CODE-DITING: Automatic Evaluation of Code Generation without References or Test Cases

Guang Yang[†‡§], Yu Zhou[†*], Xiang Chen[¶], Wei Zheng[§], Xing Hu[‖], Xin Zhou[‡‡], David Lo[‡‡], Taolue Chen[ˣ*]
[†]*College of Computer Science and Technology*, *Nanjing University of Aeronautics and Astronautics*, China
[‡]*Institute of Artificial Intelligence and Educational Big Data*, *Nantong Normal College*, China
[§]*School of Artificial Intelligence and Computer Science*, *Nantong University*, China
[¶]*School of Software*, *Northwestern Polytechnical University*, China
[‖]*School of Software Technology*, *Zhejiang University*, China
[‡‡]*School of Computing and Information Systems*,*Singapore Management University*, Singapore
[ˣ]*School of Computing and Mathematical Sciences*,*Birkbeck, University of London*, UK
Email: novelyg@outlook.com, zhouyu@nuaa.edu.cn, xchencs@ntu.edu.cn, wzheng@nwpu.edu.cn,
xinghu@zju.edu.cn, xinzhou.2020@phdcs.smu.edu.sg, davidlo@smu.edu.sg, t.chen@bbk.ac.uk

***Abstract*—Trustworthy evaluation methods for code snippets play a crucial role in neural code generation. Traditional methods, which either rely on reference solutions or require executable test cases, have inherent limitation in flexibility and scalability. The recent LLM-as-Judge methodology offers a promising alternative by directly evaluating functional consistency between the problem description and the generated code. To systematically understand the landscape of these LLM-as-Judge methods, we conduct a comprehensive empirical study across three diverse datasets. Our investigation reveals the pros and cons of two categories of LLM-as-Judge methods: the methods based on general foundation models can achieve good performance but require complex prompts and lack explainability, while the methods based on reasoning foundation models provide better explainability with simpler prompts but demand substantial computational resources due to their large parameter sizes. To address these limitations, we propose CODE-DITING, a novel code evaluation method that balances accuracy, efficiency and explainability. We develop a data distillation framework that effectively transfers reasoning capabilities from DeepSeek-R1-671B to our CODE-DITING 1.5B and 7B models, significantly enhancing evaluation explainability and reducing the computational cost. With the majority vote strategy in the inference process, CODE-DITING 1.5B outperforms all models with the same magnitude of parameters and achieves performance which would normally exhibit in a model with 5 times of parameter scale. CODE-DITING 7B surpasses GPT-4o and DeepSeek-V3 671B, even though it only uses 1% of the parameter volume of these large models. Further experiments show that CODE-DITING is robust to preference leakage and can serve as a promising alternative for code evaluation.**

***Index Terms*—Code Generation, Evaluation, LLM-as-Judge**

I. INTRODUCTION

Large Language Models (LLMs) have emerged as a fundamental tool in modern software development [1]–[3], demonstrating exceptional language understanding and generation capabilities. Their application has shown remarkable potential across various software engineering tasks [4], [5], particularly in code generation [6]–[8]. However, as LLMs are increasingly deployed, evaluating the correctness of generated code remains a significant challenge [9], [10], primarily because multiple correct or semantically equivalent solutions [11] may exist for a given programming problem.

Traditional evaluation metrics, which are either reference-based or test-based, have been widely adopted. However, these metrics suffer from inherent limitations. Reference-based metrics (e.g., BLEU [12], ROUGE [13] and ChrF [14]) depend on high-quality reference code and frequently penalize implementations that are correct but diverge from them. Test-based metrics (e.g., Pass@k [15]) require careful manual design of comprehensive test cases that cover edge cases, along with secure environments for code execution. Another evaluation method is human evaluation [16], which is accurate yet expensive, as it involves multiple domain experts who directly assess the correctness of generated artifacts. More important, human evaluation is prohibitively labor-intensive and time-consuming, rendering it impractical for large-scale assessments. These constraints significantly limit the flexibility and scalability of human evaluation for code generation evaluation [17], [18].

Recent advancements in LLMs have motivated the development of LLM-as-Judge methods [19]–[21], which directly evaluate the functional consistency between problem descriptions and generated code. These methods offer a promising alternative to traditional evaluation methods [22]. However, with the rapid proliferation of LLM-as-Judge methods, there remains considerable uncertainty regarding their performance in code generation evaluation and it is far from clear which method delivers optimal results [23].

Empirical study. We first conduct a large-scale empirical study to systematically compare different LLM-as-Judge methods in code generation evaluation. Specifically, we classify existing LLM-as-Judge methods into two categories, i.e., methods based on general models (e.g., GPT-3.5-turbo and GPT-4o) and methods based on reasoning-focused models (e.g., DeepSeek-R1 [24]). To ensure the comprehensive evaluation, we curate three datasets (i.e., HumanEval-Judge, MBPP-Judge and BigCodeBench-Judge) as new benchmarks for evaluating

* Yu Zhou and Taolue Chen are the Corresponding authors.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00021

the effectiveness of LLM-as-Judge methods in code generation evaluation. Our findings indicate that, while these methods generally perform well, they exhibit significant discrepancy across various dimensions. In particular, the former requires elaborate prompts and lacks explainability, whereas the latter provides enhanced explainability with simpler prompts but demands substantial computational resources due to their parameter sizes.

Our methods. To address these limitations and advance the state of code generation evaluation, we propose a novel code evaluation method that effectively balances accuracy, efficiency, and explainability. We name it CODE-DITING[1].

To reduce the computational cost, we develop a data distillation framework that transfers reasoning capabilities from the powerful DeepSeek-R1-671B model to our more compact CODE-DITING model, available in 1.5B and 7B parameter sizes. Through this process, we construct a high-quality dataset CODEJUDGE-17K consisting of 17,000 carefully curated samples with reasoning paths. This method not only enhances the explainability of the evaluation but also makes the reasoning process more accessible and comprehensible. To further enhance performance, the CODE-DITING models employ PiSSA [25] technique for model training and the majority vote strategy during inference.

Experimental results demonstrate that CODE-DITING 1.5B outperforms all models of comparable parameter magnitude and achieves performance equivalent to models with five times the parameter count. Notably, CODE-DITING 7B surpasses even large-scale models such as GPT-4o and DeepSeek-V3 671B [26], despite utilizing only 1% of their parameter volume. Our ablation studies reveal that all components of CODE-DITING are essential for its superior performance. In addition, we demonstrate that CODE-DITING is robust to preference leakage [27], where evaluation models show bias toward code produced by same series of architectures, a common issue in LLM-as-Judge methods. These findings establish CODE-DITING as a promising alternative for code generation evaluation, representing a significant advancement in the field.

Summary of contributions.

- We curate three datasets (i.e., HumanEval-Judge, MBPP-Judge and BigCodeBench-Judge) as benchmark for the empirical study. In addition, we introduce a new dataset CODEJUDGE-17K designed for training purposes.
- We design and carry out a large-scale empirical study to systematically compare different LLM-as-Judge methods in code generation evaluation.
- We propose CODE-DITING, a novel code evaluation method that effectively balances accuracy, efficiency and explainability.
- We conduct extensive experiments to evaluate the performance of CODE-DITING on different scenarios, including performance comparisons, ablation studies and analyses of preference leakage.

To facilitate reproducibility, experimental data and model weights are released at https://github.com/Code-DiTing.

[1]The name is from Chinese classic *Journey to the West*, reflecting the model's goal to accurately discern the correctness of code implementations, just as the mythical creature distinguishes truth from falsehood.

TABLE I: Comparison of Code Generation Evaluation Metrics, where Func. means functional correctness, Auto. means automatic evaluation, Expl. means explainability and Open. means using open-source models. ✓ denotes applicable, × denotes not applicable and ○ denotes optional.

