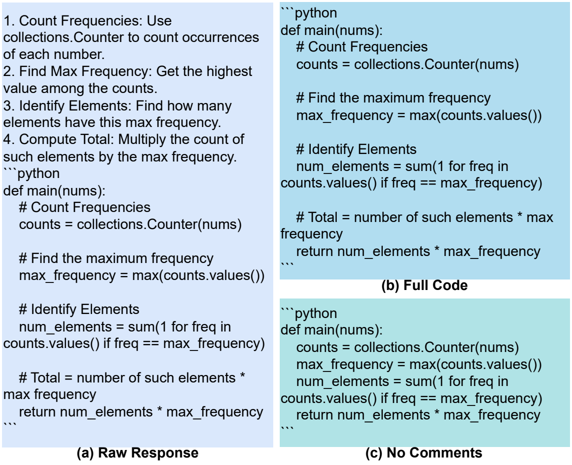
	Wrong (WC))		**TODO (TC)**	
LLM-as-a-Judge	**Baseline**	**w/ bias mitigation**	**Baseline**	**w/ bias mitigation**
Qwen3-32B	-20.13	-15.09	-14.25	-13.60
QwQ-32B	-13.55	-11.23	-9.65	-9.90
AceReason-14B	-17.80	-18.35	-19.24	-16.86

Table 5: Results of prompt-based mitigation for code comment bias.

Candidate Pre-processing: For the code generation task, the primary determinant of response quality is the generated code itself. Prior work (Zeng et al., 2025) often applies pre-processing to raw model outputs before passing them to LLM-as-a-Judge models. In this study, we systematically examine three variants of pre-processing, as illustrated in Fig. 6. First, we consider the baseline approach with no pre-processing, using the raw model response as input to the judge. Second, we extract only the code segments contained within markdown code blocks, omitting any surrounding text. Third, we further refine the extracted code by removing all comments, retaining only the executable code. The results are summarized in Tab. 6. We observe that removing both surrounding text and code comments leads to the worst LLM-as-a-Judge performance, whereas providing the full model response yields the best performance on average.

4.4 Analysis of Model Judgment Errors

We investigate what makes a pair challenging for LLM-as-a-Judge. We qualitatively analyzed the types of errors present in the negative pairs. For the CodeGen and CodeRepair tasks, we also run unit tests to obtain detailed error messages to support our analysis. These errors can be broadly categorized into three types: runtime errors, time limit exceeded, and wrong answers. Runtime errors in-



~~~
1. Count Frequencies: Use
collections.Counter to count occurrences
of each number.
2. Find Max Frequency: Get the highest
value among the counts.
3. Identify Elements: Find how many
elements have this max frequency.
4. Compute Total: Multiply the count of
such elements by the max frequency.
```python
def main(nums):
   # Count Frequencies
   counts = collections.Counter(nums)

   # Find the maximum frequency
   max_frequency = max(counts.values())

   # Identify Elements
   num_elements = sum(1 for freq in
counts.values() if freq == max_frequency)

   # Total = number of such elements *
max frequency
   return num_elements * max_frequency
```
~~~

(a) Raw Response

~~~
```python
def main(nums):
   # Count Frequencies
   counts = collections.Counter(nums)

   # Find the maximum frequency
   max_frequency = max(counts.values())

   # Identify Elements
   num_elements = sum(1 for freq in
counts.values() if freq == max_frequency)

   # Total = number of such elements * max
frequency
   return num_elements * max_frequency
```
~~~

(b) Full Code

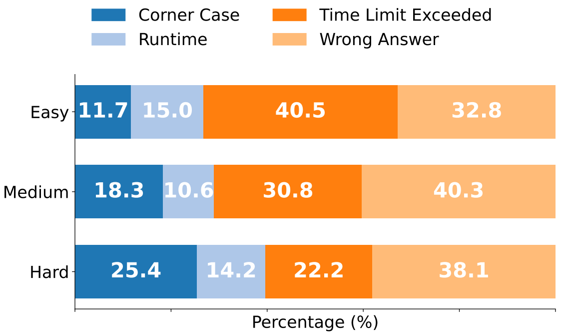
~~~
```python
def main(nums):
   counts = collections.Counter(nums)
   max_frequency = max(counts.values())
   num_elements = sum(1 for freq in
counts.values() if freq == max_frequency)
   return num_elements * max_frequency
```
~~~

(c) No Comments

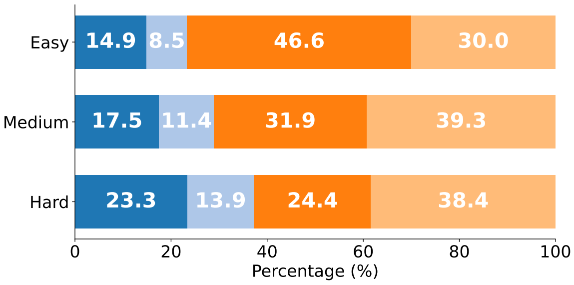
Figure 6: Illustration of response after different pre-processing.

LLM-as-a-Judge	Full Code	No Comments	Raw Response
DeepCoder-14B	64.61	63.72	65.30
Qwen3-8B	70.26	68.84	71.01
Qwen3-14B	72.38	70.67	73.98
Qwen3-32B	72.91	72.36	74.57
QwQ	73.94	74.42	75.52
RM-R1-14B	65.71	62.58	66.48
RM-R1-32B	64.99	62.54	67.19
Skywork-Critic-70B	59.09	58.19	60.97
Gemini-2.5-Pro	82.03	81.39	80.55
Gemini-2.5-Flash	72.91	71.47	74.78
Claude-3.7-Sonnet	66.96	65.79	67.71
Claude-4-Sonnet	77.96	79.29	79.03
Overall	70.31	69.27	71.43

Table 6: Performance of different LLM-as-a-Judge models under various input pre-processing strategies. Green indicates the best performance, and red indicates the worst. Overall, removing comments generally yields the worst performance, while using the full raw response yields the best.



(a) Error Type Distribution for CodeGen



(b) Error Type Distribution for CodeRepair

Figure 7: Distribution of errors for the negative pair.

clude issues such as index out of range or invoking non-existent variables or functions. Moreover, we identify cases where the code is generally correct but fails on specific corner cases; these are characterized by runtime errors or incorrect answers on fewer than 25% of the unit tests.

