CodeJudge⚖️: Evaluating Code Generation with Large Language Models

Weixi Tong
Huazhong University of Science and Technology
weixitong@hust.edu.cn

Tianyi Zhang
Purdue University
tianyi@purdue.edu

Abstract

Large Language Models (LLMs) have shown promising performance in code generation. However, how to reliably evaluate code generated by LLMs remains an unresolved problem. This paper presents CodeJudge⚖️, a code evaluation framework that leverages LLMs to evaluate the semantic correctness of generated code *without the need for test cases*. We investigate different ways to guide the LLM in performing "slow thinking" to arrive at an in-depth and reliable evaluation. We experimented with four LLMs as evaluators on four code generation datasets and five programming languages. The results show that CodeJudge significantly outperformed existing methods in most settings. Furthermore, compared with a SOTA GPT-3.5-based code evaluation method, CodeJudge achieved better results even when using a much smaller model, Llama-3-8B-Instruct. Our code and datasets are available on GitHub https://github.com/VichyTong/CodeJudge.

1 Introduction

There is an increasing interest in leveraging Large Language Models (LLMs) to generate code (Rozière et al., 2023; Shen et al., 2023). However, reliably evaluating LLMs on code generation tasks remains a challenging problem (Evtikhiev et al., 2023). Test-based methods, such as measuring pass@k (Kulal et al., 2019; Chen et al., 2021), rely on manually written test cases to evaluate code quality. This reliance presents a significant limitation, since many tasks do not come with test cases or only have insufficient test cases that miss corner cases (Liu et al., 2023a). Moreover, it is challenging to write test cases for many coding tasks, such as object serialization and web scraping, since they require extensive effort to construct and configure test stubs and mock objects.

When there are no test cases, existing work often relies on token-based metrics, such as BLEU (Papineni et al., 2002), ROUGE-L (Lin, 2004) and CodeBLEU (Ren et al., 2020), to evaluate model-generated code. However, these metrics do not account for cases where model-generated code is semantically equivalent to the ground truth while having syntactic variations, e.g., using a while loop instead of a for loop, following different naming conventions, etc. In particular, Evtikhiev et al. (2023) shows a statistically significant disagreement on code assessment between human judges and these token-based metrics.

Recent studies show that LLMs are promising alternatives to human evaluators in different tasks (Liu et al., 2023b; Zheng et al., 2023a; Chan et al., 2024). Inspired by these findings, we propose an LLM-based code evaluation framework called CodeJudge⚖️. CodeJudge supports two kinds of assessment: (1) determines whether the model-generated code is correct or not, and (2) estimates to what extent the generated code is aligned with user-intended code. While the former is the typical way of evaluating LLMs in code generation, we argue that the latter provides a more informative evaluation, since LLMs often generate partially correct code, which provides a good starting point or hints to developers (Vaithilingam et al., 2022; Barke et al., 2023). Thus, it is useful to account for partial correctness and the severity of code errors when evaluating LLMs for code generation.

We design two methods to guide the LLM to perform "slow thinking" (Kahneman, 2011) for reliable code evaluation. For the first assessment, CodeJudge instructs the LLM to perform a step-by-step analysis of the code functionality and then asks it to summarize the analysis results into a binary decision. For the second assessment, CodeJudge provides the LLM with a taxonomy of common coding errors and instructs the LLM to identify what types of errors the generated code contains. Then, it computes a code correctness score based on the severity of identified errors. Notably, our framework does not require any test cases or any

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 20032–20051
November 12-16, 2024

fine-tuning of backbone models in code evaluation.

We evaluate CODEJUDGE on five programming languages—Java, C++, Python, JavaScript, Go—and four datasets—HumanEval-X (Zheng et al., 2023b), CoNaLa (Yin et al., 2018; Evtikhiev et al., 2023), APPS (Hendrycks et al., 2021), and BigCodeBench (Zhuo et al., 2024). Following prior work on text generation evaluation (Zhang et al., 2020; Yuan et al., 2021) and code generation evaluation (Zhou et al., 2023; Yuan et al., 2021), we adopt Kendall's τ coefficient and Spearman's ρ to measure the statistical correlation between CODEJUDGE's assessment and the ground truth, which provides a robust measurement for CODEJUDGE's performance. For the first assessment, we also measure the accuracy of the binary decision made by CODEJUDGE as a more intuitive metric for CODEJUDGE's performance.

We experiment with four LLMs as code evaluators and compare CODEJUDGE with nine existing methods, including ICE-Score (Zhuo, 2024), a state-of-the-art code evaluation method based on GPT-3.5-Turbo. For all four LLMs, we observe that CODEJUDGE achieves significantly higher correlations (12.1%-41.8%) than existing methods in most settings. Even when using a relatively small model (Llama-3-8B-Instruct), CODEJUDGE still outperforms ICE-Score, which uses GPT-3.5-Turbo. CODEJUDGE also achieves a high accuracy (80.56% on average) when directly predicting whether a generated code is correct or not. Notably, when the ground-truth code is not available for comparison, CODEJUDGE still achieves reasonable performance (e.g., 0.502 Kendall's τ coefficient[1] and 73.13% accuracy) and outperforms all existing methods that rely on references. This demonstrates that CODEJUDGE can effectively guide LLMs to exert their reasoning capabilities to examine the correctness of code.

2 Background and Related Work

2.1 Problem Formulation

The objective of this work is to evaluate the semantic correctness of machine-generated code. The task of code generation is formulated as generating a *code snippet* $\mathbf{c}$ based on a given *task description* $\mathbf{t}$. We define an evaluation method as a function $f(\mathbf{c}, \mathbf{t})$.

Code evaluation is typically treated as a binary classification task (Chen et al., 2021). The evaluation method f simply determines whether the generated code is correct or not (i.e., $f(\mathbf{c}, \mathbf{t}) \in \{0, 1\}$). For instance, test-based evaluation treats the generated code as correct if it passes all test cases. Otherwise, the code is considered wrong.

Recent studies indicate that code evaluation should not be simply treated as a yes-or-no question (Vaithilingam et al., 2022; Barke et al., 2023). In practice, LLMs often generate code that is not fully correct, e.g., not handling corner cases, missing one computation step, etc. Despite these errors, many developers find the generated code a good starting point compared with writing code from scratch, since they can fix the errors by changing a few lines of code or at least get some inspiration. Thus, it would be helpful if the evaluation method f could measure to what extent the generated code deviates from the user-intended code based on the task description (i.e., $f(\mathbf{c}, \mathbf{t}) \in \mathbb{R}$).

Evaluation without reference code. Many evaluation methods assume that the correct code is available as the ground truth so that they can directly compare the generated code with the ground-truth code. All token-based methods, such as CodeBLEU (Ren et al., 2020) and CodeBERTScore (Zhou et al., 2023), fall into this category. However, this assumption does not always hold in practice, especially in online settings. In many cases, human programmers can make a good assessment only through code inspection and reasoning based on their programming knowledge, without the need for the ground truth. Since LLMs have been demonstrated as promising alternatives to human judges (Liu et al., 2023b; Zheng et al., 2023a; Chan et al., 2024), it is appealing to investigate whether LLMs can make accurate code assessments without reference code. In this work, we consider *code evaluation without references* as a special and more challenging evaluation task.

Challenges. The challenge of code evaluation is two-fold. First, the generated code may have many syntactic variations compared with the correct code while being semantically equivalent, e.g., using different variable names, using a different ordering of independent program statements, using a `for` loop instead of a `while` loop, etc. Second, there can be multiple alternative solutions for a code generation task. For instance, for the task of sorting integers, there are many different sorting algorithms that have drastically different implementations. In

[1]A correlation coefficient above 0.5 is often interpreted as strong correlation (Cohen, 1988).

Task Description: Alphabetize letters in each word of a sentence, keeping the words and spaces in the same order.

```
def anti_shuffle(s):
    return ' '.join([
        ''.join(sorted(list(i)))
        for i in s.split(' ')
    ])
```

(a) Reference code (e.g., ground truth)

```
def anti_shuffle(s):
    return ' '.join([
        sorted(list(i))
        for i in s.split(' ')
    ])
```

(b) Partially correct code

```
def anti_shuffle(s):
    pass
```

(c) Completely useless code

```
def anti_shuffle(s):
    return ' '.join([
        ''.join(sorted(list(word)))
        for word in s.split(' ')
    ])
```

(d) Correct code with syntactic variations

```
def anti_shuffle(s):
    def sort(word):
        return ''.join(sorted(list(word)))
    word_list = []
    current_word = ""
    for i in range(len(s)):
        if s[i] != " ":
            current_word += s[i]
        else:
            word_list.append(sort(current_word))
            current_word = ""
    word_list.append(sort(current_word))
    return ' '.join(word_list)
```

(e) Correct code with a different implementation

Figure 1: An intuitive example of different types of code solving a sentence sorting problem.

the following section, we will illustrate these challenges using existing code evaluation methods on a running example in Figure 1.