Metric	Category		Characteristics			
	Ref	Test	Func.	Auto.	Expl.	Open.
BLEU [12]	✓	×	×	✓	×	✓
Rouge [13]	✓	×	×	✓	×	✓
ChrF [14]	✓	×	×	✓	×	✓
EM [28]	✓	×	×	✓	×	✓
ED [28]	✓	×	×	✓	×	✓
CrystalBLEU [29]	✓	×	×	✓	×	✓
CodeBLEU [30]	✓	×	×	✓	×	✓
CodeBERTScore [31]	✓	×	×	✓	×	✓
CodeScore [32]	✓	×	○	✓	×	✓
CodeScore-R [33]	✓	×	✓	✓	×	✓
Human Study [16]	○	×	✓	×	✓	✓
Pass@k [15]	×	✓	✓	✓	✓	✓
ICE-Score [34]	○	×	✓	✓	×	×
CodeJudge [35]	○	×	✓	✓	×	×
SWE-Judge [36]	✓	×	○	✓	✓	×
CODE-DITING	×	×	✓	✓	✓	✓

II. BACKGROUND AND RELATED WORK

A. Problem Formulation

We formally define the *code generation evaluation problem* as follows. Let $\mathcal{X}$ be the space of problem descriptions, $\mathcal{Y}$ be the space of code implementations, $\mathcal{R}$ be the space of reference implementations and $\mathcal{T}$ be the space of test case sets.

Given a problem description $x \in \mathcal{X}$, a code generation model $M : \mathcal{X} \to \mathcal{Y}$ produces code $y = M(x)$. The evaluation function $\mathcal{F} : \mathcal{X}\times\mathcal{Y}\times\mathcal{R}\times\mathcal{T} \to \{0, 1\}$ determines the functional correctness of y with respect to x. Formally,

$$\mathcal{F}(x, y, r, T) = \begin{cases} 1, & \text{if } y \text{ is functionally correct} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where $r \in \mathcal{R} \cup \{\perp\}$ is an (optional) reference implementation ($r = \perp$ means that r is not provided) and $T \in \mathcal{T} \cup \{\perp\}$ is an (optional) set of test cases ($T = \perp$ means that T is not provided).

Based on the availability of r or T, the existing code generation evaluation methods can be categorized into: reference-based, test-based, and reference-and-Test-free evaluation. Table I summarizes a comparison of code generation evaluation metrics used in various methods.

B. Reference-Based Evaluation ($r \neq \perp$)

Reference-based methods compute the similarity between y and r, based on metrics ranging from token-based metrics (e.g., BLEU [12] and ChrF [14]) to semantics-aware ones (e.g., CodeBLEU [30] and CodeBERTScore [31]).

Token-based metrics are limited to the n-gram lexical similarity computation and ignore potential semantic information in the code. These metrics originate from, e.g., machine translation and text summarization, including BLEU [12], ROUGE [13] and ChrF [14]. Additionally, exact match (EM) metrics are widely used in code synthesis. Eghbali et al. [29] proposed the CrystalBLEU metric to enhance evaluation accuracy by excluding common n-grams that inflate BLEU scores due to verbose syntax and coding conventions. Furthermore, Liguori et al. [28] argued that edit distance (ED) better measures code similarity compared to other token-based metrics.

Semantics-based metrics consider the syntactic structure, data flow information and potential semantic information of code. Ren et al. [30] proposed CodeBLEU, which injects code syntax through AST and code semantics through data flow. Dong et al. [32] proposed CodeScore, which conducts supervised learning on datasets with test cases to perform functional evaluation of code synthesis. Zhou et al. [31] proposed CodeBERTScore, which uses CodeBERT to performs contextual encoding of reference and predicted code to calculate similarity scores between each token. Yang et al. [33] proposed CodeScore-R based on UniXcoder and contrastive learning, which employs sketch processing, syntax transformation and mutation testing to improve the robustness of metric.

Nevertheless, these methods cannot directly assess functional correctness, require high-quality reference code collection, and penalize correct but divergent implementations.

C. Test-Based Evaluation ($T \neq \perp$)

Test-based methods [15] execute code against test cases T to assess functional correctness. The widely-adopted pass@k metric is defined as

$$\text{pass@k} = \mathbb{E}_x \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]$$

where n (resp. c) is the total (resp. correct) number of samples for the problem x. This metric has become standard in evaluating code generation models.

Despite its popularity, pass@k requires human experts for designing high-quality test cases, and demands secure execution environments to prevent malicious code execution.

D. Reference-and-Test-Free Evaluation ($r = \perp$ and $T = \perp$)

When neither reference implementations nor test cases are available, evaluation typically relies on either human evaluation or LLM-as-judge methods [37], [38]. Human evaluation [16], while accurate, is prohibitively expensive and time-consuming for large-scale assessments.

Recent LLM-as-judge methods leverage large language models to directly evaluate the functional consistency between problem descriptions and generated code. Zhuo et al. [34] proposed ICE-Score, which uses GPT-3.5 as a judge to evaluate code generation model performance through carefully crafted prompt engineering. Tong et al. [35] introduced CodeJudge, which not only utilizes GPT-3.5 but also explores smaller open-source models as judges, employing a two-stage prompt engineering method for evaluation. Zhou et al. [36] proposed SWE-Judge, an LLM-as-Ensemble-Judge framework that employs multiple independent LLMs with different evaluation strategies and dynamic team selection to achieve human-aligned assessments. While effective, SWE-Judge incurs significant computational overhead as the ensemble method is adopted, which makes it prohibitively expensive for a comparative analysis.

Recall that current LLM-as-Judge methods incur significant costs through API fees, computational demands or prompt engineering expertise, whereas our method aims for a cost-effective balance of accuracy, efficiency, and explainability.

III. Empirical Study

In this section, we conduct an empirical study to explore the existing LLM-as-judge methods to code generation evaluation and analyze the various factors on their effectiveness.

A. Experiment Setup

Code Generation Datasets. To comprehensively evaluate LLM-as-judge methods, establishing accurate and diverse benchmarks is a crucial first step. We select three diverse and widely adopted datasets that faithfully simulate real-world code generation scenarios. Our dataset selection is guided by two principles: (1) To ensure accurate assessment of semantic correctness, we prioritize datasets with exceptional test case quality and quantity, specifically targeting those with test coverage approaching 100%; (2) Beyond algorithm-centric problems, datasets need to encompass a wide range of libraries and function call patterns typical in professional software development, enabling thorough evaluation of LLM-as-judge methods across varied programming contexts.

As a result, we select the following datasets:

- **HumanEval-plus** [39] is an enhanced variant of the HumanEval benchmark that addresses fundamental ground-truth issues in the original dataset (including unhandled edge cases, logical errors and performance limitations). It expands the test coverage from an average of 9.6 to 764.1 test cases per problem, incorporating more challenging edge cases and complex functionalities to ensure rigorous and comprehensive evaluation.
- **MBPP-plus** [39] applies similar enhancement techniques to the MBPP benchmark, resulting in a test suite 35 times larger than the original dataset.
- **BigCodeBench** [40] specifically targets real-world software development scenarios by incorporating diverse libraries and complex function call patterns. It comprises 1,140 function-level tasks that challenge LLMs to interpret instructions and orchestrate multiple function calls across 139 different libraries. Each programming task is validated through an average of 5.6 carefully designed test cases, achieving a mean branch coverage of 99%.

Data Sampling. With the chosen benchmark datasets, we proceed to sample code generated by various LLMs. We employ different models of varying sizes: Qwen2.5Coder (1.5B/7B) [41] and DeepSeekCoder (1.3B/6.7B) [42] to ensure

TABLE II: Sample Statistics for HumanEval-Judge, MBPP-Judge and BigCodeBench-Judge Datasets

Dataset	Samples	#Positive	#Negative
HumanEval-Judge	640	480 (75.0%)	160 (25.0%)
MBPP-Judge	1,512	997 (65.9%)	515 (34.1%)
BigCodeBench-Judge	800	321 (40.1%)	479 (59.9%)

diversity in the generated solutions. Using multiple models not only enhances the diversity of our dataset but also allows us to evaluate the robustness of LLM-as-Judge methods across different code generation patterns and qualities.