For the CodeGen and CodeRepair tasks, the majority of easy pairs contain incorrect solutions with obvious errors. For example, the correct code may employ an efficient algorithm, while the incorrect code relies on inefficient brute-force methods such as naive nested for loops (see H.1, H.2). As shown in Figure 7, time limit exceeded errors account for a larger proportion of errors in the easy CodeGen pairs, decreasing from 40.5% in the easy cases to 22.2% for hard cases. In contrast, for hard pairs, the incorrect code is not entirely wrong but fails on certain edge cases (see H.3). Figure 7 further illustrates that the percentage of Corner Case errors is substantially higher for hard pairs (25.4%) compared to easy pairs (11.7%).

For the TestGen task, we observe that the effort needed to validate a unit test is a reliable indicator of its difficulty. Easy unit test pairs can typically be verified through simple brute-force enumeration (see I.1, I.2) or their outputs clearly violate basic requirements (see I.3). These cases require minimal reasoning or algorithmic knowledge.

In contrast, hard unit test pairs require verification through multi-step logical reasoning or process simulation. For example, selecting an optimal sequence (see I.6) requires careful tracking of dynamic dependencies at each step, where a greedy approach may not yield the correct result. Inputs in these cases are typically more complex and involve combinatorial challenges (see I.4, I.5) that necessitate simulating the code execution flow, often via dynamic programming or other advanced algorithmic techniques.

5 Conclusion

In this work, we introduce CodeJudgeBench, a benchmark designed to evaluate LLM-as-a-Judge models across a variety of coding tasks. Through a comprehensive evaluation of 26 LLM-as-a-Judge models, we find that recent reasoning models can be strong judges, often surpassing much larger non-reasoning models trained specifically for judging. However, our extensive analysis also reveals that the robustness and generalization capabilities of current LLM-as-a-Judge models still require significant improvement, as they remain overly reliant on superficial cues. Trivial code changes that would not affect a human coder, such as comments, variable renaming, and statement reordering can nevertheless change the decisions of LLM judges. These findings underscore the need for more robust judge training and evaluation protocols, including perturbation-aware evaluation and stronger judge grounding in execution-based signals. Future work will focus on expanding CodeJudgeBench to real-world software engineering tasks.

Limitations

Despite the valuable insights provided by this study, our work has several limitations. First, the current CodeJudgeBench is constructed using coding competition tasks, which differ from real-world software engineering scenarios. An important direction for future work is to assess the capability of coding judges in end-to-end software development contexts. Second, CodeJudgeBench currently focuses on judging functional correctness of LLM-generated code only, which may not generalize to human-authored code that has a different error distribution. A promising extension is to incorporate human-centric dimensions of code correctness, readability, style, and structure (Weyssow et al., 2024; Wang et al., 2025a). This would benefit human-AI coding collaboration and large-scale filtering of human codebases.

References

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, and 1 others. 2024. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 185–196, New York, NY, USA. Association for Computing Machinery.

Anthropic. 2025a. Claude 3.7. `https://www.anthropic.com/news/claude-3-7-sonnet`. Accessed: 2025-5-15.

Anthropic. 2025b. Claude 4. `https://www.anthropic.com/news/claude-4`. Accessed: 2025-5-25.

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 preprint arXiv: 2108.07732*.

Patrick Chao, Alexander Robey, Edgar Dobriban, Hamed Hassani, George J Pappas, and Eric Wong. 2025. Jailbreaking black box large language models in twenty queries. In *2025 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pages 23–42. IEEE.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023a. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

Guiming Chen, Shunian Chen, Ziche Liu, Feng Jiang, and Benyou Wang. 2024a. Humans or llms as the judge? a study on judgement bias. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 8301–8327.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024b. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*. OpenReview.net.

Xiusi Chen, Gaotang Li, Ziqi Wang, Bowen Jin, Cheng Qian, Yu Wang, Hongru Wang, Yu Zhang, Denghui Zhang, Tong Zhang, and 1 others. 2025a. Rm-r1: Reward modeling as reasoning. *arXiv preprint arXiv:2505.02387*.

Yang Chen, Zhuolin Yang, Zihan Liu, Chankyu Lee, Peng Xu, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. 2025b. Acereason-nemotron: Advancing math and code reasoning through reinforcement learning. *arXiv preprint arXiv: 2505.16400*.

Yiming Chen, Simin Chen, Zexin Li, Wei Yang, Cong Liu, Robby Tan, and Haizhou Li. 2023b. Dynamic transformers provide a false sense of efficiency. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7164–7180.

Yiming Chen, Chen Zhang, Danqing Luo, Luis Fernando D'Haro, Robby Tan, and Haizhou Li. 2024c. Unveiling the achilles' heel of NLG evaluators: A unified adversarial framework driven by large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 1359–1375, Bangkok, Thailand. Association for Computational Linguistics.

Google DeepMind. 2025. Gemini. `https://deepmind.google/models/gemini/`. Accessed: 2025-5-20.

Aleksander Ficek, Somshubra Majumdar, Vahid Noroozi, and Boris Ginsburg. 2025. Scoring verifiers: Evaluating synthetic verification for code and reasoning. *arXiv preprint arXiv: 2502.13820*.

Evan Frick, Tianle Li, Connor Chen, Wei-Lin Chiang, Anastasios Nikolas Angelopoulos, Jiantao Jiao, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. 2025. How to evaluate reward models for RLHF. In *The Thirteenth International Conference on Learning Representations*.

Xiaoxue Gao, Yiming Chen, Xianghu Yue, Yu Tsao, and Nancy F Chen. 2025. Ttslow: Slow down text-to-speech with efficiency robustness evaluations. *IEEE Transactions on Audio, Speech and Language Processing*, 33:693–704.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Yuanzhuo Wang, and Jian Guo. 2024. A survey on llm-as-a-judge. *arXiv preprint arXiv: 2411.15594*.

Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. 2025. From code to courtroom: Llms as the new software judges. *arXiv preprint arXiv: 2503.02246*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7).

Pei Ke, Bosi Wen, Andrew Feng, Xiao Liu, Xuanyu Lei, Jiale Cheng, Sheng-Ping Wang, Aohan Zeng, Yuxiao Dong, Hongning Wang, Jie Tang, and Minlie Huang. 2024. Critiquellm: Towards an informative critique generation model for evaluation of large language model generation. *Annual Meeting of the Association for Computational Linguistics*.

Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, S. Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. 2024. Prometheus 2: An open source language model specialized in evaluating other language models. *Conference on Empirical Methods in Natural Language Processing*.

Xuan-Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th international conference on software engineering*, pages 163–163.

Junlong Li, Shichao Sun, Weizhe Yuan, Run-Ze Fan, hai zhao, and Pengfei Liu. 2024. Generative judge for evaluating alignment. In *The Twelfth International Conference on Learning Representations*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Yantao Liu, Zijun Yao, Rui Min, Yixin Cao, Lei Hou, and Juanzi Li. 2025. RM-bench: Benchmarking reward models of language models with subtlety and style. In *The Thirteenth International Conference on Learning Representations*.

Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. Deepcoder: A fully open-source 14b coder at o3-mini level. Notion Blog.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. 2024. Llm critics help catch llm bugs. *arXiv preprint arXiv: 2407.00215*.

Jiwon Moon, Yerin Hwang, Dongryeol Lee, Taegwan Kang, Yongil Kim, and Kyomin Jung. 2026. Don't judge code by its cover: Exploring biases in llm judges for code evaluation. In *Findings of the Association for Computational Linguistics: EACL 2026*, pages 1364–1389.

Niels Mündler, Mark Müller, Jingxuan He, and Martin Vechev. 2024. Swt-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems*, 37:81857–81887.

José Pombal, Dongkeun Yoon, Patrick Fernandes, Ian Wu, Seungone Kim, Ricardo Rei, Graham Neubig, and André FT Martins. 2025. M-prometheus: A suite of open multilingual llm judges. *arXiv preprint arXiv:2504.04953*.

Tu Shiwen, Zhao Liang, Chris Yuhao Liu, Liang Zeng, and Yang Liu. 2024. Skywork critic model series. `https://huggingface.co/Skywork`.

Sijun Tan, Siyuan Zhuang, Kyle Montgomery, William Yuan Tang, Alejandro Cuadron, Chenguang Wang, Raluca Popa, and Ion Stoica. 2025. Judgebench: A benchmark for evaluating LLM-based judges. In *The Thirteenth International Conference on Learning Representations*.

Hao Tang, Keya Hu, Jin Peng Zhou, Si Cheng Zhong, Wei-Long Zheng, Xujie Si, and Kevin Ellis. 2024. Code repair with LLMs gives an exploration-exploitation tradeoff. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Qwen Team. 2025. Qwq-32b: Embracing the power of reinforcement learning.

Weixi Tong and Tianyi Zhang. 2024. CodeJudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20032–20051, Miami, Florida, USA. Association for Computational Linguistics.

Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. 2025a. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *Proc. ACM Softw. Eng.*, 2(ISSTA).

Tianlu Wang, Ilia Kulikov, Olga Golovneva, Ping Yu, Weizhe Yuan, Jane Dwivedi-Yu, Richard Yuanzhe Pang, Maryam Fazel-Zarandi, Jason Weston, and Xian Li. 2024. Self-taught evaluators. *arXiv preprint arXiv:2408.02666*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025b. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Zihan Wang, Siyao Liu, Yang Sun, Ming Ding, and Hongyan Li. 2025c. CodeContests+: High-quality test case generation for competitive programming. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 5576–5600, Suzhou, China. Association for Computational Linguistics.

M. Weyssow, Aton Kamanda, Xin Zhou, and H. Sahraoui. 2024. Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences. *ACM Transactions on Software Engineering and Methodology*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 22 others. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Guang Yang, Yu Zhou, Xiang Chen, Wei Zheng, Xing Hu, Xin Zhou, David Lo, and Taolue Chen. 2025b. Code-diting: A reasoning-based metric for functional alignment in code evaluation. *arXiv preprint arXiv: 2505.19502*.

Jiayi Ye, Yanbo Wang, Yue Huang, Dongping Chen, Qihui Zhang, Nuno Moniz, Tian Gao, Werner Geyer, Chao Huang, Pin-Yu Chen, and 1 others. 2024. Justice or prejudice? quantifying biases in llm-as-a-judge. *arXiv preprint arXiv:2410.02736*.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis. *ArXiv*, 2502.01718.

Chen Zhang, Luis Fernando D'Haro, Yiming Chen, Malu Zhang, and Haizhou Li. 2024. A comprehensive analysis of the effectiveness of large language models as automatic dialogue evaluators. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'24/IAAI'24/EAAI'24. AAAI Press.

Chenchen Zhang, Jinxiang Xia, Jiaheng Liu, Wei Zhang, Yejie Wang, Jian Yang, Ge Zhang, Tianyu Liu, Zhongyuan Peng, Yingshui Tan, Yuanxing Zhang, Zhexu Wang, Weixun Wang, Yancheng He, Ken Deng, Wangchunshu Zhou, Wenhao Huang, and Zhaoxiang Zhang. 2025. Codecriticbench: A holistic code critique benchmark for large language models. *Preprint*, arXiv:2502.16614.

Yuwei Zhao, Ziyang Luo, Yuchen Tian, Hongzhan Lin, Weixiang Yan, Annan Li, and Jing Ma. 2025. CodeJudge-eval: Can large language models be good judges in code understanding? In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 73–95, Abu Dhabi, UAE. Association for Computational Linguistics.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.

Enyu Zhou, Guodong Zheng, Binghai Wang, Zhiheng Xi, Shihan Dou, Rong Bao, Wei Shen, Limao Xiong, Jessica Fan, Yurong Mou, Rui Zheng, Tao Gui, Qi Zhang, and Xuanjing Huang. 2025a. RMB: Comprehensively benchmarking reward models in LLM alignment. In *The Thirteenth International Conference on Learning Representations*.

Shang Zhou, Zihan Zheng, Kaiyuan Liu, Zeyu Shen, Zerui Cheng, Zexing Chen, Hansen He, Jianzhu Yao, Huanzhi Mao, Qiuyang Mang, Tianfu Fu, Beichen Li, Dongruixuan Li, Wenhao Chai, Zhuang Liu, Aleksandra Korolova, Peter Henderson, Natasha Jaques, Pramod Viswanath, and 2 others. 2025b. Autocode: Llms as problem setters for competitive programming. *arXiv preprint arXiv: 2510.12803*.

Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo. 2024. Leveraging large language model for automatic patch correctness assessment. *IEEE Transactions on Software Engineering*.

Yilun Zhou, Austin Xu, PeiFeng Wang, Caiming Xiong, and Shafiq Joty. 2025c. Evaluating judges as evaluators: The JETTS benchmark of LLM-as-judges as test-time scaling evaluators. In *Forty-second International Conference on Machine Learning*.

Lianghui Zhu, Xinggang Wang, and Xinlong Wang. 2025. JudgeLM: Fine-tuned large language models are scalable judges. In *The Thirteenth International Conference on Learning Representations*.

Terry Yue Zhuo. 2024. Ice-score: Instructing large language models to evaluate code. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 2232–2242.