2.2 Existing Evaluation Methods

Existing evaluation methods for code generation can be categorized into four categories: *test-based*, *token-based*, *embedding-based*, and more recently, *LLM-based* methods.

Test-based Methods. Pass@k (Kulal et al., 2019) is defined as the percentage of code generation tasks where at least one of the top k code samples generated for a task passes the unit tests of the task. Chen et al. (2021) then introduces an unbiased version of pass@k to reduce variances, which is widely used to evaluate code generation models these days. However, since many tasks lack a comprehensive set of test cases, this often leads to incorrect code snippets incidentally passing given tests. To address this issue, EvalPlus (Liu et al., 2023a) augments the test cases of a given task using LLMs and mutation-based strategies. However, this method still relies on hand-written test cases as the initial seeds.

Token-based Methods. Conventional methods for evaluating machine translation or text generation have been adopted for code evaluation. Basically, these methods compute the token-level similarity between the generated text and the ground-truth text to measure the generation quality. For instance, BLEU (Papineni et al., 2002) calculates modified n-gram precision and includes a brevity penalty. ROUGE-L (Lin, 2004) computes sequence n-grams based on the longest common subsequence. METEOR (Denkowski and Lavie, 2014) relies on the recall and precision of unigrams, while also considering the order of the matched words. ChrF (Popović, 2015) calculates character-level n-gram precision and recall.

CodeBLEU (Ren et al., 2020) and RUBY (Tran et al., 2019) extend traditional token-based methods for code evaluation. CodeBLEU incorporates the similarity of data-flow graphs and abstract syntax trees into the calculation. RUBY calculates similarity based on three representation levels of code: text, AST, and the program dependence graph.

Embedding-based Method. Zhou et al. (2023) proposed CodeBERTScore based on a machine translation evaluation method called BERTScore (Zhang et al., 2020). CodeBERTScore first encodes the generated code and reference code using a fine-tuned CodeBERT (Feng et al., 2020) model. Then, it computes a cosine similarity matrix between the embeddings, based on which CodeBERTScore calculates precision and recall by taking the maximum across rows and columns and averaging the results. CodeBERTScore employs F_1 and F_3 scores to represent the alignment between the generated code and reference code.

LLM-based Method. To the best of our knowledge, ICE-Score (Zhuo, 2024) is the only work that also adopts LLMs for code evaluation. ICE-Score performs multi-dimensional evaluation (Zhong et al., 2022; Liu et al., 2023b; Fu et al., 2023) and instructs the LLM to predict an evaluation score from 0 to 4 based on the definition of an evaluation criterion. Unlike token-based and embedding-based methods, ICE-Score does not require the availability of the reference code. Furthermore,

	Fig. 1(b)	Fig. 1(c)	Fig. 1(d)	Fig. 1(e)
Test-based Methods				
pass@1	0	0	1	1
Token-based Methods				
BLEU	0.779	0.010	0.858	0.231
ROUGE-L	0.914	0.267	0.947	0.431
chrF	0.852	0.266	0.891	0.466
CodeBLEU	0.852	0.052	0.983	0.851
RUBY	0.811	0.364	0.990	0.533
METEOR	0.846	0.164	0.947	0.705
Embedding-based Methods				
CodeBERTScore$_{F_1}$	0.990	0.796	0.976	0.800
CodeBERTScore$_{F_3}$	0.988	0.746	0.976	0.841
LLM-based Methods				
ICE-Score	3.0	0	4.0	3.0
w/o REF	2.0	2.0	3.5	3.0
CODEJUDGE$_{\text{A.S.}}$	0	0	1	1
w/o REF	0	0	1	1
CODEJUDGE$_{\text{F.L.}}$	0.50	0	1.00	1.00
w/o REF	0.50	0	1.00	1.00

Table 1: Scores assigned by various code evaluation methods to the code snippets shown in Figure 1, where Fig. 1(b) is partially correct, Fig. 1(c) is completely useless, Fig. 1(d) is correct but with syntactic variations, and Fig. 1(e) is correct but implemented differently. CODEJUDGE$_{\text{A.S.}}$ and CODEJUDGE$_{\text{F.L.}}$ correspond to the *analyze then summarize* method and *taxonomy-guided fault localization* method described in Section 3.1 and Section 3.2, respectively.

their evaluation shows that including the reference code in the prompt does not significantly improve ICE-Score's performance.

Drawbacks of Existing Methods. Table 1 shows the evaluation scores computed by different methods for the four types of code solutions in Figure 1. We made several interesting observations about the alignment between evaluation scores and the actual correctness of the generated code.

First, token-based and embedding-based methods assign higher scores to partially correct code (Figure 1(b)) compared to correct code with a different implementation (Figure 1(e)). This indicates that these methods face challenges in appropriately scoring code that is correct but syntactically very different.

Second, without the reference code, ICE-Score cannot differentiate between partially correct code (Figure 1(b)) and completely useless code (Figure 1(c)), as it assigns both a score of 2.0. Adding the reference code to ICE-Score addresses this problem but still cannot distinguish between partially correct code (Figure 1(b)) and correct code with a different implementation (Figure 1(e)), assigning both a score of 3.0. These drawbacks may stem from the prompt design of ICE-Score, which simply asks the LLM to predict a score based on the definition of a criterion. In this work, we investigate better ways to exert the inherent reasoning capabilities of LLMs for reliable code evaluation.

3 CODEJUDGE

Our key insight is to guide LLMs to perform "slow thinking" (Kahneman, 2011) in code evaluation, instead of predicting an evaluation score in one step. We design two methods for the two kinds of code evaluation assessment defined in Section 2.1.

3.1 Analyze then Summarize

For the binary evaluation task, we decompose the evaluation task into two subtasks: *analysis* and *summarization*, as illustrated in Figure 2. Specifically, the analysis task provides a step-by-step evaluation guideline and asks the LLM to identify the required functionalities from the task description, examine the logic of the generated code, and report any requirement that is not fulfilled. Optionally, a reference solution can be added to the prompt to aid the analysis. Subsequently, the summarization task asks the LLM to check the analysis report and decide whether the code is correct or not.

This design is inspired by how developers perform code review in practice. Instead of directly arriving at a decision, developers typically do a round of careful analysis of task requirements and code functionality and then decide whether there is any inconsistency. By explicitly asking the LLM to generate a detailed analysis report and double-check it, CODEJUDGE forces the LLM to exert its reasoning capabilities and perform a more careful analysis, instead of making a quick decision.

3.2 Taxonomy-Guided Fault Localization

To decide to what extent a generated code deviates from the user-intended code, we augment the analysis step in the previous section by supplementing the LLM with a taxonomy of common inconsistencies in LLM-generated code and instructing it to identify any potential inconsistencies. As discussed in Section 2.1, different kinds of inconsistencies have different kinds of consequences. Some errors are simple and easy to fix, while others are more severe. Therefore, we incorporate the severity of each identified inconsistency into the summarization step to calculate the correctness score. We explain the details below.

A Taxonomy of Common Inconsistencies. To design the taxonomy, we manually inspected code