During the data processing phase, we extract natural language problem descriptions and corresponding code implementations from the generated samples through rigorous data cleaning and deduplication processes. Additionally, we remove code comments to enhance conciseness and focus the evaluation on functional implementation rather than documentation.
Data Labeling. (1) Automatic Labeling. We utilize test cases from the existing datasets to automatically label code samples. Functional correctness is determined using the pass@1 metric, serving as the ground-truth for evaluation. (2) Manual Verification. To address potential mislabeling from expanded test cases in HumanEval-plus/MBPP-plus, three authors independently review samples that passed original benchmarks but failed enhanced test suites. Labels are assigned directly when judgments align, or through discussion when opinions differ, ensuring high-quality ground-truth labels.

We hence curate three datasets: HumanEval-Judge (640 samples), MBPP-Judge (1,512 samples) and BigCodeBench-Judge (800 samples). Detailed statistics, including class distributions, are provided in Table II.

B. LLM-as-Judge Methods

Foundation Models. To comprehensively evaluate LLM-as-judge methods across different model scales and architectures, we select a diverse set of foundation models:

- Closed-source models: GPT-3.5-turbo and GPT-4o.
- Large-scale open-source models: DeepSeek-v3-671B and DeepSeek-r1-671B.
- Medium-scale open-source models:Llama3-8B, Qwen2.5-7B and DeepSeek-r1-distill-7B.
- Small-scale open-source models: Llama3-1.5B, Qwen2.5-1.5B and DeepSeek-r1-distill-1.5B.

The DeepSeek series models are classified as *reasoning models* because of their powerful reasoning capabilities, and the rest of the models are classified as *general models*. We limited our study to these models as they provide sufficient representativeness across different architectures, capabilities, and parameter scales.

This selection enables us to analyze how model size affects the performance of LLM-as-judge methods and investigate whether smaller, more computationally efficient models can achieve comparable evaluation quality to their larger counterparts.

Existing Prompting Methods. We evaluate four representative prompting methods that represent different perspectives to eliciting code evaluation capabilities from LLMs:

- **Vanilla**, which is a straightforward prompting method that directly asks the model to evaluate code correctness based on the problem description and implementation, without additional guidance.
- **CoT** [43], which encourages the model to perform step-by-step reasoning by analyzing the code's logic, identifying potential issues, and then making a final judgment on correctness.
- **ICE_SCORE** [34], which performs multi-dimensional evaluation and instructs the LLM to predict an evaluation score from 0 to 4 based on an evaluation criterion. In our experiments, we adopt the evaluation score as 0 or 1 for functional correctness.
- **CodeJudge** [35], which is a two-phase method, where a summary of the given code is first generated and then is evaluated to determine whether the code is correct, based on the generated summary and the given problem description.

C. Evaluation Metrics

To comprehensively assess the performance of LLM-as-judge methods for code evaluation, we employ three metrics.
Accuracy (Acc). It measures the proportion of correctly classified instances among all evaluated samples. For n code samples with ground truth labels y_i and predicted labels $\hat{y}_i$:

$$\text{Accuracy} = \frac{1}{n}\sum_{i=1}^{n}\mathbb{I}(\hat{y}_i = y_i)$$

where $\mathbb{I}(\cdot)$ is the indicator function that returns 1 for correct predictions and 0 otherwise.
F1 Score (F1). It is the macro-average of precision and recall, particularly valuable for our datasets with class imbalance:

$$\text{Precision} = \frac{\text{TP}}{\text{TP}+\text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}$$
$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

F1 ranges from 0 to 1, with higher values indicating better performance in identifying functionally correct code.
Matthews Correlation Coefficient (MCC). It provides a balanced measure by considering all confusion matrix entries:

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP}+\text{FP})(\text{TP}+\text{FN})(\text{TN}+\text{FP})(\text{TN}+\text{FN})}}$$

MCC ranges from -1 to 1, where 1 indicates perfect prediction, 0 random prediction, and -1 inverse prediction. It is less sensitive to class imbalance than the accuracy and F1 score.

D. Implementation Details

Across all experiments, we fix the maximum context length at 8k tokens. Temperature settings were tailored to model type: 0.6 for reasoning-focused models (to promote exploratory

reasoning) and 0.0 for general-purpose models (to ensure deterministic outputs).

We interact with the following large-scale models (via their official APIs): DeepSeek-v3-67B, DeepSeek-r1-67B, GPT-3.5-turbo and GPT-4o. Medium/small-scale open-source models were from Hugging Face, with inference optimized via VLLM [44] on a single RTX 4090 GPU to maximize throughput.

E. Empirical Findings

The results are shown in Table III. Based on the extensive experiments with different models and prompting methods for code evaluation tasks, we have identified differences between general models (GPT/DeepSeek-V3/Llama3/Qwen2.5 series) and reasoning models (DeepSeek-R1 series):

(1) General Models Depend on Prompt Engineering. Our analysis reveals that general-purpose models show high sensitivity to prompt engineering.

Large-scale models respond differently to prompts: GPT-3.5-turbo and DeepSeek-v3 perform best with CodeJudge, while GPT-4o excels with CoT. For medium and small-scale models, structured approaches like ICE_SCORE significantly improve performance. A notable example is Llama3 8B, which achieved an accuracy of 0.658 and MCC of 0.265 using ICE_SCORE, substantially outperforming its Vanilla baseline (accuracy of 0.622 and MCC of 0.194).

Finding 1

For general models, optimal prompting strategies vary by architecture and scale, requiring model-specific customization.

(2) Reasoning Models Prefer Simple Prompts. In contrast to their general counterparts, reasoning models exhibit consistent superior performance with simpler prompts. The Vanilla method emerges as the most effective approach across all DeepSeek-r1-distill model sizes (7B and 1.5B).

Notably, increased prompt complexity often leads to performance degradation, with the 7B model achieving a remarkable 0.737 accuracy using the basic Vanilla approach.

Finding 2

For reasoning models, they have already internalized the reasoning capabilities, requiring no external provision of reasoning steps or structured frameworks.

(3) Performance Comparison. Our evaluation reveals the superior stability of reasoning models across diverse datasets, highlighting their robustness and generalizability.

For large-scale models, DeepSeek-r1-671B with the best prompting method achieves an accuracy of 0.834, F1 score of 0.815 and MCC of 0.632, significantly higher than others. Similarly, for the 7B-scale, DeepSeek-r1-distill 7B with the best prompting method achieves an accuracy of 0.737, F1 score of 0.710, and MCC of 0.443. For the 1.5B-scale models, DeepSeek-r1-distill 1.5B achieves the best accuracy of 0.652, F1 score of 0.604 and MCC of 0.241.

Finding 3

At comparable parameter scales, reasoning models demonstrate superior and more stable performance across different datasets compared to general models.

IV. Methods

In this section, we introduce our method CODE-DITING. Based on the empirical findings in Section III, we build on two key insights: (1) explicit reasoning paths significantly enhance code evaluation accuracy while enabling better sample explainability; and (2) smaller models with appropriate training can potentially match or exceed the performance of much larger models.

CODE-DITING distills reasoning capabilities into compact models to balance accuracy with computational efficiency, as shown in Figure 1.

A. Dataset Construction

To effectively transfer reasoning capabilities from large-scale models to CODE-DITING, high-quality training data are essential.

1) Source Benchmark Collection: We follow three key principles for dataset selection:

- **Diversity:** The dataset needs to cover a wide range of programming scenarios, including algorithmic problems, system programming, and library usage.
- **Difficulty:** In addition to basic syntax tasks, the dataset must encompass complex logical challenges and multi-step reasoning problems.
- **Quality:** The dataset needs to contain high-coverage test cases to ensure reliable functional correctness assessment.