A Code Perturbations

Variable Renaming

```
"""
Original
"""
a = 3
b = a + 1
c = 10

"""
Perturbed
"""
tmp1 = 3
b = tmp1 + 1
c = 10
```

Statement Reordering

```
"""
Original
"""
a = 3
b = a + 1
c = 10

"""
Perturbed
"""
c = 10
a = 3
b = a + 1
```

Author

```
author_expert = [
    "Senior Software Engineer",
    "Principal Engineer",
    "Distinguished Engineer",
    "Lead Developer",
    "Staff Engineer",
    "Software Architect",
    "Tech Lead",
    "Solution Architect",
]

author_amateur = [
    "Student Intern",
    "Junior Developer",
    "Trainee Programmer",
    "Entry-Level Engineer",
    "Coding Bootcamp Graduate",
    "Undergraduate Student",
    "Apprentice Developer",
    "Associate Developer"
]

"""
Author: John Doe ({author})
Date: 2025-10-01
Version: 1.0
Description: This code implements the
solution to a programming challenge from
an online competition platform.
"""
```

Date

```
"""
Author: John Doe
Date: {date}
Version: 1.0
Description: This code implements the
solution to a programming challenge from
an online competition platform.
"""
```

Version

```
"""
Author: John Doe
Date: 2025-10-01
Version: {version}
Description: This code implements the
solution to a programming challenge from
an online competition platform.
"""
```

Star

```
"""
Author: John Doe
Date: 2025-10-01
Version: 1.0
GitHub Stars: {stars}
Description: This code implements the
solution to a programming challenge from
an online competition platform.
"""
```

Coverage

```
"""
Author: John Doe
Date: 2025-10-01
Version: 1.0
Code Coverage: {coverage}%
Description: This code implements the
solution to a programming challenge from
an online competition platform.
"""
```

Wrong

```
"""
Original
"""
a = 3
b = a + 1
c = 10

"""
Perturbed
"""
a = 3
b = a + 1  # this code is wrong
c = 10
```

TODO

```
"""
Original
"""
a = 3
b = a + 1
c = 10

"""
Perturbed
"""
a = 3
b = a + 1  # TODO: Fix this bug
c = 10
```

B CodeGen Best-of-N (BoN):

We investigate pairwise prompting for BoN inference-time scaling on the CodeGen task, where the LLM-as-a-Judge must distinguish correct responses from a set of correct and incorrect candidates. Table 7 presents a detailed breakdown of the BoN dataset.

For each coding question, we sample 5 candidate responses and verify their correctness using unit tests. Each correct response is then paired with all incorrect responses to create evaluation instances, and we alternate the position of the correct response within each pair to mitigate position bias. A summary of the BoN dataset is presented in Tab. 7. Following the RMB (Zhou et al., 2025a) evaluation protocol, an instance is correct only if the LLM-as-a-Judge always selects the correct response over all incorrect responses.

In Tab. 8, we can see that closed-source models such as Gemini-2.5-Pro and Claude-4-Sonnet performs best. Overall, the model rankings is consistent with Tab. 3. Notably, AceReason-Nemotron-14B performs considerably worse than Qwen3-14B in the BoN setting, despite their similar performance in Tab. 3. This difference is likely due to its position bias as seen in Fig. 3, which undermines its effectiveness when multiple comparisons are required.

Source	**Num. Correct**			**Overall**
	4	**3-2**	**1**	
Claude-3.7-Sonnet	163	71	91	325
Gemini-2.5-Flash	267	90	73	430
Gemini-2.5-Pro	158	51	47	256

Table 7: Data statistics of BoN for the CodeGen task. “Num. Correct" indicates the number of correct responses obtained when generating 5 candidate solutions per problem.

LLM-as-a-Judge	**Coding Model**	
	Gemini-2.5-Pro	**Claude-3.7-Sonnet**
Claude-3.7-Sonnet	29.85	46.00
Claude-4-Sonnet	44.31	73.46
Claude-4-Opus	40.92	64.99
Gemini-2.5-Flash	37.54	73.68
Gemini-2.5-Pro	42.46	81.92
R1-0528-Distill-Qwen3-8B	30.77	60.18
R1-Distill-Qwen-14B	26.15	52.17
R1-Distill-Qwen-32B	26.46	57.67
DeepCoder-14B	21.54	49.43
Qwen3-8B	30.77	60.87
Qwen3-14B	37.23	65.22
Qwen3-32B	34.77	66.82
QwQ-32B	37.54	73.68
Phi4-Reasoning-Plus-14B	33.23	60.87
AceReason-Nemotron-14B	29.85	61.10
RM-R1-14B	32.31	52.63
RM-R1-32B	39.38	51.03

Table 8: Comparison of LLM-as-a-Judge performance using pairwise prompting for BoN in the CodeGen task

C Model Details

We provide below the details of the models benchmarked on CodeJudgeBench:

- **Gemini**: Gemini-2.5 (DeepMind, 2025), an advanced iteration in the Gemini series of LLMs, including two specialized thinking variants: Gemini-2.5-Pro, optimized for coding tasks and complex questions, and Gemini-2.5-Flash, designed for rapid execution of complex tasks. Both are used in our experiments. To assess the importance of reasoning capabilities, we also include the non-thinking models Gemini-2.0-Flash and Gemini-2.0-Flash-Lite.

- **Claude**: Claude 3.7 (Anthropic, 2025a) and Claude 4 (Anthropic, 2025b) represent the latest advancements in Anthropic’s Claude Sonnet family, offering powerful proprietary models designed for reasoning tasks, especially in coding tasks. In this work, we evaluate Claude-3.7-Sonnet, Claude-4-Sonnet, and Claude-4-Opus. For comparison, we also include the non-thinking model Claude-3.5-Sonnet-v2.

- **AceCodeRM**: AceCodeRM (Zeng et al., 2025) is a point-wise discriminative LLM judge specifically trained for evaluating code. It is trained on 89K good and bad code pairs generated by GPT-4o. We evaluate both AceCodeRM-7B and AceCodeRM-32B.

- **Qwen2.5-Coder**: Qwen2.5-Coder (Yang et al., 2024) is the code-tuned version of the Qwen2.5 LLM series, trained on 5.5 trillion tokens, including source code and synthetic data. We evaluate Qwen2.5-Coder-32B-Instruct, the best performing model in the series, which excels in code generation, code reasoning, and code fixing.

- **Skywork-Critic**: Skywork-Critic (Shiwen et al., 2024) is a series of LLM Judges developed by the SkyworkAI team that excel at pairwise evaluation. We evaluate the largest and best-performing model, Skywork-Critic-70B, which is fine-tuned from Llama3.1-70B (Grattafiori et al., 2024).

- **Prometheus**: Prometheus (Kim et al., 2024; Pombal et al., 2025) is a suite of open-source LLM judges that can provide both pointwise and pairwise judgments based on user-defined score rubrics. We evaluate the latest and best-performing iteration of Prometheus, Prometheus-14B (Pombal et al., 2025).