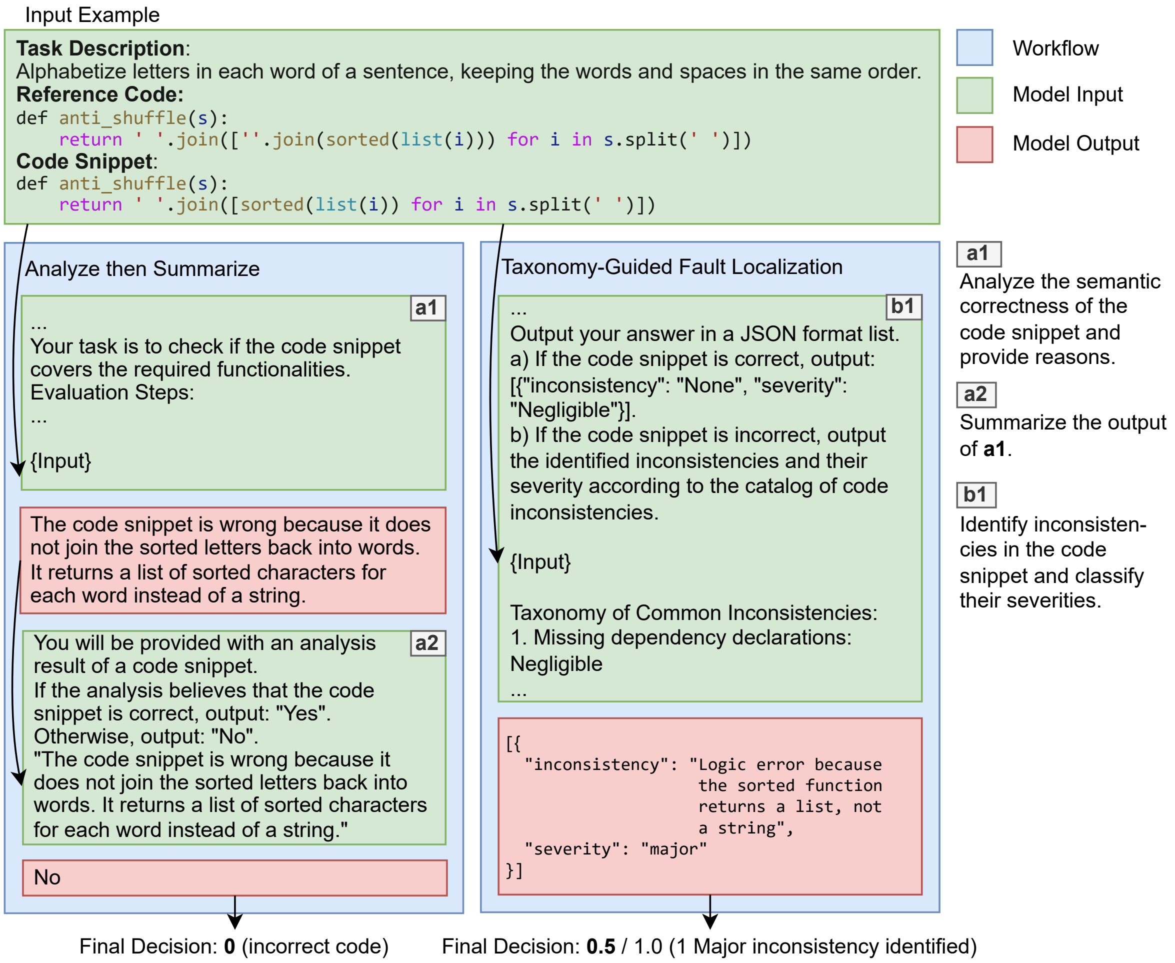

Figure 2: An overview of CODEJUDGE. Full prompts can be found in Appendix F.

snippets generated by different LLMs in different programming languages and also referred to the literature on code error analysis (Hristova et al., 2003; Weimer and Necula, 2004; Chabbi and Mellor-Crummey, 2012; Chen et al., 2018). We summarized eight distinct types of frequent code inconsistencies and categorized them into four severity levels based on their impact on the semantic correctness of the generated code, as shown in Table 2.

- **Negligible.** Code inconsistencies in this category have little impact on semantic correctness. Specifically, we consider missing import statements or exception handling not semantically wrong, since the code generated in such cases indeed achieves the functionality in the task description while not being perfect.
- **Small.** We classify input handling as small due to their limited impact on the core functionality of the code snippet and the straightforward nature of their correction.
- **Major.** Logical errors directly affect the semantic correctness of the code, leading to incorrect outputs. These errors are considered to have a major impact on semantic correctness.
- **Fatal.** Code generation models sometimes hallucinate function calls or variable names that are never defined in the code. Furthermore, in many cases, they generate code with incomplete expressions and statements. These issues often lead to runtime exceptions or compilation errors that crash the program execution. Thus, we considered them as fatal errors.

Given the potential inconsistencies identified by the LLM, we aggregate them via a weighted sum based on their severity levels to compute the final score. To better compare with other methods, we normalize the score to the range of $[0, 1]$. More details can be found in Appendix B.

Type	Description
Negligible	
Alternative	Using different methods or algorithms to solve the problem.
Dependency	Missing import statements.
Error Handling	No exception handling for unexpected events, e.g., invalid inputs.
Efficiency	Including inefficient or unnecessary statements.
Small	
Input Handling	Failing to handle edge cases.
Major	
Logic Error	Containing logical errors.
Fatal	
Declaration	Using undefined functions or variables.
Incompletion	Incomplete code.

Table 2: The catalog of code inconsistencies.

4 Experiments

4.1 Datasets

As described in Section 2.1, CODEJUDGE makes two kinds of code assessment. Following Zhuo (2024), we use HumanEval-X (Zheng et al., 2023b) for the binary assessment task and CoNaLa (Yin et al., 2018) for the code deviation assessment task. The rationale is that HumanEval-X includes test cases for each task so we can easily obtain binary correctness labels based on test results. By contrast, CoNaLa (Yin et al., 2018) does not have test cases. Instead, it provides human-annotated code usefulness scores in the range of 0 to 4, which were obtained via crowdsourcing.

Since HumanEval-X only includes introductory coding tasks, we also include two more challenging datasets, APPS (Hendrycks et al., 2021) and BigCodeBench (Zhuo et al., 2024). Compared with HumanEval-X, APPS includes competition-level coding problems and BigCodeBench includes more complex instructions and more API calls. For instance, Codex achieves a pass@1 rate of 28.81% on HumanEval, but only 0.92% on APPS (Le et al., 2022; Chen et al., 2021). Similarly, GPT-4o achieves a pass@1 rate of 90.2% on HumanEval but only 56.1% on the BigCodeBench (Anthropic, 2024; Zhuo et al., 2024). Since both APPS and BigCodeBench provide only test cases, we use them for the binary assessment task.

We apply our *analyze then summarize* method for binary assessment task datasets (HumanEval-X, APPS, and BigCodeBench) and *Taxonomy-Guided Fault Localization* method for the code deviation assessment task dataset (CoNaLa). We briefly describe each dataset below.

HumanEval-X (Zheng et al., 2023b) is a multi-language version of HumanEval, a popular code generation benchmark originally from the Codex paper (Chen et al., 2021). It contains 164 introductory coding tasks, each of which includes a natural language task description, some test cases, and a human-created reference. We evaluate CODEJUDGE on five programming languages in the initial release of HumanEval-X, including Python, C++, Java, JavaScript, and Go.[2]

CoNaLa (Yin et al., 2018) is a Python code generation benchmark with 472 tasks collected from StackOverflow. We use the human annotation collected by Evtikhiev et al. (2023) as ground truth for the code deviation assessment. For each task, Evtikhiev et al. (2023) asked experienced software developers to grade a score of usefulness between 0 and 4 for the generated code snippets from five different models.

APPS (Hendrycks et al., 2021) is a Python code generation benchmark. It includes introductory-level problems, interview-level, and competition-level coding tasks collected from code competition websites. We randomly sampled 100 competition-level tasks to form a challenging dataset.

BigCodeBench (Zhuo et al., 2024) is a recently released code generation dataset in Python with 1,140 practical and challenging programming tasks. This dataset challenges the ability of LLMs to invoke multiple function calls from various libraries.

4.2 Evaluation Metrics

Statistical Correlations. Recent studies have used statistical correlation metrics, such as Kendall's τ coefficient (τ) and Spearman's rank correlation coefficient (r_s), as a robust way to measure the correlation between code evaluation results and the ground truth (Zhou et al., 2023; Zhuo, 2024). Thus, we adopt these correlation metrics to evaluate CODEJUDGE on both kinds of assessment tasks.

Accuracy. For the binary classification task, we also measure the correctness prediction accuracy of CODEJUDGE as a more intuitive metric.

[2]We tried other languages such as Rust in the latest version of HumanEval-X but encountered issues when running their test cases. Thus, we chose not to evaluate those languages.

Method	HumanEval-X		CoNaLa		APPS		BigCodeBench	
	τ	r_s	τ	r_s	τ	r_s	τ	r_s
EXISTING METHODS								
BLEU	0.306	0.373	0.437	0.485	0.035	0.042	0.072	0.089
ROUGE-L	0.318	0.388	0.450	0.501	0.035	0.043	0.117	0.143
METEOR	0.357	0.436	0.412	0.463	0.085	0.104	0.247	0.302
chrF	0.328	0.400	0.457	0.514	0.036	0.044	0.167	0.205
CodeBLEU	0.362	0.442	0.292	0.332	0.135	0.164	0.173	0.212
RUBY	0.309	0.376	0.332	0.373	0.092	0.113	0.119	0.146
$\text{CodeBERTScore}_{F_1}$	0.339	0.414	0.499	0.558	-0.003	-0.003	0.048	0.059
$\text{CodeBERTScore}_{F_3}$	0.372	0.454	0.485	0.542	0.008	0.010	0.133	0.163
VANILLA	0.570	0.570	0.357	0.386	0.103	0.103	0.251	0.251
VANILLA w/o REF	0.390	0.390	0.465	0.499	-0.058	-0.058	0.131	0.131
ICE-Score	0.475	0.492	0.253	0.271	0.224	0.224	0.321	0.330
ICE-Score w/o REF	0.349	0.363	0.462	0.491	0.140	0.143	0.117	0.118
CODEJUDGE	**0.612**	**0.612**	0.457	0.478	**0.354**	**0.354**	**0.392**	**0.392**
CODEJUDGE w/o REF	0.502	0.502	**0.538**	**0.562**	0.153	0.153	0.317	0.317

Table 3: The results on four datasets when using GPT-3.5-Turbo-1106 as the evaluator. The best results are in **bold**. Due to space limitations, tables with standard deviation and results of each language are shown in Appendix E.