Based on these principles, we select three large-scale code generation benchmarks, i.e, KodCode [45], OpenCoder [46], and CodeHarmony [47], as the seed data. These benchmarks are widely used to train large-scale CodeLLMs.

2) Code Generation and Labeling: To ensure a balanced and representative training set, we implement a systematic data generation and labeling process. In addition, to control the distribution of correct and incorrect examples in our dataset, we generate multiple candidate solutions using Qwen2.5-Coder (1.5B/7B) for each programming task. For quality assurance, we employ a multi-step validation process as HumanEval-Judge:

- We evaluate each solution's functional correctness through test cases and compute the pass@1 as the label.
- We apply static analysis tools to identify and filter out solutions containing syntax errors.
- We remove code comments to focus the evaluation on core implementation logic.

TABLE III: Performance Comparison of Different Models and Prompting Methods across Datasets

Base Model	Prompt	HumanEval-Judge			MBPP-Judge			BigCodeBench-Judge			Avg.		
		Acc	F1	MCC	Acc	F1	MCC	Acc	F1	MCC	Acc	F1	MCC
GPT-3.5-turbo Close-Source	Vanilla	0.730	0.658	0.319	0.663	0.642	0.293	**0.584**	**0.584**	0.219	0.659	**0.628**	0.277
	CoT	**0.781**	0.601	0.303	0.687	0.558	0.214	0.493	0.449	0.210	0.654	0.536	0.242
	ICE_SCORE	0.752	0.571	0.200	0.695	0.575	0.242	0.516	0.487	0.229	0.654	0.544	0.224
	CodeJudge	0.773	**0.666**	**0.343**	**0.726**	**0.661**	**0.349**	0.525	0.498	**0.246**	**0.675**	0.608	**0.313**
GPT-4o Close-Source	Vanilla	0.731	0.679	0.375	0.645	0.633	0.293	0.639	0.614	**0.361**	0.672	0.642	0.343
	CoT	0.873	0.816	0.643	**0.807**	**0.763**	**0.554**	0.706	**0.706**	0.294	**0.795**	**0.762**	0.497
	ICE_SCORE	**0.880**	0.821	0.660	0.785	0.732	0.499	0.666	0.666	0.334	0.777	0.740	**0.498**
	CodeJudge	**0.880**	**0.843**	**0.686**	0.735	0.718	0.446	**0.714**	0.705	0.286	0.776	0.755	0.473
DS-v3 671B	Vanilla	0.873	0.822	0.649	**0.792**	**0.742**	**0.516**	0.626	0.623	0.367	0.764	0.729	0.511
	CoT	0.867	0.788	0.623	0.786	0.722	0.507	0.638	0.632	0.410	0.764	0.714	0.513
	ICE_SCORE	0.814	0.702	0.445	0.737	0.640	0.371	0.564	0.553	0.271	0.705	0.632	0.362
	CodeJudge	**0.884**	**0.831**	**0.675**	0.783	0.730	0.494	**0.675**	**0.674**	**0.325**	**0.781**	**0.745**	**0.498**
DS-r1 671B	Vanilla	**0.925**	**0.904**	**0.812**	**0.828**	**0.806**	**0.613**	**0.748**	**0.735**	**0.471**	**0.834**	**0.815**	**0.632**
	CoT	0.920	0.899	0.802	0.821	0.798	0.597	0.728	0.710	0.423	0.823	0.802	0.607
	ICE_SCORE	**0.925**	**0.904**	0.811	0.825	0.801	0.604	0.744	0.730	0.461	0.831	0.812	0.625
	CodeJudge	0.897	0.875	0.765	0.791	0.773	0.547	0.731	0.708	0.428	0.806	0.785	0.580
Llama3 8B	Vanilla	0.639	0.581	0.184	0.645	0.622	0.252	0.581	0.560	0.147	0.622	0.588	0.194
	CoT	0.667	0.608	0.235	0.682	**0.657**	**0.318**	0.569	**0.569**	**0.178**	0.639	0.611	0.244
	ICE_SCORE	**0.738**	**0.660**	**0.320**	**0.700**	0.656	0.315	0.536	0.533	0.161	**0.658**	**0.616**	**0.265**
	CodeJudge	0.509	0.496	0.121	0.562	0.559	0.184	**0.585**	0.563	0.127	0.552	0.539	0.144
Qwen2.5 7B	Vanilla	0.769	0.662	0.333	**0.748**	**0.694**	**0.408**	0.575	0.563	0.305	0.697	0.640	0.349
	CoT	**0.789**	0.673	0.372	0.743	0.684	0.394	0.563	0.545	0.306	0.698	0.634	0.357
	ICE_SCORE	**0.789**	0.684	0.384	0.745	0.684	0.398	**0.591**	0.582	**0.327**	**0.708**	0.650	**0.370**
	CodeJudge	0.783	**0.694**	**0.391**	0.739	0.693	0.395	0.589	**0.584**	0.287	0.704	**0.657**	0.358
DS-r1-distill 7B	Vanilla	0.816	**0.770**	**0.546**	**0.766**	**0.730**	**0.464**	**0.629**	**0.629**	**0.319**	**0.737**	**0.710**	**0.443**
	CoT	0.773	0.705	0.410	0.765	0.723	0.456	0.524	0.507	0.197	0.687	0.645	0.354
	ICE_SCORE	**0.817**	0.753	0.506	0.747	0.698	0.410	0.571	0.565	0.262	0.712	0.672	0.393
	CodeJudge	0.788	0.720	0.440	0.718	0.676	0.355	0.604	0.604	0.265	0.703	0.667	0.353
Llama3 1B	Vanilla	0.267	0.235	-0.030	0.343	0.277	-0.051	0.551	0.405	-0.092	0.387	0.306	-0.058
	CoT	0.603	0.475	-0.050	0.553	0.490	-0.019	0.540	**0.540**	**0.125**	0.565	**0.502**	0.019
	ICE_SCORE	**0.625**	**0.523**	**0.049**	**0.610**	**0.496**	**0.025**	0.479	0.479	0.000	**0.571**	0.499	**0.025**
	CodeJudge	0.400	0.399	0.000	0.401	0.400	-0.082	**0.561**	0.487	0.013	0.454	0.429	-0.023
Qwen2.5 1.5B	Vanilla	0.630	**0.567**	**0.155**	**0.663**	**0.630**	**0.262**	**0.586**	**0.573**	**0.147**	0.626	**0.590**	**0.188**
	CoT	0.684	0.533	0.075	0.650	0.546	0.133	0.439	0.376	0.076	0.591	0.485	0.095
	ICE_SCORE	0.686	0.480	-0.014	0.651	0.522	0.111	0.446	0.394	0.078	0.594	0.465	0.058
	CodeJudge	**0.717**	0.512	0.069	0.658	0.522	0.123	0.561	0.487	0.095	**0.645**	0.507	0.096
DS-r1-distill 1.5B	Vanilla	0.728	0.639	0.278	**0.714**	**0.664**	**0.336**	0.514	0.510	0.109	**0.652**	**0.604**	**0.241**
	CoT	0.728	0.575	0.172	0.702	0.612	0.275	0.466	0.435	0.087	0.632	0.541	0.178
	ICE_SCORE	**0.747**	**0.658**	**0.317**	0.702	0.643	0.299	0.480	0.467	0.072	0.643	0.589	0.229
	CodeJudge	0.713	0.570	0.152	0.648	0.547	0.131	**0.549**	**0.549**	**0.152**	0.637	0.555	0.145

3) Reasoning Knowledge Distillation: To transfer the logical reasoning capabilities of large-scale reasoning models to our target dataset and enhance sample explainability, we implement a distillation process. For each triple ⟨nl, code, label⟩, we use DeepSeek-R1-671B (the SOTA reasoning model, as shown in Table III) in Vanilla setting to produce independent judgments on code functional correctness, including both predicted labels and reasoning paths. This process yields raw distillation data in the format ⟨nl, code, label, reasoning⟩.