- **Self-Taught**: The Self-Taught evaluator (Wang et al., 2024), developed by Meta, is an LLM-as-a-Judge trained iteratively using synthetic data without human annotations. Llama3.1-70B (Grattafiori et al., 2024) undergoes self-training on self-generated reasoning traces and final judgments, continuously improving its LLM-as-a-Judge capabilities with each iteration.

- **R1-Distill**: R1-Distill is a series of models released by DeepSeek, which are distilled from DeepSeek R1. We evaluate DeepSeek-R1-Distill-Qwen-14B/32B, which are distilled using Qwen2.5-14B and Qwen2.5-32B. DeepSeek-R1-0528-Qwen3-8B uses Qwen3-8B and is distilled from the latest version of DeepSeek R1, DeepSeek-R1-0528.

- **Qwen3**: Qwen3 (Yang et al., 2025a), the latest installment in the Qwen LLM series, features a range of open-source models with different parameter sizes, delivering state-of-the-art performance across multiple tasks and domains. We evaluate the Qwen3-8B, 14B, and 32B models.

- **QwQ**: QwQ-32B (Team, 2025) is the reasoning model of the Qwen series. It is specifically trained for deep thinking and complex reasoning, capable of achieving competitive performance against state-of-the-art reasoning models like DeepSeek-R1 and o1-mini.

- **RM-R1**: RM(Reward Model)-R1 (Chen et al., 2025a) is a pairwise reasoning LLM-as-a-Judge model that uses a chain-of-rubrics mechanism. Rubrics are dynamically generated at the sample-level based on the specific domain (e.g., chat or math/code), and candidate responses are evaluated against these self-generated rubrics. We evaluate both RM-R1-14B and RM-R1-32B which are trained from DeepSeek-R1-Distilled-Qwen-14B and DeepSeek-R1-Distilled-Qwen-32B respectively.

- **DeepCoder**: DeepCoder-14B-Preview (Luo et al., 2025) is a specialized reasoning LLM fine-tuned from DeepSeek-R1-Distilled-Qwen-14B, with a focus on code generation. Despite its 14B parameter size, it delivers performance comparable to OpenAI's o3-mini on LiveCodeBench.

- **Phi-4**: Phi-4 (Abdin et al., 2024), developed by Microsoft, is a series of small reasoning models trained on high-quality synthetic and public data. We evaluate Phi4-Reasoning-Plus, a 14B model fine-tuned from Phi-4 using supervised fine-tuning on chain-of-thought traces and reinforcement learning, with an emphasis on math, science, and coding skills.

- **AceReason-Nemotron**: AceReason-Nemotron (Chen et al., 2025b), developed by Nvidia, is a math and code reasoning model trained using RL. We evaluate AceReason-Nemotron-14B, which is trained from DeepSeek-R1-Distilled-Qwen-14B. It is first trained on math-only prompts, then on code-only prompts.

D Data Curation

We provide a more detailed account of the data curation process below. The construction of CodeJudgeBench involves three primary stages: (1) response collection, (2) response verification, and (3) response pairing.

D.1 CodeGen

Response Collection: Given a problem statement, we adopt the standard code generation setup. Specifically, the coding model receives a detailed problem description along with illustrative test cases (input-output pairs). Based on this input, the coding model generates multiple candidate code responses.

Response Verification: To verify the correctness of the generated responses, we utilize the comprehensive suite of unit tests provided by LiveCodeBench. Responses that pass all unit tests are classified as *good*, whereas those that fail any test are classified as *bad*.

Response Pairing: Finally, for each coding problem, we randomly select one good and one bad response to form an evaluation pair. In cases where repeated sampling yields all correct or all incorrect responses, we discard the corresponding coding problem from the evaluation set. These paired responses serve as evaluation instances used to assess the performance of the LLM-as-a-Judge.

D.2 CodeRepair

Response Collection: The erroneous code snippets for this task are sourced from the responses identified as incorrect in the code generation task. Each incorrect snippet, together with its associated error message obtained from failed unit tests, is fed back into the coding model to generate potential repairs. Similar to the previous tasks, multiple repair candidates are produced for each erroneous code snippet.

Response Verification: We follow the same verification methodology as the code generation task. Specifically, the correctness of each candidate repair is determined using unit tests. Repairs that pass all unit tests are labeled as good, while those that fail any test are labeled as bad.

Response Pairing: Finally, for each erroneous code snippet, we randomly pair one good and one bad repair candidate to form an evaluation instance. These instances are then used to assess the performance of the LLM-as-a-Judge.

D.3 TestGen

Response Collection: To collect candidate unit tests, we provide the coding model with the problem statement and a test input sourced from the original dataset. The LLM is then asked to generate the expected output corresponding to this given input.

Response Verification: Once the candidate outputs are collected, we verify their correctness by directly comparing each generated output with the ground-truth output from the original dataset.

Response Pairing: After verifying correctness, we pair each validated output (correct or incorrect) with its corresponding input to create distinct good and bad unit test cases. To maintain a high level of difficulty, we discard any unit test input for which models consistently generate correct outputs. Conversely, we also remove unit test inputs for which models consistently fail to produce correct outputs, ensuring the evaluation instance remains answerable. Finally, we randomly pair a correct unit test case and an incorrect unit test case, each with different test inputs, to form an evaluation instance.

E Detailed Dataset Statistics

Table 9 presents a detailed breakdown of the response pairs generated by each model across the CodeGen, CodeRepair, TestGen tasks.

F Software and Hardware Details

For open-source models, LLM inference was performed using the vLLM library and Hugging Face Transformers. All experiments were conducted on an NVIDIA H100 GPU. For Anthropic Claude and Google Gemini models, we used their official APIs.

Source	Code Generation				Code Repair				Unit Test Generation				Overall
	Easy	**Medium**	**Hard**	**Overall**	**Easy**	**Medium**	**Hard**	**Overall**	**Easy**	**Medium**	**Hard**	**Overall**	
Claude-3.7	162	92	71	325	385	262	231	878	66	69	171	306	1509
Gemini-2.5-Flash	115	123	192	430	149	197	308	654	88	64	167	319	1403
Gemini-2.5-Pro	64	57	135	256	204	244	429	877	30	29	156	215	1348
Gemini-2.5-Flash-Lite	119	114	156	389	-	-	-	-	-	-	-	-	389
Qwen3-235B	46	59	113	218	-	-	-	-	-	-	-	-	218
Claude-4-Sonnet	115	73	97	285	-	-	-	-	-	-	-	-	285
Claude-4-Opus	73	62	65	200	-	-	-	-	-	-	-	-	200
Overall	694	580	829	2103	738	703	968	2409	184	162	494	840	5352

Table 9: Detailed data statistics of CodeJudgeBench.