4.3 Comparison Baselines

We employ the six token-based evaluation methods, the embedding-based method, and the recent LLM-based evaluation method described in Section 2.2 as our comparison baselines. We also introduce a vanilla LLM-based method as the baseline. VANILLA directly prompts LLMs to determine the binary semantic correctness of generated code. Table 16 and Table 17 show the prompts used by VANILLA for the two kinds of assessment.

Notably, all token-based and embedding-based methods require the existence of reference code. For LLM-based methods, we evaluate them with and without reference code.

4.4 Experiment Setup

We experiment with four different LLMs as the LLM evaluator in CODEJUDGE, including GPT-3.5, CodeLlama-Instruct (34B), Llama-3-Instruct (8B), and Llama-3-Instruct (70B). For GPT-3.5, we used the `GPT-3.5-Turbo-1106` API. For other models, we run them locally on eight A100-80GB GPUs. Since the cost of prompting GPT-3.5 is high, we only ran the experiments with GPT-3.5 once and set the temperature to 0 and `top_p` to 1 to obtain consistent outputs. For other models, we set the temperature to 0.4 and `top_p` to 0.9. We repeat the experiments three times to account for the randomness in model inference.

Method	HE-X	APPS	BCB
VANILLA	75.96	52.67	65.79
VANILLA w/o REF	65.15	40.33	41.67
ICE-Score	70.91	60.00	66.93
ICE-Score w/o REF	62.47	52.00	46.49
CODEJUDGE	**80.56**	**68.33**	**74.56**
CODEJUDGE w/o REF	73.13	57.00	54.56

Table 4: Average accuracies (%) on HumanEval-X, APPS, and BigCodeBench using GPT-3.5-Turbo.

4.5 Experiment Results

Given the large number of experiments in this evaluation, we first report the results on HumanEval-X and CoNaLa and then report the results on the more challenging APPS and BigCodeBench datasets. Then we report the impact of different factors, including programming languages, LLM evaluators, and prompt design. In the end, we report a failure analysis of 600 cases where CODEJUDGE makes the wrong prediction.

Statistical Correlation with Ground Truth. Table 3 shows the correlations between the code evaluation results of each method and the ground truth on HumanEval and CoNaLa. CODEJUDGE achieves the highest correlations in all settings. For instance, CODEJUDGE achieves 0.612 and 0.562 Spearman's coefficient on HumanEval-X and CoNaLa. Note that a correlation coefficient above 0.5 is often interpreted as a strong correlation (Cohen, 1988).

Surprisingly, providing reference code leads to worse performance for all three LLM-based meth-

Metric	Java		C++		Python		JavaScript		Go	
	τ	r_s	τ	r_s	τ	r_s	τ	r_s	τ	r_s
CODEJUDGE	**0.638**	**0.638**	**0.580**	**0.580**	**0.707**	**0.707**	**0.591**	**0.591**	**0.543**	**0.543**
CODEJUDGE w/o REF	0.508	0.508	0.474	0.474	0.629	0.629	0.453	0.453	0.446	0.446

Table 5: The Kendall-Tau (τ) and Spearman (r_s) correlations between CODEJUDGE using GPT-3.5-Turbo and semantic correctness in HumanEval-X.

Method	CoNaLa		HE-X	APPS	BCB
	τ	r_s	$\tau = r_s$	$\tau = r_s$	$\tau = r_s$
CodeLlama-Instruct-34B					
CODEJUDGE	0.559	0.581	**0.492**	**0.210**	**0.334**
w/o REF	**0.582**	**0.607**	0.412	0.062	0.097
Llama-3-8B-Instruct					
CODEJUDGE	0.523	0.547	**0.480**	**0.161**	**0.383**
w/o REF	**0.576**	**0.602**	0.388	0.072	0.258
Llama-3-70B-Instruct					
CODEJUDGE	0.572	0.598	**0.681**	**0.391**	**0.440**
w/o REF	**0.628**	**0.654**	0.619	0.153	0.298

Table 6: The results of CODEJUDGE using three open-source models (more results in Appendix E).

ods in the CoNaLa dataset. One plausible explanation is that for the code deviation task, the LLM evaluator focuses too much on the differences between the generated code and reference code rather than high-level semantic similarities. This implies future opportunities to calibrate LLMs for code assessment.

Results on More Challenging Benchmarks. Table 3 shows the correlations on APPS and BigCodeBench. While CODEJUDGE still achieves the best performance, we observe that all evaluation methods suffer from a significant drop in performance on APPS and BigCodeBench. The vanilla LLM-based method, which performs comparably to ICE-SCore on the other benchmarks, experienced the biggest degradation. Such a performance drop is not surprising, since these competition-level tasks are challenging to human developers, not even to mention LLMs. Without running and debugging the code, many developers may also struggle with assessing the code. Table 3 shows that LLM-based methods consistently perform better when reference code is provided to aid code evaluation. We also observe that for BigCodeBench, LLM-based methods with reference show a significantly smaller performance degradation compared to methods without reference. This implies that providing reference code is more helpful for challenging tasks compared with relatively simple tasks.

Accuracy of Binary Evaluation. Table 4 shows the accuracy of different methods in the binary assessment task. Since ICE-Score produces a rating in the range of 0 to 4, we treat the rating of 4 as fully correct, while the other ratings as not correct in the binary assessment task. CODEJUDGE outperforms both ICE-Score and VANILLA regardless of whether the reference code is provided or not.

Evaluating without References. We want to highlight that even when reference code is not provided to CODEJUDGE but is provided to all other methods, CODEJUDGE still outperforms all existing methods in most settings. This implies the power of performing "slow thinking" in code evaluation.

Impact of Programming Languages. Table 5 shows the statistical correlation results of CODEJUDGE on different programming languages. When reference code is provided, CODEJUDGE consistently achieves a coefficient above 0.5, which indicates a strong correlation with the ground truth. CODEJUDGE performs much better on Python and Java compared with the other three languages.

Generalizaiblity to Open-Source LLMs. Table 6 shows the correlation results of CODEJUDGE when substituting GPT-3.5 with three open-source models. Compared with GPT-3.5, CODEJUDGE achieves better correlations when using Llama-3-70B. Besides, even when using a relatively small model (Llama-3-8B-Instruct), CODEJUDGE still achieves better or comparable performance to all existing methods, including ICE-Score, which uses GPT-3.5 as the evaluator. This demonstrates that CODEJUDGE can be easily applied to other LLMs and obtain evaluations with a reasonable correlation to semantic correctness.

Prompt Design. We further test CODEJUDGE with few-shot learning, Chain-of-Thought (CoT), and the combination of them. However, CODEJUDGE with these prompting methods do not outperform the original one. Our analysis of the drawbacks of employing CoT and few-shot learning can be found in Appendix A.

Failure Case Analysis. To understand the limitations of CODEJUDGE, we manually inspected

600 failure cases, especially those from APPS. We identified three failure patterns:

- **Wrong Analysis of Code Logic (52.83%).** The most common pattern is that the LLM evaluator fails to infer the code logic correctly. For example, the LLM may mention that the code implements a logic while it does not.
- **Wrong Identification of Task Requirements (26.42%).** For some complex tasks, the LLM evaluator struggles to identify all requirements from the task description correctly.
- **Requirements of Error Handling (20.75%).** We find that the LLM evaluator tends to report many error-handling errors (e.g., not handling invalid inputs) in generated code, even though it is not necessary in many cases. This makes CODEJUDGE over-conservative when evaluating some partially correct code.

5 Conclusion

We propose CODEJUDGE⚖️, a framework that leverages LLMs to evaluate code generation without the need for test cases. We demonstrate that by guiding LLMs to perform slow thinking, CODEJUDGE outperforms all existing code evaluation methods. This demonstrates a promising future direction to replace human evaluators with LLM evaluators. This is also beneficial for alignment methods that rely on human evaluation as feedback. Finally, we release our code and dataset at https://github.com/VichyTong/CodeJudge.

6 Limitations

While we demonstrate that CODEJUDGE achieves state-of-the-art correlation with semantic correctness compared to existing methods, our work does face certain limitations. As we analyzed in Section 4.5, LLMs can generate incorrect judgments or fail to completely follow system prompts when evaluating challenging and complex cases such as the APPS benchmark. However, since CODEJUDGE is an off-the-shelf framework that can easily change the backbone model to powerful LLMs, CODEJUDGE can be continuously improved.