4) Data Filtering and Sampling: We implement a multi-stage filtering mechanism:

- **Accuracy filtering.** We remove samples where DeepSeek-R1-671B's predictions disagreed with test case labels to ensure consistency;
- **Logical coherence filtering.** We employ DeepSeek-V3 as a discriminator[2] to detect and eliminate reasoning paths containing hallucinations or logical inconsistencies;
- **Class balancing.** We downsample the filtered data to achieve a 1:1 ratio between positive and negative samples, addressing the imbalance in the original dataset where correct samples were overrepresented.

As a result, we construct CODEJUDGE-17K, a high-quality dataset containing 17,000 samples. CODEJUDGE-17K features a balanced distribution of correct and incorrect code samples across diverse programming tasks, spanning from basic algorithmic challenges to complex system implementations. Each sample is accompanied by a detailed reasoning path that explains the judgment process, making the dataset valuable for training explainable code judgment models.

[2]The prompt can be found in the our GitHub repository.

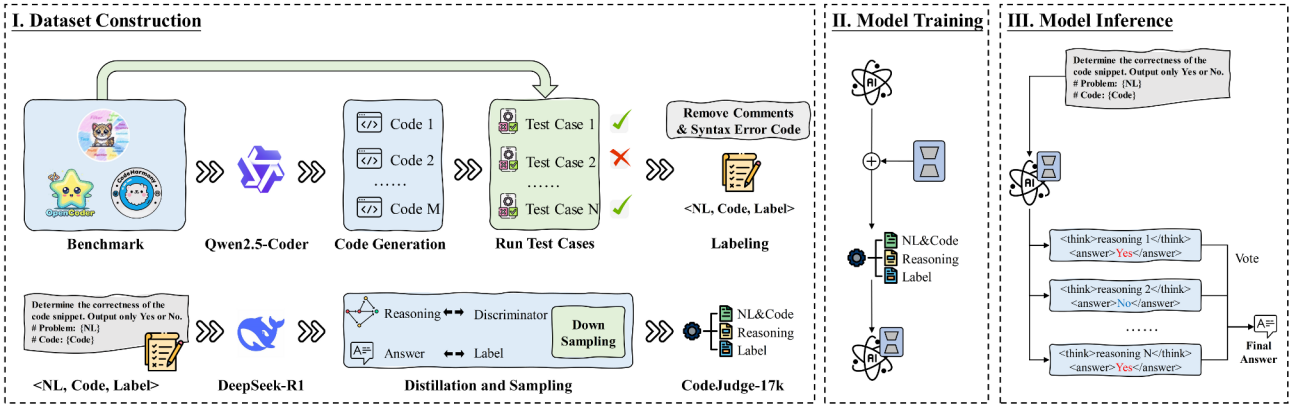


Fig. 1: The overall method of CODE-DITING.

B. Model Training

To transfer reasoning capabilities to smaller models while maintaining efficiency, we train the model in three stages.

1. Knowledge Injection We hypothesize that explicit reasoning paths are crucial for code evaluation tasks. To inject this capability while minimizing deployment costs, we use DS-R1-distil (1.5B/7B) as base models which are fine-tuned on CODEJUDGE-17K. This enables smaller models to learn from larger experts while requiring only 1% of the parameters.

2. Parameter-Efficient Fine-tuning with LoRA To optimize model training while maintaining performance, we adopt Low-Rank Adaptation (LoRA), a parameter-efficient fine-tuning technique. This freezes pre-trained weights $W_0 \in \mathbb{R}^{d\times k}$ and introduces trainable low-rank matrices $A \in \mathbb{R}^{r\times k}$, $B \in \mathbb{R}^{d\times r}$ ($r \ll \min(d,k)$):

$$W = W_0 + BA$$

This reduces trainable parameters from dk to $r(d+k)$, preserving performance with minimal overhead.

3. PiSSA Initialization To enhance training efficiency and model performance, we leverage Principal Singular Vector Adaptation (PiSSA) [25] for initializing LoRA matrices. Instead of Kaiming-uniform [48] initialization used in LoRA, PiSSA leverages the intrinsic low-rank structure of W_0 through truncated SVD, i.e., $W_0 \approx U_r \Sigma_r V_r^\top$. The LoRA matrices are then initialized as

$$B = U_r \Sigma_r^{1/2}, \quad A = \Sigma_r^{1/2} V_r^\top$$

This ensures $\Delta W = BA$ initially aligns with W_0's principal subspace, concentrating updates on directions critical for functional preservation. Compared to the Kaiming-uniform initialization, PiSSA provides structured starting points that improve convergence speed and final performance, particularly in low-rank regimes.

C. Model Inference

Considering that the reasoning model may have inconsistent reasoning paths when the temperature is set to 0.6, we use the Majority Vote strategy to determine the final reasoning result and further enhance model inference performance. This belongs to parallel inference methods, where the model performs multiple independent inferences on the same input, and the most frequent result is selected as the final judgment.

From a probabilistic perspective, if the probability of a correct judgment in a single inference is $P(A)$, the probability of the final result being correct can be modeled through a binomial distribution when conducting T independent inferences. Specifically, if at least $(T+1)/2$ inference results are correct (i.e., the majority vote is correct), then the probability of the final judgment being correct is

$$P\left(X \geq \frac{T+1}{2}\right) = \sum_{k=\lceil \frac{T+1}{2} \rceil}^{T} \binom{T}{k} P(A)^k (1-P(A))^{T-k}.$$

When $P(A) > 0.5$, according to the Law of Large Numbers, as T increases, the success probability of the majority vote strategy $P\left(X \geq \frac{T+1}{2}\right)$ will continuously improve. This explains why majority voting can effectively enhance model performance: as long as the accuracy of a single inference exceeds random guessing (i.e., $P(A) > 0.5$), multiple voting can significantly reduce the probability of misjudgment.

In our experiments, we perform $T = 7$ independent inferences for each test sample and use majority voting to determine the final judgment result. Note that T is set 7 based on RQ3 findings (Section V-C) as the optimal trade-off between model performance and inference latency.

V. EXPERIMENTS AND ANALYSIS

To evaluate the effectiveness and benefits of CODE-DITING, we mainly study the following three research questions (RQs):

A. RQ1: How does CODE-DITING perform compared to the state-of-the-art methods?

To evaluate the performance of CODE-DITING, we compare it with various models mentioned in Section III. For a fair comparison, we use the most effective prompt for each model and employ the same evaluation metrics. The results are presented in Table IV.