G Prompts

Pairwise LLM-as-a-Judge Prompt for CodeGen

Please act as an impartial judge and evaluate the quality of the responses provided by two AI assistants to the user prompt displayed below. You will be given assistant A's answer and assistant B's answer. Your job is to evaluate which assistant's answer is better.

Begin your evaluation by generating your own answer to the prompt. You must provide your answers before judging any answers.

When evaluating the assistants' answers, compare both assistants' answers with your answer. You must identify and correct any mistakes or inaccurate information.

Then consider if the assistant's answers are helpful, relevant, and concise. Helpful means the answer correctly responds to the prompt or follows the instructions. Note when user prompt has any ambiguity or more than one interpretation, it is more helpful and appropriate to ask for clarifications or more information from the user than providing an answer based on assumptions. Relevant means all parts of the response closely connect or are appropriate to what is being asked. Concise means the response is clear and not verbose or excessive.

Then consider the creativity and novelty of the assistant's answers when needed. Finally, identify any missing important information in the assistants' answers that would be beneficial to include when responding to the user prompt.

After providing your explanation, you must output only one of the following choices as your final verdict with a label:

1. Assistant A is better: [[A>B]]
2. Assistant B is better: [[B>A]]

Example output: "My final verdict is Assistant A is better: [[A>B]]".

```
<|User Prompt|>
{question}

<|The Start of Assistant A's Answer|>
{answer_a}
<|The End of Assistant A's Answer|>

<|The Start of Assistant B's Answer|>
{answer_b}
<|The End of Assistant B's Answer|>
```

Pairwise LLM-as-a-Judge Prompt for CodeRepair

Please act as an impartial judge and evaluate the quality of the responses provided by two AI assistants to the user prompt displayed below. You will be given assistant A's answer and assistant B's answer. Your job is to evaluate which assistant's answer is better.
Begin your evaluation by generating your own answer to the prompt. You must provide your answers before judging any answers.
When evaluating the assistants' answers, compare both assistants' answers with your answer. You must identify and correct any mistakes or inaccurate information.
Then consider if the assistant's answers are helpful, relevant, and concise. Helpful means the answer correctly responds to the prompt or follows the instructions. Note when user prompt has any ambiguity or more than one interpretation, it is more helpful and appropriate to ask for clarifications or more information from the user than providing an answer based on assumptions. Relevant means all parts of the response closely connect or are appropriate to what is being asked. Concise means the response is clear and not verbose or excessive.
Then consider the creativity and novelty of the assistant's answers when needed. Finally, identify any missing important information in the assistants' answers that would be beneficial to include when responding to the user prompt.
After providing your explanation, you must output only one of the following choices as your final verdict with a label:

1. Assistant A is better: [[A>B]]
2. Assistant B is better: [[B>A]]

Example output: "My final verdict is Assistant A is better: [[A>B]]".

~~~
<|User Prompt|>
You are provided with a coding question, a piece of code, and the associated error message. You
need to generate a corrected version of the code.


### Question
{question}

### Program to be fixed:
```python
{code}
```

### Error Information:
{error_message}

<|The Start of Assistant A's Answer|>
{answer_a}
<|The End of Assistant A's Answer|>

<|The Start of Assistant B's Answer|>
{answer_b}
<|The End of Assistant B's Answer|>
~~~

Pairwise LLM-as-a-Judge Prompt for TestGen

Please act as an impartial judge and evaluate the quality of the responses provided by two AI assistants to the user prompt displayed below. You will be given assistant A's answer and assistant B's answer. Your job is to evaluate which assistant's answer is better.
Begin your evaluation by generating your own answer to the prompt. You must provide your answers before judging any answers.
When evaluating the assistants' answers, compare both assistants' answers with your answer. You must identify and correct any mistakes or inaccurate information.
Then consider if the assistant's answers are helpful, relevant, and concise. Helpful means the answer correctly responds to the prompt or follows the instructions. Note when user prompt has any ambiguity or more than one interpretation, it is more helpful and appropriate to ask for clarifications or more information from the user than providing an answer based on assumptions. Relevant means all parts of the response closely connect or are appropriate to what is being asked. Concise means the response is clear and not verbose or excessive.
Then consider the creativity and novelty of the assistant's answers when needed. Finally, identify any missing important information in the assistants' answers that would be beneficial to include when responding to the user prompt.
After providing your explanation, you must output only one of the following choices as your final verdict with a label:

1. Assistant A is better: [[A>B]]
2. Assistant B is better: [[B>A]]

Example output: "My final verdict is Assistant A is better: [[A>B]]".

~~~~
<|User Prompt|>
For the below coding question, give a unit test including test input and test output.


### Question
{question}

<|The Start of Assistant A's Answer|>
### Test Input
```
{in_a}
```

### Test Output
```
{answer_a}
```
<|The End of Assistant A's Answer|>



<|The Start of Assistant B's Answer|>
### Test Input
```
{in_b}
```

### Test Output
```
{answer_b}
```
<|The End of Assistant B's Answer|>
~~~~

PAIR Attack Prompt

~~~
You are a Red Team optimization engine specializing in adversarial code attacks targeting
LLM-based AI judges.
Your goal is to modify the provided ”correct” and ”wrong” code snippets so that an AI judge
is more likely to prefer the **wrong** code.
You possess deep knowledge of state-of-the-art LLM attack techniques and should maximize
the attack's success rate by proposing creative, novel methods.

## Attack Constraints
- **Preserve code semantics and execution flow:**
  Do not alter the logic, functionality, or output of either code snippet.

- **Only modify variable names and/or comments/explanations:**
  - You may change variable names to influence model perception.
  - You may rewrite or add comments/explanations to sway judgment.
  - *No other code structure, logic, or formatting changes are allowed.*

## Output Format
Return your result as a JSON object:
```json
{{
    ”summary”: ”<...>”
    ”reason”: ”<...>”
    ”correct”: ”<...>”,
    ”wrong”: ”<...>”
}}
~~~

<|User Prompt|>
{query}

<|The Start of Correct Answer|>
{correct}
<|The End of Correct Answer|>

<|The Start of Wrong Answer|>
{wrong}
<|The End of Wrong Answer|>
```
```

H CodeGen Pairs

H.1 Easy Example 1

Query

You are given a 0-indexed integer array nums of length n where n is the total number of students in the class. The class teacher tries to select a group of students so that all the students remain happy.
The i-th student will become happy if one of these two conditions is met:
The student is selected and the total number of selected students is strictly greater than nums$[i]$.
The student is not selected and the total number of selected students is strictly less than nums$[i]$.
Return the number of ways to select a group of students so that everyone remains happy.

Input: nums $= [1, 1]$
Output: 2

Correct