References

AI Anthropic. 2024. Claude 3.5 sonnet model card addendum. *Claude-3.5 Model Card*.

Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111.

Milind Chabbi and John Mellor-Crummey. 2012. Deadspy: a tool to pinpoint program inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 124–134, New York, NY, USA. Association for Computing Machinery.

Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2024. Chateval: Towards better LLM-based evaluators through multi-agent debate. In *The Twelfth International Conference on Learning Representations*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wei Chen, Guoquan Wu, and Jun Wei. 2018. An approach to identifying error patterns for infrastructure as code. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 124–129.

Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates.

Michael Denkowski and Alon Lavie. 2014. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 376–380, Baltimore, Maryland, USA. Association for Computational Linguistics.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the BLEU: How should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. 2023. Gptscore: Evaluate as you desire. *arXiv preprint arXiv:2302.04166*.

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. 2024. GPTScore: Evaluate as you desire. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6556–6576, Mexico City, Mexico. Association for Computational Linguistics.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, page 153–156, New York, NY, USA. Association for Computing Machinery.

Daniel Kahneman. 2011. *Thinking, fast and slow*. macmillan.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. 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*.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023b. G-eval: NLG evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2511–2522, Singapore. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal. Association for Computational Linguistics.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, Yuenan Guo, and Qianxiang Wang. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936*.

N. Tran, H. Tran, S. Nguyen, H. Nguyen, and T. Nguyen. 2019. Does bleu score work for code migration? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 165–176, Los Alamitos, CA, USA. IEEE Computer Society.

Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pages 1–7.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Westley Weimer and George C. Necula. 2004. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN*

Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '04, page 419–431, New York, NY, USA. Association for Computing Machinery.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 476–486, New York, NY, USA. Association for Computing Machinery.

Weizhe Yuan, Graham Neubig, and Pengfei Liu. 2021. Bartscore: Evaluating generated text as text generation. In *Advances in Neural Information Processing Systems*, volume 34, pages 27263–27277. Curran Associates, Inc.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023a. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023b. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, page 5673–5684, New York, NY, USA. Association for Computing Machinery.

Ming Zhong, Yang Liu, Da Yin, Yuning Mao, Yizhu Jiao, Pengfei Liu, Chenguang Zhu, Heng Ji, and Jiawei Han. 2022. Towards a unified multi-dimensional evaluator for text generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2023–2038.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating code generation with pretrained models of code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13921–13937, Singapore. Association for Computational Linguistics.

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, St. Julian's, Malta. Association for Computational Linguistics.

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

A Prompt Design

Method	Acc.
CODEJUDGE	**81.63**
CODEJUDGE w/o REF	74.43
CoT	77.65
CoT w/o REF	68.56
CoT + Few-shot	78.22
CoT + Few-shot w/o REF	67.61
CODEJUDGE + CoT	78.60
CODEJUDGE + CoT w/o REF	72.16
CODEJUDGE + Few-shot	77.84
CODEJUDGE + Few-shot w/o REF	69.89
CODEJUDGE + CoT + Few-shot	77.27
CODEJUDGE + CoT + Few-shot w/o REF	69.51

Table 7: Average accuracy (%) across five programming languages in HumanEval-X using different prompts.

We use Chain-of-Thought (CoT) (Wei et al., 2022) and few-shot learning methods (three examples) to construct different prompts and test them using GPT-3.5-Turbo in HumanEval-X. Table 7 shows the results, helping us understand the effects of CoT and few-shot learning. These two methods slightly reduce the correlation of the results. We observe the following drawbacks of the CoT method and few-shot prompting:

- **Incorrect judgments:** The CoT method can cause the model to make incorrect logical judgments when thinking step by step. If the initial determination is incorrect, the model will generate incorrect evidence to support that determination. For instance, Table 8 shows that when the LLM determines that the code snippet does not cover all the required functionalities in the first sentence, it then makes the incorrect judgment that the model does not account for frequency.

- **Misleading fixed code:** When using the CoT method, the model may ignore the system prompt and create its own process: first, find the errors, then fix the code. This can be misleading during summarization because the model might evaluate the correctness of the fixed code instead of the original, incorrect version. For example, Table 9 shows that

Task Description

```
You are given a non-empty list of positive integers. Return the greatest integer that is greater
than zero, and has a frequency greater than or equal to the value of the integer itself. The
frequency of an integer is the number of times it appears in the list. If no such a value exist,
return -1.
```

Code Snippet

```python
def search(lst):
    # your code here
    d = {}
    for i in lst:
        if i in d:
            d[i] += 1
        else:
            d[i] = 1
    for i in sorted(d, reverse=True):
        if d[i] >= i:
            return i
    return -1
```

Evaluation

```
The code snippet provided does not cover all the required functionalities. It does not correctly
implement the condition for finding the greatest integer that is greater than zero and has a
frequency greater than or equal to the value of the integer itself.

The code snippet iterates through the dictionary keys in reverse order and returns the first key
that satisfies the condition, but it does not consider the frequency of the integer.

The reference solution correctly calculates the frequency of each integer and then iterates through
the frequencies to find the greatest integer that satisfies the condition.
```

Table 8: As the initial determination is that the code "does not cover all the required functionalities", the model provides incorrect evidence—the model does not consider the frequency—in support of the determination.

the LLM generates a conclusion based on the fixed code, and this conclusion further misleads the summarization step.

- **Less example few-shot limitation:** Similar to GPTScore (Fu et al., 2024), we observe a similar performance degradation when the number of few-shot examples is less than four. One possible reason is that LLMs become constrained by the given examples, leading to a preference for particular correctness and reduced generalization ability.

B Postprocessing Steps

For the binary evaluation task, since the LLM generates a free-form response to the summarization task. We use a regex parser that assigns a score of 1 to answers that mention "Yes" and 0 to answers that mention "No". While this postprocessing method may sound simple, it turns out to work very well. In our experiments, we did not observe any cases where the LLMs generated ambiguous answers that cannot be handled by this method.

For the more complex code deviation estimation task, we set the initial correctness score to 100 and deduct a penalty score for each inconsistency identified by CODEJUDGE. We experimented with different penalty score settings on a small validation set, which includes 32 tasks from HumanEval (20%). We found that setting the penalty score of *Small*, *Major*, and *Fatal* inconsistencies to 5, 50, and 100 points achieve the best correlation. We calculate the final score with the following equations.

$$
\begin{aligned}
S &= \text{Num}_{\text{Small}} \times 5 \\
M &= \text{Num}_{\text{Major}} \times 50 \\
F &= \text{Num}_{\text{Fatal}} \times 100 \\
Penalty &= \max(-100, -(S + M + F)) \\
Score &= 1 - \frac{Penalty}{100}
\end{aligned} \tag{1}
$$

C Experiment Details

We use the official version of the HumanEval-X, CoNaLa, APPS, and the BigCodeBench datasets. To generate code snippets for HumanEval-X and APPS, we adopt the code from MultiPL-E.[3] For BigCodeBench, we use the official pre-generated code samples from LLMs.[4] For token-based meth-

[3] https://github.com/nuprl/MultiPL-E

[4] https://github.com/bigcode-project/bigcodebench

Task Description

```
You are given a word. Your task is to find the closest vowel that stands between two consonants from
the right side of the word (case sensitive). Vowels in the beginning and ending doesn't count.
Return empty string if you didn't find any vowel met the above condition. You may assume that the
given string contains English letter only.
```

Code Snippet

```python
def get_closest_vowel(word):
    pass
```

Evaluation

```
The given problem requires finding the closest vowel that stands between two consonants from the
right side of the word. The reference solution iterates through the word from the right side and
checks if the current character is a vowel and the characters on its right and left are consonants.
If this condition is met, the function returns the current vowel. If no such vowel is found, it
returns an empty string.

The provided code snippet is incomplete and does not contain any implementation. To address the
problem statement, the code should iterate through the word from the right side and implement the
logic to find the closest vowel between two consonants.

Here's the complete code snippet:
```

```python
def get_closest_vowel(word):
    if len(word) < 3:
        return ""

    vowels = {"a", "e", "i", "o", "u", "A", "E", 'O', 'U', 'I'}
    for i in range(len(word)-2, 0, -1):
        if word[i] in vowels:
            if (word[i+1] not in vowels) and (word[i-1] not in vowels):
                return word[i]
    return ""
```

```
This code snippet covers the required functionalities as per the problem statement and is
consistent with the reference solution.
```

Table 9: When using the CoT method, the model may design an evaluation plan by itself and generate the fixed code. This fixed code may mislead the summarization step.