TABLE IV: Performance Comparison of Different Models and Prompting Methods across Datasets

Base Model	Prompt	HumanEval-Judge			MBPP-Judge			BigCodeBench-Judge			Avg.		
		Acc	F1	MCC	Acc	F1	MCC	Acc	F1	MCC	Acc	F1	MCC
GPT-3.5-turbo	CodeJudge	0.773	0.666	0.343	0.726	0.661	0.349	0.525	0.498	0.246	0.675	0.608	0.313
GPT-4o	CoT	0.873	0.816	0.643	0.807	0.763	0.554	0.706	0.706	0.294	0.795	0.762	0.497
DS-v3 671B	CodeJudge	0.884	0.831	0.675	0.783	0.730	0.494	0.675	0.674	0.325	0.781	0.745	0.498
DS-r1 671B	Vanilla	0.925	0.904	0.812	0.828	0.806	0.613	0.748	0.735	0.471	0.834	0.815	0.632
Llama3 8B	ICE_SCORE	0.738	0.660	0.320	0.700	0.656	0.315	0.536	0.533	0.161	0.658	0.616	0.265
Qwen2.5 7B	ICE_SCORE	0.789	0.684	0.384	0.745	0.684	0.398	0.591	0.582	0.327	0.708	0.650	0.370
DS-r1-distill 7B	Vanilla	0.816	0.770	0.546	0.766	0.730	0.464	0.629	0.629	0.319	0.737	0.710	0.443
CODE-DITING 7B	Vanilla	**0.883**	**0.847**	**0.695**	**0.806**	**0.782**	**0.564**	**0.729**	**0.717**	**0.435**	**0.806**	**0.782**	**0.565**
Llama3 1B	ICE_SCORE	0.625	0.523	0.049	0.610	0.496	0.025	0.479	0.479	0.000	0.571	0.499	0.025
Qwen2.5 1.5B	Vanilla	0.630	0.567	0.155	0.663	0.630	0.262	0.586	0.573	0.147	0.626	0.590	0.188
DS-r1-distill 1.5B	Vanilla	0.728	0.639	0.278	0.714	0.664	0.336	0.514	0.510	0.109	0.652	0.604	0.241
CODE-DITING 1.5B	Vanilla	**0.842**	**0.799**	**0.601**	**0.778**	**0.755**	**0.510**	**0.681**	**0.653**	**0.318**	**0.767**	**0.736**	**0.476**

(1) Performance Comparison. Both CODE-DITING 1.5B and 7B models significantly outperform other models in their respective parameter scales, with substantial improvements across accuracy, F1 score and MCC metrics. In particular, CODE-DITING 1.5B surpasses Llama3 1B, Qwen2.5 1.5B and even the base DS-r1-distill 1.5B model by large margins. Similarly, CODE-DITING 7B shows clear advantages over Llama3 8B, Qwen2.5 7B, and the base DS-r1-distill 7B model.
(2) Parameter Efficiency. The parameter efficiency of our method is particularly noteworthy, as CODE-DITING 1.5B achieves performance comparable to DS-r1-distill 7B despite using only about 20% of its parameters, demonstrating the effectiveness of our knowledge distillation method in transferring reasoning capabilities to smaller models.

Most impressively, CODE-DITING 7B outperforms both (closed-source) GPT-4o and DeepSeek-V3 (671B) across all three datasets, falling short only of DeepSeek-R1 671B. This is remarkable considering that CODE-DITING 7B uses only about 1% of the parameters of these larger models.

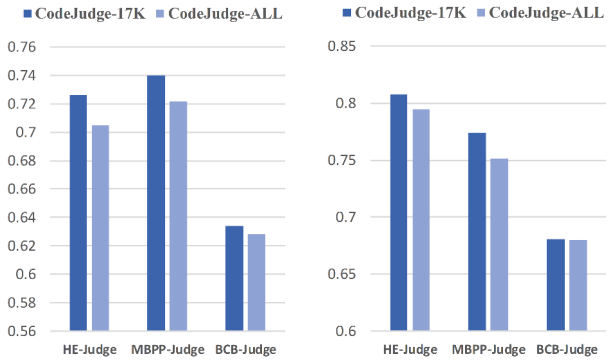
Both CODE-DITING variants maintain strong performance across all evaluation datasets, indicating robust generalization capabilities. These results validate our hypothesis that explicit reasoning paths are crucial for code evaluation tasks and demonstrate that smaller models can effectively learn these reasoning patterns through our proposed fine-tuning method.

Summary of RQ1

CODE-DITING demonstrates superior performance in code evaluation compared to state-of-the-art methods. The 1.5B variant outperforms all models in its parameter class, matching models of 5x larger. The 7B variant surpasses GPT-4o and DeepSeek-V3(671B), using only 1% of their parameters.

B. RQ2: What is the impact of different components of CODE-DITING*?*

To evaluate the effectiveness of different components of CODE-DITING, we conducted a series of ablation studies, focusing on three key aspects: data filtering, parameter initialization and inference strategy.



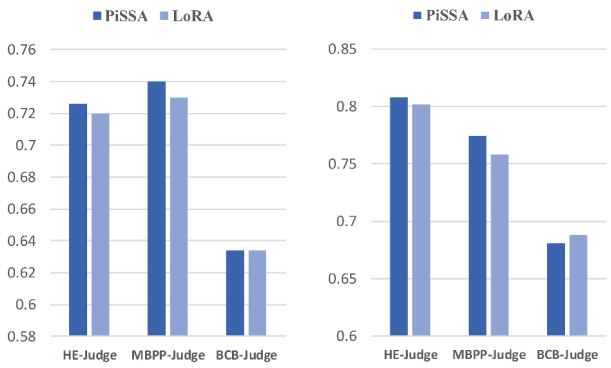
(a) CODE-DITING 1.5B (b) CODE-DITING 7B

Fig. 2: Ablation Study (F1 Score) of Data Filtering Component

(1) Data Filtering Component. Figure 2 illustrates the impact of the data filtering component on model performance. We compare the F1 scores under k=1 (single inference) across different datasets and observe that the data filtering strategy consistently and significantly improves model performance. This empirical evidence strongly supports the hypothesis that high-quality reasoning paths are crucial for models to develop accurate code evaluation capabilities.

Specifically, the relative improvement from filtering is notably more pronounced in the smaller 1.5B model compared to the 7B model. This distinct impact suggests that smaller models, with their inherently limited representational capacity, benefit disproportionately from high-quality training data, as they lack the parameter space to effectively learn from noisy or ambiguous examples.

(2) PiSSA Component. Figure 3 shows the impact of PiSSA initialization on model performance. We also compare F1 scores at k=1 across different initialization methods to isolate this component's contribution. In standard LoRA implementations, the A matrix is typically initialized using Kaiming-



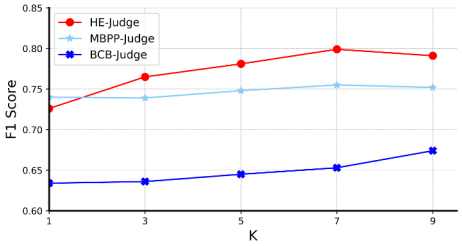
(a) CODE-DITING 1.5B (b) CODE-DITING 7B

Fig. 3: Ablation Study (F1 Score) of PiSSA Component

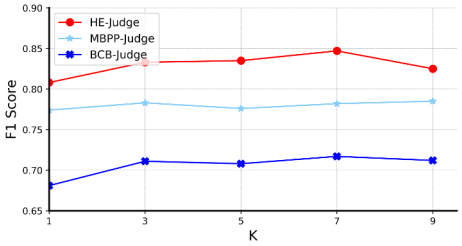
uniform initialization, while the B matrix is initialized to zero. In contrast, PiSSA derives both A and B matrices through SVD decomposition, which fundamentally aligns the initialization with model's intrinsic parameter structure.

The experimental results reveal that PiSSA yields substantial performance improvements on the HumanEval-Judge and MBPP-Judge datasets compared to standard LoRA initialization techniques. However, we observe that the performance enhancement on the more challenging BigCodeBench-Judge dataset is less pronounced, suggesting that initialization benefits may vary with task complexity and dataset characteristics.

These findings indicate that PiSSA initialization helps models converge to more optimal solution spaces, particularly in parameter-constrained low-rank adaptation scenarios.



(a) CODE-DITING 1.5B



(b) CODE-DITING 7B

Fig. 4: Ablation study (F1 Score) of the inference component

(3) Inference Component. Figure 4 presents a detailed analysis of how our inference strategy affects model performance. We systematically compare F1 scores across different values of k (the number of inference passes) to identify the optimal configuration. The results demonstrate a clear pattern: as k increases, model performance consistently improves, though with diminishing returns at higher values.