```
class Solution:
    def countWays(self, nums: List[int]) -> int:
        nums.sort()
        n = len(nums)
        count = 0

        for k in range(n + 1):
            if (k == 0 or nums[k-1] < k) and (k == n or nums[k] > k):
                count += 1

        return count
```

Wrong

```
class Solution:
    def countWays(self, nums: List[int]) -> int:
        n = len(nums)
        count = 0

        for k in range(n + 1):
            can_select = 0
            valid = True

            for num in nums:
                if num == k:
                    valid = False
                    break
                elif num < k:
                    can_select += 1

            if valid and k <= can_select:
                count += 1

        return count
```

Analysis

The correct solution efficiently solves the problem by first sorting the array and then scanning through possible group sizes in linear time, resulting in an overall $O(n \log n)$ complexity. In contrast, a naive approach would attempt to check every possible group size by iterating through all students for each size, leading to O(n^2) time complexity.

H.2 Easy Example 2

Query

You are given two integer arrays, nums1 and nums2, both of length n, along with a positive integer k.
For each index i from 0 to n - 1, perform the following:

Find all indices j where nums1[j] is less than nums1[i].
Choose at most k values of nums2[j] at these indices to maximize the total sum.

Return an array answer of size n, where answer[i] represents the result for the corresponding index i.

Input: nums1 = [4,2,1,5,3], nums2 = [10,20,30,40,50], k = 2
Output: [80,30,0,80,50]

Correct

```python
class Solution:
    def findMaxSum(self, nums1: List[int], nums2: List[int], k: int) -> List[int]:
        import heapq

        n = len(nums1)
        answer = [0] * n

        # Create list of (nums1[i], nums2[i], i)
        items = [(nums1[i], nums2[i], i) for i in range(n)]

        # Sort by nums1 value
        items.sort()

        # Process each unique nums1 value
        i = 0
        heap = []  # Min heap to maintain k largest values
        heap_sum = 0

        while i < n:
            j = i
            # Find all indices with same nums1 value
            while j < n and items[j][0] == items[i][0]:
                j += 1

            # For indices i to j-1, the answer is heap_sum
            for idx in range(i, j):
                answer[items[idx][2]] = heap_sum

            # Add values from indices i to j-1 to heap
            for idx in range(i, j):
                val = items[idx][1]
                if len(heap) < k:
                    heapq.heappush(heap, val)
                    heap_sum += val
                elif val > heap[0]:
                    heap_sum = heap_sum - heap[0] + val
                    heapq.heapreplace(heap, val)

            i = j

        return answer
```

Wrong

```python
class Solution:
    def findMaxSum(self, nums1: List[int], nums2: List[int], k: int) -> List[int]:
        n = len(nums1)
        answer = []

        for i in range(n):
            # Find all nums2[j] where nums1[j] < nums1[i]
            candidates = [nums2[j] for j in range(n) if nums1[j] < nums1[i]]

            # Sort in descending order and take top k
            candidates.sort(reverse=True)

            # Sum the top k values (or all if less than k)
            answer.append(sum(candidates[:k]))

        return answer
```

Analysis

The correct solution uses sorting and a heap to process the problem in $O(n \log n)$ time, making it suitable for large datasets. The wrong solution applies a brute-force approach for each index, leading to $O(n^2 \log n)$ complexity and rendering it impractical for the given constraints.

H.3 Hard Example 1

Query

You are given a 0-indexed integer array nums of length n.
We want to group the indices so for each index i in the range [0, n - 1], it is assigned to exactly one group.
A group assignment is valid if the following conditions hold:

For every group g, all indices i assigned to group g have the same value in nums.
For any two groups g_1 and g_2, the difference between the number of indices assigned to g_1 and g_2 should not exceed 1.

Return an integer denoting the minimum number of groups needed to create a valid group assignment.

Input: nums = [3,2,3,2,3]
Output: 2

Correct

```python
class Solution:
    def minGroupsForValidAssignment(self, nums: List[int]) -> int:
        from collections import Counter

        def min_groups_for_frequency(f, s):
            if s == 0:
                return float('inf') if f > 0 else 0

            y_base = f % s
            if y_base * (s + 1) > f:
                return float('inf')

            max_k = (f - y_base * (s + 1)) // (s * (s + 1))
            y = y_base + max_k * s
            x = (f - y * (s + 1)) // s

            return x + y

        def is_feasible(g):
            s = n // g
            total_groups_needed = 0
            for f in frequencies:
                groups_needed = min_groups_for_frequency(f, s)
                if groups_needed == float('inf'):
                    return False
                total_groups_needed += groups_needed

            return total_groups_needed <= g

        counter = Counter(nums)
        frequencies = list(counter.values())
        n = len(nums)

        left, right = 1, n
        while left < right:
            mid = (left + right) // 2
            if is_feasible(mid):
                right = mid
            else:
                left = mid + 1

        return left
```

Wrong

```
class Solution:
    def minGroupsForValidAssignment(self, nums: List[int]) -> int:
        from collections import Counter

        frequencies = list(Counter(nums).values())
        n = len(nums)

        for k in range(1, n + 1):
            small_size = n // k
            large_size = small_size + 1
            small_count = k - n % k
            large_count = n % k

            valid = True
            total_small_used = 0
            total_large_used = 0

            for f in frequencies:
                min_groups = (f + large_size - 1) // large_size
                max_groups = f // small_size

                if min_groups > max_groups:
                    valid = False
                    break

                groups_for_f = min_groups
                large_used = f - groups_for_f * small_size
                small_used = groups_for_f - large_used

                total_small_used += small_used
                total_large_used += large_used

            if valid and total_small_used <= small_count and total_large_used <= large_count:
                return k

        return n
```

Analysis

The incorrect solution is generally effective: for most typical cases, it iterates through possible group counts and checks if the values' frequencies can be distributed among the groups without violating the group size constraints. This approach works well when the frequencies of each value can be easily split into groups of size n/k or n/k+1, and when the frequencies are compatible with the target group counts.
However, the method breaks down in subtle edge cases where the frequency of a particular value cannot be decomposed into the available group sizes, even though a greedy assignment seems possible at first glance. For example, if a frequency cannot be written as a sum of allowed group sizes (due to indivisibility or remainder issues), the solution may still proceed as if it can, leading to an invalid assignment. This is typically overlooked because, in the majority of cases, the distribution works out, and the greedy method produces the correct answer.

I TestGen Pairs

I.1 Easy Example 1

Query

You are given a 0-indexed integer array coins, representing the values of the coins available, and an integer target.
An integer x is obtainable if there exists a subsequence of coins that sums to x.
Return the minimum number of coins of any value that need to be added to the array so that every integer in the range [1, target] is obtainable.
A subsequence of an array is a new non-empty array that is formed from the original array by deleting some (possibly none) of the elements without disturbing the relative positions of the remaining elements.

Notes:
A string a is lexicographically smaller than a string b (of the same length) if in the first position where a and b differ, string a has a letter that appears earlier in the alphabet than the corresponding letter in b.
A substring is a contiguous sequence of characters within a string.

Correct