Method	GPT-3.5-Turbo-1106	CodeLlama-Instruct-34B	Llama3-Instruct-8B	Llama3-Instruct-70B
CODEJUDGE$_{A.S.}$	2.36	17.42	5.15	7.97
CODEJUDGE$_{A.S.}$ W/O REF	2.73	19.32	6.23	12.23
CODEJUDGE$_{F.L.}$	1.14	14.72	3.04	3.15
CODEJUDGE$_{F.L.}$ W/O REF	1.08	15.18	3.16	3.60

Table 10: Average single execution times (in seconds) over 100 runs.

ods, we adopt implementations from Jetbrains.[5] For CodeBERTScore and ICE-Score, we use their implementations available on GitHub.[6,7]. To evaluate CODEJUDGE, we use the implementations of correlation metrics from https://scipy.org/.

D Latency Discussion

Table 10 shows the average execution times of CODEJUDGE using four different models over 100 runs. The results for GPT-3.5-Turbo-1106 were obtained via the official API. For CodeLlama-Instruct-34B and Llama-3-Instruct-8B, a single A100-80GB GPU was utilized. The execution times of Llama-3-Instruct-70B were measured using two A100-80GB GPUs to load the model. The generating time of CODEJUDGE is less than 20 seconds, which is reasonable for code evaluation compared to manual human annotation.

E Full Results

We report the numbers with standard deviations in the HumanEval-X dataset in Table 11. We also report the accuracy of the binary classification task of the HumanEval-X dataset in Table 12. The full results of the CoNaLa, APPS, and BigCodeBench are in Table 13, Table 14, and Table 15, respectively.

[5] https://github.com/JetBrains-Research/codegen-metrics
[6] https://github.com/neulab/code-bert-score
[7] https://github.com/terryyz/ice-score

F Prompts

We present the full prompts of VANILLA in Tables 16 and 17. Full prompts of CODEJUDGE are shown in Table 18 and 19.

Metric	Java		C++		Python		JavaScript		Go	
	τ	r_s	τ	r_s	τ	r_s	τ	r_s	τ	r_s
EXISTING METHODS										
BLEU	0.230$_{\pm 0.00}$	0.280$_{\pm 0.00}$	0.306$_{\pm 0.00}$	0.373$_{\pm 0.00}$	0.446$_{\pm 0.00}$	0.541$_{\pm 0.00}$	0.288$_{\pm 0.00}$	0.352$_{\pm 0.00}$	0.261$_{\pm 0.00}$	0.318$_{\pm 0.00}$
ROUGE-L	0.249$_{\pm 0.00}$	0.304$_{\pm 0.00}$	0.305$_{\pm 0.00}$	0.372$_{\pm 0.00}$	0.450$_{\pm 0.00}$	0.546$_{\pm 0.00}$	0.329$_{\pm 0.00}$	0.401$_{\pm 0.00}$	0.260$_{\pm 0.00}$	0.317$_{\pm 0.00}$
METEOR	0.299$_{\pm 0.00}$	0.365$_{\pm 0.00}$	0.338$_{\pm 0.00}$	0.412$_{\pm 0.00}$	0.487$_{\pm 0.00}$	0.594$_{\pm 0.00}$	0.379$_{\pm 0.00}$	0.462$_{\pm 0.00}$	0.284$_{\pm 0.00}$	0.346$_{\pm 0.00}$
chrF	0.267$_{\pm 0.00}$	0.326$_{\pm 0.00}$	0.314$_{\pm 0.00}$	0.383$_{\pm 0.00}$	0.448$_{\pm 0.00}$	0.545$_{\pm 0.00}$	0.368$_{\pm 0.00}$	0.449$_{\pm 0.00}$	0.242$_{\pm 0.00}$	0.295$_{\pm 0.00}$
CodeBLEU	0.318$_{\pm 0.00}$	0.388$_{\pm 0.00}$	0.341$_{\pm 0.00}$	0.417$_{\pm 0.00}$	0.501$_{\pm 0.00}$	0.611$_{\pm 0.00}$	0.384$_{\pm 0.00}$	0.468$_{\pm 0.00}$	0.268$_{\pm 0.00}$	0.326$_{\pm 0.00}$
RUBY	0.260$_{\pm 0.00}$	0.318$_{\pm 0.00}$	0.284$_{\pm 0.00}$	0.346$_{\pm 0.00}$	0.425$_{\pm 0.00}$	0.516$_{\pm 0.00}$	0.329$_{\pm 0.00}$	0.401$_{\pm 0.00}$	0.245$_{\pm 0.00}$	0.299$_{\pm 0.00}$
CodeBERTScore$_{F_1}$	0.282$_{\pm 0.00}$	0.344$_{\pm 0.00}$	0.334$_{\pm 0.00}$	0.408$_{\pm 0.00}$	0.453$_{\pm 0.00}$	0.553$_{\pm 0.00}$	0.318$_{\pm 0.00}$	0.388$_{\pm 0.00}$	0.308$_{\pm 0.00}$	0.376$_{\pm 0.00}$
CodeBERTScore$_{F_3}$	0.303$_{\pm 0.00}$	0.370$_{\pm 0.00}$	0.375$_{\pm 0.00}$	0.458$_{\pm 0.00}$	0.495$_{\pm 0.00}$	0.604$_{\pm 0.00}$	0.363$_{\pm 0.00}$	0.443$_{\pm 0.00}$	0.324$_{\pm 0.00}$	0.396$_{\pm 0.00}$
CodeLlama-Instruct-34B										
VANILLA	0.300$_{\pm 0.01}$	0.300$_{\pm 0.01}$	0.345$_{\pm 0.01}$	0.345$_{\pm 0.01}$	0.489$_{\pm 0.03}$	0.489$_{\pm 0.03}$	0.316$_{\pm 0.03}$	0.316$_{\pm 0.03}$	0.314$_{\pm 0.01}$	0.314$_{\pm 0.01}$
VANILLA w/o REF	0.297$_{\pm 0.01}$	0.297$_{\pm 0.01}$	0.373$_{\pm 0.02}$	0.373$_{\pm 0.02}$	0.541$_{\pm 0.03}$	0.541$_{\pm 0.03}$	0.277$_{\pm 0.03}$	0.277$_{\pm 0.03}$	0.348$_{\pm 0.05}$	0.348$_{\pm 0.05}$
ICE-Score	0.418$_{\pm 0.06}$	0.449$_{\pm 0.06}$	0.309$_{\pm 0.04}$	0.331$_{\pm 0.05}$	0.440$_{\pm 0.04}$	0.477$_{\pm 0.04}$	0.308$_{\pm 0.06}$	0.332$_{\pm 0.07}$	0.297$_{\pm 0.06}$	0.320$_{\pm 0.07}$
ICE-Score w/o REF	0.263$_{\pm 0.04}$	0.279$_{\pm 0.04}$	0.282$_{\pm 0.04}$	0.303$_{\pm 0.04}$	0.471$_{\pm 0.05}$	0.503$_{\pm 0.05}$	0.382$_{\pm 0.04}$	0.404$_{\pm 0.04}$	0.338$_{\pm 0.05}$	0.362$_{\pm 0.05}$
CODEJUDGE$_{A.S.}$	**0.515**$_{\pm 0.04}$	**0.515**$_{\pm 0.04}$	**0.464**$_{\pm 0.03}$	**0.464**$_{\pm 0.03}$	**0.625**$_{\pm 0.00}$	**0.625**$_{\pm 0.00}$	**0.503**$_{\pm 0.03}$	**0.503**$_{\pm 0.03}$	0.354$_{\pm 0.02}$	0.354$_{\pm 0.02}$
CODEJUDGE$_{A.S.}$ w/o REF	0.355$_{\pm 0.06}$	0.355$_{\pm 0.06}$	0.408$_{\pm 0.02}$	0.408$_{\pm 0.02}$	0.561$_{\pm 0.02}$	0.561$_{\pm 0.02}$	0.338$_{\pm 0.04}$	0.338$_{\pm 0.04}$	**0.396**$_{\pm 0.02}$	**0.396**$_{\pm 0.02}$
Meta-Llama-3-8B-Instruct										
VANILLA	0.342$_{\pm 0.01}$	0.342$_{\pm 0.01}$	0.216$_{\pm 0.01}$	0.216$_{\pm 0.01}$	0.409$_{\pm 0.02}$	0.409$_{\pm 0.02}$	0.265$_{\pm 0.03}$	0.265$_{\pm 0.03}$	0.192$_{\pm 0.01}$	0.192$_{\pm 0.01}$
VANILLA w/o REF	0.282$_{\pm 0.01}$	0.282$_{\pm 0.01}$	0.159$_{\pm 0.04}$	0.159$_{\pm 0.04}$	0.446$_{\pm 0.02}$	0.446$_{\pm 0.02}$	0.356$_{\pm 0.01}$	0.356$_{\pm 0.01}$	0.331$_{\pm 0.01}$	0.331$_{\pm 0.01}$
ICE-Score	0.389$_{\pm 0.01}$	0.400$_{\pm 0.01}$	0.242$_{\pm 0.01}$	0.248$_{\pm 0.01}$	0.440$_{\pm 0.00}$	0.455$_{\pm 0.00}$	0.296$_{\pm 0.01}$	0.303$_{\pm 0.01}$	0.269$_{\pm 0.00}$	0.281$_{\pm 0.00}$
ICE-Score w/o REF	0.290$_{\pm 0.02}$	0.296$_{\pm 0.02}$	0.306$_{\pm 0.04}$	0.316$_{\pm 0.04}$	0.481$_{\pm 0.03}$	0.499$_{\pm 0.03}$	0.275$_{\pm 0.00}$	0.283$_{\pm 0.00}$	0.287$_{\pm 0.02}$	0.299$_{\pm 0.02}$
CODEJUDGE	**0.523**$_{\pm 0.01}$	**0.523**$_{\pm 0.01}$	**0.387**$_{\pm 0.02}$	0.387$_{\pm 0.02}$	**0.637**$_{\pm 0.04}$	**0.637**$_{\pm 0.04}$	**0.446**$_{\pm 0.03}$	0.446$_{\pm 0.03}$	**0.407**$_{\pm 0.03}$	**0.407**$_{\pm 0.03}$
CODEJUDGE w/o REF	0.411$_{\pm 0.06}$	0.411$_{\pm 0.06}$	0.309$_{\pm 0.04}$	0.309$_{\pm 0.04}$	0.586$_{\pm 0.03}$	0.586$_{\pm 0.03}$	0.339$_{\pm 0.06}$	0.339$_{\pm 0.06}$	0.295$_{\pm 0.01}$	0.295$_{\pm 0.01}$
Meta-Llama-3-70B-Instruct										
VANILLA	0.607$_{\pm 0.01}$	0.607$_{\pm 0.01}$	0.624$_{\pm 0.01}$	0.624$_{\pm 0.01}$	0.685$_{\pm 0.00}$	0.685$_{\pm 0.00}$	0.554$_{\pm 0.00}$	0.554$_{\pm 0.00}$	0.529$_{\pm 0.00}$	0.529$_{\pm 0.00}$
VANILLA w/o REF	0.554$_{\pm 0.01}$	0.554$_{\pm 0.01}$	0.541$_{\pm 0.01}$	0.541$_{\pm 0.01}$	0.651$_{\pm 0.01}$	0.651$_{\pm 0.01}$	0.553$_{\pm 0.01}$	0.553$_{\pm 0.01}$	0.571$_{\pm 0.01}$	0.571$_{\pm 0.01}$
ICE-Score	0.552$_{\pm 0.00}$	0.576$_{\pm 0.00}$	0.516$_{\pm 0.01}$	0.543$_{\pm 0.01}$	0.626$_{\pm 0.01}$	0.654$_{\pm 0.01}$	0.471$_{\pm 0.00}$	0.490$_{\pm 0.00}$	0.389$_{\pm 0.01}$	0.411$_{\pm 0.01}$
ICE-Score w/o REF	0.509$_{\pm 0.01}$	0.531$_{\pm 0.00}$	0.507$_{\pm 0.00}$	0.533$_{\pm 0.00}$	0.591$_{\pm 0.00}$	0.620$_{\pm 0.00}$	0.425$_{\pm 0.00}$	0.444$_{\pm 0.00}$	0.478$_{\pm 0.00}$	0.508$_{\pm 0.00}$
CODEJUDGE	**0.640**$_{\pm 0.02}$	**0.640**$_{\pm 0.02}$	**0.700**$_{\pm 0.03}$	**0.700**$_{\pm 0.03}$	**0.803**$_{\pm 0.02}$	**0.803**$_{\pm 0.02}$	**0.675**$_{\pm 0.01}$	**0.675**$_{\pm 0.01}$	**0.589**$_{\pm 0.02}$	**0.589**$_{\pm 0.02}$
CODEJUDGE w/o REF	0.583$_{\pm 0.02}$	0.583$_{\pm 0.02}$	0.611$_{\pm 0.01}$	0.611$_{\pm 0.01}$	0.698$_{\pm 0.02}$	0.698$_{\pm 0.02}$	0.617$_{\pm 0.04}$	0.617$_{\pm 0.04}$	0.587$_{\pm 0.05}$	0.587$_{\pm 0.05}$
GPT-3.5-Turbo-1106										
VANILLA	0.615	0.615	0.482	0.482	0.675	0.675	0.550	0.550	0.528	0.528
VANILLA w/o REF	0.343	0.343	0.328	0.328	0.537	0.537	0.345	0.345	0.398	0.398
ICE-Score	0.499	0.510	0.436	0.455	0.514	0.537	0.524	0.542	0.402	0.415
ICE-Score w/o REF	0.275	0.278	0.410	0.429	0.485	0.513	0.253	0.258	0.324	0.337
CODEJUDGE	**0.638**	**0.638**	**0.580**	**0.580**	**0.707**	**0.707**	**0.591**	**0.591**	**0.543**	**0.543**
CODEJUDGE w/o REF	0.508	0.508	0.474	0.474	0.629	0.629	0.453	0.453	0.446	0.446