To determine the most practical configuration for real-world applications, we conduct an analysis of the performance-efficiency trade-off. Our experiments are performed using vllm as the inference server on a single NVIDIA RTX 4090 GPU. The baseline latency (k=1) for a single inference pass is 0.15s and 0.30s for the 1.5B and 7B models, respectively. As expected, the time cost scales linearly with k, reaching approximately 1s (1.5B) and 2s (7B) when k=7.

By analyzing both the performance improvements and computational overhead across different k values, we identify k=7 as the optimal one. This configuration delivers substantial accuracy gains while maintaining reasonable inference latency, making it well-suited for practical applications where both prediction quality and response time are critical considerations.

Summary of RQ2

Our ablation studies demonstrate that each component of CODE-DITING contributes significantly to its overall performance. With the combination of data filtering, PiSSA initialization, and the optimal inference strategy, CODE-DITING achieves state-of-the-art performance while maintaining computational efficiency.

C. *RQ3: Does* CODE-DITING *suffer from preference leakage?*

Preference leakage [27] refers to a contamination issue in LLM-as-judge frameworks where correlations between the synthetic data generator and the LLM-based evaluator lead to biased assessments.

In our training process, we have used code generated by models in the same families (DeepSeek and Qwen Coder) that serve as our base models. This raises a legitimate concern: does CODE-DITING exhibit preference bias toward code generated by models similar to those used in its training data?

To systematically investigate this potential issue, we consider Agreement Rate and Cohen's Kappa [49] as the evaluation metrics. Specifically, Agreement Rate measures the consistency of judgments between different evaluation scenarios:

$$\text{Agreement Rate} = \frac{\text{Number of consistent judgments}}{\text{Total number of samples}}$$

Cohen's Kappa quantifies the agreement between evaluators while accounting for chance agreement:

$$\text{Cohen's Kappa} = \frac{p_o - p_e}{1 - p_e}$$

where p_o is the observed agreement rate and p_e is the expected agreement rate by chance. The chance agreement p_e is calculated based on the marginal distributions of each evaluator's judgments:

$$p_e = \sum_i p_{i1} \times p_{i2}$$

TABLE V: Consistency analysis across different code generation models

Model	Dataset	Agreement Rate	Kappa
GPT-4o	HumanEval-Judge	98.0%	0.96
	MBPP-Judge	96.0%	0.92
	BigCodeBench-Judge	94.0%	0.88
Claude-3.5	HumanEval-Judge	97.0%	0.94
	MBPP-Judge	95.0%	0.90
	BigCodeBench-Judge	93.0%	0.86

TABLE VI: Consistency analysis across different problem descriptions

Model	Dataset	Agreement Rate	Kappa
GPT-4o	HumanEval-Judge	96.0%	0.92
	MBPP-Judge	95.0%	0.90
	BigCodeBench-Judge	94.0%	0.88
Claude-3.5	HumanEval-Judge	95.0%	0.90
	MBPP-Judge	94.5%	0.89
	BigCodeBench-Judge	94.0%	0.87

where p_{i1} and p_{i2} represent the proportion of samples classified as category i by the first and second evaluator, respectively. This adjustment for chance agreement makes Cohen's Kappa a more robust measure than simple agreement rate, especially when the distribution of categories is imbalanced.

We carry out experiments the assess the consistency of CODE-DITING from different perspectives.

(1) Consistency across different code generators. This experiment evaluates whether CODE-DITING maintains consistent judgments when evaluating code generated by different models for the same programming task. We selected 50 problems from each dataset and used two models not involved in our training data generation (i.e., GPT-4o and Claude-3.5) to generate code solutions. We then assessed whether CODE-DITING produced consistent evaluations regardless of the code's source.

As shown in Table V, CODE-DITING demonstrates high consistency in its judgments across different code generators, with agreement rates exceeding 93% across all datasets. The exceptionally high Cohen's Kappa values (ranging from 0.86 to 0.96) indicate near-perfect agreement beyond what would be expected by chance. This consistency is particularly evident on the HumanEval-Judge dataset, where agreement rates reach 98% with GPT-4o-generated code and 97% with Claude-3.5-generated code. Even on the more challenging BigCodeBench-Judge dataset, which involves complex library interactions, CODE-DITING maintains agreement rates of 94% and 93% respectively. These results strongly suggest that CODE-DITING's evaluation mechanism focuses on the intrinsic quality and correctness of code rather than superficial patterns associated with specific code generators.

(2) Consistency across different problem descriptions. This experiment examines whether CODE-DITING maintains consistent judgments when the same code is evaluated against semantically equivalent–but differently phrased–problem descriptions. We also select 50 code samples from each dataset and use GPT-4o and Claude-3.5 to generate paraphrased versions of the original problem descriptions while preserving their semantic meaning. We then evaluate whether CODE-DITING's judgments remained consistent across these different problem formulations.

Table VI shows that CODE-DITING maintains even higher consistency with agreement rates of 94–96% across datasets. The Cohen's Kappa values (0.87-0.92) indicate near-perfect agreement, substantially exceeding what would be expected by chance. Notably, the consistency remains stable across all three datasets, with minimal variation between HumanEval-Judge, MBPP-Judge and BigCodeBench-Judge. The stability is particularly significant for BigCodeBench-Judge, where the complexity of library interactions could potentially make the model more sensitive to variations in problem descriptions. The high agreement rates for both GPT-4o and Claude-3.5 paraphrases demonstrate that CODE-DITING robustly captures the semantic relationship between code and requirements, focusing on functional alignment rather than superficial textual patterns in the problem description. This resilience to paraphrasing suggests that CODE-DITING has developed a deep understanding of programming tasks that transcends specific wording choices.

Summary of RQ3

CODE-DITING does not suffer from significant preference leakage. It maintains high consistency when evaluating code from different generators and when assessing code against semantically equivalent problem descriptions.

VI. CASE STUDY

To better understand CODE-DITING's reasoning capabilities and limitations, we conducted a comprehensive qualitative analysis of both successful and failed cases. Our analysis reveals that CODE-DITING demonstrates strong explainability through step-by-step reasoning paths, as evidenced by correct evaluations where the model systematically analyzes code logic, identifies potential issues, and arrives at sound conclusions.

However, our failure case analysis identifies two primary error categories: **Over-analysis**, where the model correctly identifies the solution initially but continues reasoning and ultimately reaches an incorrect conclusion, and **Ambiguous Problem Descriptions**, where unclear task specifications lead to misjudgments. For instance, in over-analysis cases, CODE-DITING may correctly determine that a simple function works as intended but then introduce unwarranted assumptions about additional requirements not mentioned in the problem statement.

Due to space constraints, detailed case studies with specific examples and comprehensive discussions are available on

our project homepage[3] and failure analysis[4]. These analyses provide valuable insights into the model's reasoning patterns and inform future improvements in reasoning depth control and ambiguity handling.

VII. Threats to Validity

Internal Validity. The primary threat to internal validity concerns implementation fidelity. We mitigated this by carefully implementing baseline methods according to their original descriptions, using public implementations where available, and thoroughly validating our CODE-DITING implementation. Regarding potential bias in the distilled CODEJUDGE-17K training dataset, we employed multi-stage filtering to ensure high-quality reasoning paths and accurate labels.

External Validity. External validity threats stem from our dataset and model selections. We chose HumanEval-plus, MBPP-plus, and BigCodeBench for their high-quality test cases and diverse programming scenarios, though future work could explore additional programming paradigms and domain-specific languages. Our model selection spans various scales and architectures (closed-source GPT models, large-scale DeepSeek models, and smaller open-source models ranging from 1.5B to 8B parameters), providing meaningful insights within our hardware constraints (single RTX 4090 GPU).

Construct Validity. Construct threats concern the performance metrics used to evaluate the performance of CODE-DITING and the compared methods. To evaluate the performance of models, we utilized Accuracy, F1-score, and MCC as the evaluation metrics. Furthermore, to evaluate the performance leakage issue of CODE-DITING, we used Agreement Rate and Cohen's Kappa as the evaluation metrics.