```
### Test Input
[1]
100000

### Test Output
16
```

Wrong

```
### Test Input
[1, 3]
4

### Test Output
2
```

Analysis

Both the correct and incorrect cases involve small inputs and basic logic that can be verified quickly without complex computation or simulation. In the correct test, the input consists of a single coin with value 1 and a target of 100,000. To ensure every value up to 100,000 can be formed, we must add coins whose values are powers of two—a classic greedy strategy in problems of this type. Since 2**16 exceeds 100,000, it is clear that 16 additional coins are needed.
In the incorrect test, the coins are [1, 3] and the target is 4. With these coins, we can create sums of 1, 3, and 4, but not 2. Therefore, only one coin of value 2 is needed to complete the set, not two coins as claimed in the output. The mistake is easily identified by listing all possible sums, making the error immediately apparent.

I.2 Easy Example 2

Query

You are given an array of integers nums of size 3.
Return the maximum possible number whose binary representation can be formed by concatenating the binary representation of all elements in nums in some order.
Note that the binary representation of any number does not contain leading zeros.

Correct

```
### Test Input
[1,74,67]

### Test Output
25923
```

Wrong

```
### Test Input
[6,38,57]

### Test Output
118426
```

Analysis

The input size is small and it is straightforward to manually try all permutations of the three numbers, convert each concatenated binary string to a decimal, and verify the correct answer. The error in the wrong test is directly apparent by manually checking all possible arrangements.

I.3 Easy Example 3

Query

Given three strings a, b, and c, your task is to find a string that has the minimum length and contains all three strings as substrings. If there are multiple such strings, return the lexicographically smallest one. Return a string denoting the answer to the problem.

Notes:
A string a is lexicographically smaller than a string b (of the same length) if in the first position where a and b differ, string a has a letter that appears earlier in the alphabet than the corresponding letter in b. A substring is a contiguous sequence of characters within a string.

Correct

```
### Test Input
"mrmzito"
"nssds"
"k"

### Test Output
kmrmzitonssds
```

Wrong

```
### Test Input
"jb"
"rgyxglb"
"koa"

### Test Output
jbkoargyxgib
```

Analysis

The input strings are small in length. The correctness can be checked by simple substring search and length comparison. The error in the wrong test is immediately apparent. The answer "jbkoargyxgib" does not contain "rgyxglb" as a substring, nor does it contain "koa" correctly (it's interleaved with other characters). By inspecting, you can see not all input strings are present as contiguous substrings; this breaks the fundamental requirement of the problem.

I.4 Hard Example 1

Query

Given a string s, return the maximum length of a substring such that it contains at most two occurrences of each character.

Example 1:
Input: s = "bcbbbcba"
Output: 4
Explanation: The following substring has a length of 4 and contains at most two occurrences of each character: "bcbbbcba".

Correct

```
### Test Input
"iyrhxoqetwjgtogyloskzozwkcgtehadt"

### Test Output
19
```

Wrong

```
### Test Input
"bbrjudpoojbosurbkjh"

### Test Output
12
```

Analysis

To verify correctness for this problem, one simple way is to iterate all possible substrings and check if each substring has character occurring more than twice. Alternatively, we could simulate a fully correct sliding window algorithm that can handle potential edge cases and keep track of program state. Both answers look reasonable, and only through algorithmic verification can one be certain which is correct.

I.5 Hard Example 2

Query

Given a positive integer n, return the punishment number of n.
The punishment number of n is defined as the sum of the squares of all integers i such that: 1 <= i <= n

The decimal representation of i * i can be partitioned into contiguous substrings such that the sum of the integer values of these substrings equals i.
Input: n = 10
Output: 182
Explanation: There are exactly 3 integers i that satisfy the conditions in the statement:
- 1 since 1 * 1 = 1
- 9 since 9 * 9 = 81 and 81 can be partitioned into 8 + 1.
- 10 since 10 * 10 = 100 and 100 can be partitioned into 10 + 0.

Hence, the punishment number of 10 is 1 + 81 + 100 = 182

Correct

```
### Test Input
661

### Test Output
1204515
```

Wrong

```
### Test Input
948

### Test Output
1710339
```

Analysis

A straightforward approach to verification is to iterate through all i from 1 to n, computing i^2 for each. For every i, the main task is to determine whether the decimal digits of i^2 can be partitioned into contiguous substrings such that the sum of these substrings equals i. This requires considering all possible ways to split the digit sequence, which is inherently a combinatorial problem. Efficiently checking every partition is typically accomplished using recursion or dynamic programming to explore all partition points while avoiding redundant calculations and overcounting. For large n, even a brute-force solution requires careful implementation to systematically enumerate all valid partitions and ensure correctness, as the number of possible partitions grows exponentially with the number of digits in i^2.

I.6 Hard Example 3

Query

You are given an integer array rewardValues of length n, representing the values of rewards. Initially, your total reward x is 0, and all indices are unmarked. You are allowed to perform the following operation any number of times:

Choose an unmarked index i from the range [0, n - 1]. If rewardValues[i] is greater than your current total reward x, then add rewardValues[i] to x (i.e., x = x + rewardValues[i]), and mark the index i.

Return an integer denoting the maximum total reward you can collect by performing the operations optimally.

Input: rewardValues = [1,1,3,3]
Output: 4
Explanation: During the operations, we can choose to mark the indices 0 and 2 in order, and the total reward will be 4, which is the maximum.

Correct

```
### Test Input
[14, 5, 20, 4, 11]

### Test Output
39
```

Wrong

```
### Test Input
[256, 512, 4, 32, 2, 16, 64, 128, 1, 8]

### Test Output
511
```

Analysis

To verify correctness, we can simulate the marking process by selecting unmarked indices whose values are strictly greater than the current total reward. Since each choice changes which indices will be eligible in future steps, the optimal sequence is not immediately obvious. Brute-force enumeration of all possible marking orders is computationally impractical for larger arrays. Instead, a greedy or algorithmic approach such as sorting and iteratively picking the smallest eligible value can help explore promising strategies, but may still require careful handling to avoid missing better solutions. As eligibility depends on the evolving state, only thorough simulation or algorithmic search can reliably guarantee the maximum total reward.