Table 11: The Kendall-Tau (τ) and Spearman (r_s) correlations of each method with semantic correctness on HumanEval-X in multiple languages. "w/ REF" indicates that this method contains the reference code in the prompt. The correlation coefficients are reported across three runs using open-source models, along with the standard deviation.

Method	Java	C++	Python	JavaScript	Go
CodeLlama-Instruct-34B					
VANILLA	$57.07_{\pm 0.01}$	$61.11_{\pm 0.01}$	$72.22_{\pm 0.01}$	$58.33_{\pm 0.01}$	$62.37_{\pm 0.00}$
VANILLA w/o REF	$59.09_{\pm 0.00}$	$65.91_{\pm 0.01}$	$73.48_{\pm 0.02}$	$58.84_{\pm 0.02}$	$57.32_{\pm 0.02}$
CODEJUDGE	$75.00_{\pm 0.02}$	$75.25_{\pm 0.01}$	$80.56_{\pm 0.00}$	$73.74_{\pm 0.01}$	$75.51_{\pm 0.01}$
CODEJUDGE w/o REF	$67.93_{\pm 0.03}$	$73.48_{\pm 0.01}$	$78.03_{\pm 0.01}$	$66.16_{\pm 0.02}$	$71.97_{\pm 0.01}$
Meta-Llama-3-8B-Instruct					
VANILLA	$57.83_{\pm 0.00}$	$47.47_{\pm 0.01}$	$67.42_{\pm 0.01}$	$55.05_{\pm 0.01}$	$47.73_{\pm 0.01}$
VANILLA w/o REF	$58.84_{\pm 0.01}$	$47.47_{\pm 0.02}$	$70.20_{\pm 0.01}$	$62.12_{\pm 0.01}$	$60.10_{\pm 0.00}$
CODEJUDGE	$74.49_{\pm 0.01}$	$65.91_{\pm 0.01}$	$81.57_{\pm 0.02}$	$69.44_{\pm 0.02}$	$69.70_{\pm 0.02}$
CODEJUDGE w/o REF	$70.20_{\pm 0.03}$	$66.16_{\pm 0.02}$	$78.79_{\pm 0.01}$	$65.15_{\pm 0.02}$	$66.16_{\pm 0.01}$
Meta-Llama-3-70B-Instruct					
VANILLA	$78.28_{\pm 0.00}$	$79.29_{\pm 0.00}$	$83.33_{\pm 0.00}$	$74.24_{\pm 0.00}$	$73.48_{\pm 0.00}$
VANILLA w/o REF	$75.51_{\pm 0.00}$	$75.51_{\pm 0.00}$	$82.07_{\pm 0.01}$	$75.76_{\pm 0.01}$	$78.03_{\pm 0.01}$
CODEJUDGE	$81.31_{\pm 0.01}$	$84.60_{\pm 0.02}$	$90.15_{\pm 0.01}$	$81.82_{\pm 0.01}$	$80.30_{\pm 0.01}$
CODEJUDGE w/o REF	$79.55_{\pm 0.01}$	$81.82_{\pm 0.01}$	$84.60_{\pm 0.01}$	$80.56_{\pm 0.02}$	$81.82_{\pm 0.02}$
GPT-3.5-Turbo-1106					
VANILLA	77.27	71.21	82.07	72.98	76.26
VANILLA w/o REF	60.86	67.17	74.24	61.36	62.12
CODEJUDGE	81.57	78.28	85.35	78.28	79.29
CODEJUDGE w/o REF	73.48	72.22	80.81	68.43	70.71

Table 12: Accuracies (%) across five programming languages in the binary classification task of HumanEval-X dataset. The accuracies are reported across three runs using open-source models, along with the standard deviation.