VIII. Conclusion and Future Work

Our work systematically analyzes LLM-as-Judge methods for code generation evaluation, identifying key differences between general and reasoning models. We then proposed a novel code evaluation metric CODE-DITING and demonstrate its superior performance and robustness compared to existing methods.

Future work will focus on: (1) evaluating CODE-DITING across additional programming languages and more complex datasets to better understand its generalization capabilities, and (2) applying CODE-DITING as an environment for reinforcement learning to improve code generation models.

Acknowledgment

This research is supported by the National Natural Science Foundation of China (NSFC, No.62372232) and the OUB Professorial Chair awarded to David Lo by Singapore Management University.

[3]https://github.com/Code-DiTing/Code-DiTing/blob/main/Case_Study.md

[4]https://github.com/Code-DiTing/Code-DiTing/blob/main/Failure_Case_Study.md

References

[1] J. He, C. Treude, and D. Lo, "Llm-based multi-agent systems for software engineering: Literature review, vision and the road ahead," *ACM Transactions on Software Engineering and Methodology*, 2024.

[2] J. Shi, Z. Yang, and D. Lo, "Efficient and green large language models for software engineering: Vision and the road ahead," *ACM Transactions on Software Engineering and Methodology*, 2024.

[3] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[4] L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the capabilities of llms for code change related tasks," *ACM Transactions on Software Engineering and Methodology*, 2024.

[5] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Y. Zhuo, and T. Chen, "Chain-of-thought in neural code generation: From and for lightweight language models," *IEEE Transactions on Software Engineering*, 2024.

[6] Z. Yang, S. Chen, C. Gao, Z. Li, X. Hu, K. Liu, and X. Xia, "An empirical study of retrieval-augmented code generation: Challenges and opportunities," *ACM Transactions on Software Engineering and Methodology*, 2025.

[7] X. Zhang, Y. Zhou, G. Yang, H. C. Gall, and T. Chen, "Anchor attention, small cache: Code generation with large language models," *IEEE Transactions on Software Engineering*, 2025.

[8] X. Zhang, Y. Zhou, G. Yang, W. Cheng, and T. Chen, "Beyond sequences: Two-dimensional representation and dependency encoding for code generation," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025, pp. 6157–6172.

[9] J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 2023, pp. 284–289.

[10] D. Zheng, Y. Wang, E. Shi, R. Zhang, Y. Ma, H. Zhang, and Z. Zheng, "Towards more realistic evaluation of llm-based code generation: an experimental study and beyond," *arXiv preprint arXiv:2406.06918*, 2024.

[11] G. Yang, Y. Zhou, X. Zhang, X. Chen, T. Han, and T. Chen, "Assessing and improving syntactic adversarial robustness of pre-trained models for code translation," *Information and Software Technology*, vol. 181, p. 107699, 2025.

[12] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[13] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[14] M. Popović, "chrf: character n-gram f-score for automatic mt evaluation," in *Proceedings of the tenth workshop on statistical machine translation*, 2015, pp. 392–395.

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

[16] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Han, and T. Chen, "Exploitgen: Template-augmented exploit code generation based on codebert," *Journal of Systems and Software*, vol. 197, p. 111577, 2023.

[17] R. Takaichi, Y. Higo, S. Matsumoto, S. Kusumoto, T. Kurabayashi, H. Kirinuki, and H. Tanno, "Are nlp metrics suitable for evaluating generated code?" in *International Conference on Product-Focused Software Process Improvement*. Springer, 2022, pp. 531–537.

[18] A. Naik, "On the limitations of embedding based methods for measuring functional correctness for code generation," *arXiv preprint arXiv:2405.01580*, 2024.

[19] H. Li, Q. Dong, J. Chen, H. Su, Y. Zhou, Q. Ai, Z. Ye, and Y. Liu, "Llms-as-judges: a comprehensive survey on llm-based evaluation methods," *arXiv preprint arXiv:2412.05579*, 2024.

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

[21] D. Li, B. Jiang, L. Huang, A. Beigi, C. Zhao, Z. Tan, A. Bhattacharjee, Y. Jiang, C. Chen, T. Wu *et al.*, "From generation to judgment: Opportunities and challenges of llm-as-a-judge," *arXiv preprint arXiv:2411.16594*, 2024.

[22] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 595–46 623, 2023.

[23] X. Yu, L. Liu, X. Hu, J. W. Keung, J. Liu, and X. Xia, "Fight fire with fire: How much can we trust chatgpt on source code-related tasks?" *IEEE Transactions on Software Engineering*, 2024.

[24] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[25] F. Meng, Z. Wang, and M. Zhang, "Pissa: Principal singular values and singular vectors adaptation of large language models," *Advances in Neural Information Processing Systems*, vol. 37, pp. 121 038–121 072, 2024.

[26] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[27] D. Li, R. Sun, Y. Huang, M. Zhong, B. Jiang, J. Han, X. Zhang, W. Wang, and H. Liu, "Preference leakage: A contamination problem in llm-as-a-judge," *arXiv preprint arXiv:2502.01534*, 2025.

[28] P. Liguori, C. Improta, R. Natella, B. Cukic, and D. Cotroneo, "Who evaluates the evaluators? on automatic metrics for assessing ai-based offensive code generators," *Expert Systems with Applications*, vol. 225, p. 120073, 2023.

[29] A. Eghbali and M. Pradel, "Crystalbleu: precisely and efficiently measuring the similarity of code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[30] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[31] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "Codebertscore: Evaluating code generation with pretrained models of code," in *The 2023 Conference on Empirical Methods in Natural Language Processing*.

[32] Y. Dong, J. Ding, X. Jiang, G. Li, Z. Li, and Z. Jin, "Codescore: Evaluating code generation by learning code execution," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, pp. 1–22, 2025.

[33] G. Yang, Y. Zhou, X. Chen, and X. Zhang, "Codescore-r: An automated robustness metric for assessing the functionalcorrectness of code synthesis," *arXiv preprint arXiv:2406.06902*, 2024.

[34] T. Y. Zhuo, "Ice-score: Instructing large language models to evaluate code," in *Findings of the Association for Computational Linguistics: EACL 2024*, 2024, pp. 2232–2242.

[35] W. Tong and T. Zhang, "Codejudge: Evaluating code generation with large language models," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 20 032–20 051.

[36] X. Zhou, K. Kim, T. Zhang, M. Weyssow, L. F. Gomes, G. Yang, and D. Lo, "An llm-as-judge metric for bridging the gap with human evaluation in se tasks," 2025. [Online]. Available: https://arxiv.org/abs/2505.20854

[37] D. G. Paul, H. Zhu, and I. Bayley, "Benchmarks and metrics for evaluations of code generation: A critical review," in *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, 2024, pp. 87–94.

[38] J. He, J. Shi, T. Y. Zhuo, C. Treude, J. Sun, Z. Xing, X. Du, and D. Lo, "From code to courtroom: Llms as the new software judges," *arXiv preprint arXiv:2503.02246*, 2025.

[39] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=1qvx610Cu7

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

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

[42] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[43] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[44] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[45] Z. Xu, Y. Liu, Y. Yin, M. Zhou, and R. Poovendran, "Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding," *arXiv preprint arXiv:2503.02951*, 2025.

[46] S. Huang, T. Cheng, J. K. Liu, J. Hao, L. Song, Y. Xu, J. Yang, J. Liu, C. Zhang, L. Chai *et al.*, "Opencoder: The open cookbook for top-tier code large language models," *arXiv preprint arXiv:2411.04905*, 2024.

[47] G. Yang, Y. Zhou, X. Zhang, W. Cheng, K. Liu, X. Chen, T. Y. Zhuo, and T. Chen, "Less is more: Towards green code large language models via unified structural pruning," *arXiv preprint arXiv:2412.15921*, 2024.

[48] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[49] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.