Method	τ	r_s
BLEU	0.437	0.485
ROUGE-L	0.450	0.501
METEOR	0.412	0.463
chrF	0.457	0.514
CodeBLEU	0.292	0.332
RUBY	0.332	0.373
CodeBERTScore$_{f1}$	0.499	0.558
CodeBERTScore$_{f3}$	0.485	0.542
Code Llama - Instruct 34B		
VANILLA	0.317	0.344
VANILLA w/o REF	0.448	0.486
ICE-Score	0.397	0.425
ICE-Score w/o REF	0.534	0.572
CODEJUDGE	0.559	0.581
CODEJUDGE w/o REF	**0.582**	**0.607**
Meta-Llama-3-8B-Instruct		
VANILLA	0.524	0.560
VANILLA w/o REF	0.555	0.592
ICE-Score	0.481	0.513
ICE-Score w/o REF	0.482	0.512
CODEJUDGE	0.523	0.547
CODEJUDGE w/o REF	**0.576**	**0.602**
Meta-Llama-3-70B-Instruct		
VANILLA	0.580	0.611
VANILLA w/o REF	0.583	0.624
ICE-Score	0.481	0.515
ICE-Score w/o REF	0.603	0.641
CODEJUDGE	0.572	0.598
CODEJUDGE w/o REF	**0.628**	**0.654**
GPT-3.5-Turbo-1106		
VANILLA	0.357	0.386
VANILLA w/o REF	0.465	0.499
ICE-Score	0.253	0.271
ICE-Score w/o REF	0.462	0.491
CODEJUDGE	0.457	0.478
CODEJUDGE w/o REF	**0.538**	**0.562**

Table 13: The Kendall-Tau(τ), Pearson(r_p), and Spearman(r_s) correlations of each method with the correctness on the CoNaLa dataset. w/ REF indicates that this method contains the reference code in the prompt.

Method	τ	r_s
EXISTING METHODS		
BLEU	$0.035_{\pm 0.00}$	$0.042_{\pm 0.00}$
ROUGE-L	$0.035_{\pm 0.00}$	$0.043_{\pm 0.00}$
METEOR	$0.085_{\pm 0.00}$	$0.104_{\pm 0.00}$
chrF	$0.036_{\pm 0.00}$	$0.044_{\pm 0.00}$
CodeBLEU	$0.135_{\pm 0.00}$	$0.164_{\pm 0.00}$
RUBY	$0.092_{\pm 0.00}$	$0.113_{\pm 0.00}$
CodeBERTScore$_{\mathrm{F}_1}$	$-0.003_{\pm 0.00}$	$-0.003_{\pm 0.00}$
CodeBERTScore$_{\mathrm{F}_3}$	$0.008_{\pm 0.00}$	$0.010_{\pm 0.00}$
CodeLlama-Instruct-34B		
VANILLA	$0.005_{\pm 0.05}$	$0.005_{\pm 0.05}$
VANILLA w/o REF	$0.080_{\pm 0.00}$	$0.080_{\pm 0.00}$
ICE-Score	$0.174_{\pm 0.06}$	$0.185_{\pm 0.06}$
ICE-Score w/o REF	$-0.032_{\pm 0.02}$	$-0.034_{\pm 0.02}$
CODEJUDGE	$\mathbf{0.210}_{\pm 0.09}$	$\mathbf{0.210}_{\pm 0.09}$
CODEJUDGE w/o REF	$0.062_{\pm 0.04}$	$0.062_{\pm 0.04}$
Meta-Llama-3-8B-Instruct		
VANILLA	$0.123_{\pm 0.01}$	$0.123_{\pm 0.01}$
VANILLA w/o REF	$\mathbf{0.168}_{\pm 0.01}$	$\mathbf{0.168}_{\pm 0.01}$
ICE-Score	$0.003_{\pm 0.03}$	$0.003_{\pm 0.03}$
ICE-Score w/o REF	$0.090_{\pm 0.08}$	$0.091_{\pm 0.08}$
CODEJUDGE	$0.161_{\pm 0.07}$	$0.161_{\pm 0.07}$
CODEJUDGE w/o REF	$0.072_{\pm 0.10}$	$0.072_{\pm 0.10}$
Meta-Llama-3-70B-Instruct		
VANILLA	$0.334_{\pm 0.01}$	$0.334_{\pm 0.01}$
VANILLA w/o REF	$0.279_{\pm 0.04}$	$0.279_{\pm 0.04}$
ICE-Score	$0.297_{\pm 0.07}$	$0.307_{\pm 0.07}$
ICE-Score w/o REF	$0.251_{\pm 0.01}$	$0.256_{\pm 0.01}$
CODEJUDGE	$\mathbf{0.391}_{\pm 0.02}$	$\mathbf{0.391}_{\pm 0.02}$
CODEJUDGE w/o REF	$0.359_{\pm 0.04}$	$0.359_{\pm 0.04}$
GPT-3.5-Turbo-1106		
VANILLA	0.103	0.103
VANILLA w/o REF	-0.058	-0.058
ICE-Score	0.224	0.224
ICE-Score w/o REF	0.140	0.143
CODEJUDGE	**0.354**	**0.354**
CODEJUDGE w/o REF	0.153	0.153

Table 14: The Kendall-Tau (τ) and Spearman (r_s) correlations of each method with semantic correctness on APPS. "w/ REF" indicates that this method contains the reference code in the prompt. The correlation coefficients are reported across three runs using open-source models, along with the standard deviation.

Method	τ	r_s
EXISTING METHODS		
BLEU	$0.072_{\pm 0.00}$	$0.089_{\pm 0.00}$
ROUGE-L	$0.117_{\pm 0.00}$	$0.143_{\pm 0.00}$
METEOR	$0.247_{\pm 0.00}$	$0.302_{\pm 0.00}$
chrF	$0.167_{\pm 0.00}$	$0.205_{\pm 0.00}$
CodeBLEU	$0.173_{\pm 0.00}$	$0.212_{\pm 0.00}$
RUBY	$0.119_{\pm 0.00}$	$0.146_{\pm 0.00}$
CodeBERTScore$_{\mathrm{F_1}}$	$0.048_{\pm 0.00}$	$0.059_{\pm 0.00}$
CodeBERTScore$_{\mathrm{F_3}}$	$0.133_{\pm 0.00}$	$0.163_{\pm 0.00}$
CodeLlama-Instruct-34B		
VANILLA	$0.104_{\pm 0.02}$	$0.104_{\pm 0.02}$
VANILLA w/o REF	$0.047_{\pm 0.02}$	$0.047_{\pm 0.02}$
ICE-Score	$-0.023_{\pm 0.01}$	$-0.023_{\pm 0.01}$
ICE-Score w/o REF	$0.025_{\pm 0.02}$	$0.025_{\pm 0.02}$
CODEJUDGE$_{\mathrm{A.S.}}$	$\mathbf{0.334}_{\pm 0.03}$	$\mathbf{0.334}_{\pm 0.03}$
CODEJUDGE$_{\mathrm{A.S.}}$ w/o REF	$0.097_{\pm 0.02}$	$0.097_{\pm 0.02}$
Meta-Llama-3-8B-Instruct		
VANILLA	$0.070_{\pm 0.01}$	$0.070_{\pm 0.01}$
VANILLA w/o REF	$0.064_{\pm 0.00}$	$0.064_{\pm 0.00}$
ICE-Score	$0.107_{\pm 0.02}$	$0.108_{\pm 0.02}$
ICE-Score w/o REF	$0.007_{\pm 0.02}$	$0.007_{\pm 0.02}$
CODEJUDGE	$\mathbf{0.383}_{\pm 0.01}$	$\mathbf{0.383}_{\pm 0.01}$
CODEJUDGE w/o REF	$0.258_{\pm 0.02}$	$0.258_{\pm 0.02}$
Meta-Llama-3-70B-Instruct		
VANILLA	$0.316_{\pm 0.00}$	$0.316_{\pm 0.00}$
VANILLA w/o REF	$0.225_{\pm 0.00}$	$0.225_{\pm 0.00}$
ICE-Score	$0.297_{\pm 0.00}$	$0.307_{\pm 0.00}$
ICE-Score w/o REF	$0.164_{\pm 0.00}$	$0.166_{\pm 0.00}$
CODEJUDGE	$\mathbf{0.440}_{\pm 0.01}$	$\mathbf{0.440}_{\pm 0.01}$
CODEJUDGE w/o REF	$0.298_{\pm 0.01}$	$0.298_{\pm 0.01}$
GPT-3.5-Turbo-1106		
VANILLA	0.251	0.251
VANILLA w/o REF	0.131	0.131
ICE-Score	0.321	0.330
ICE-Score w/o REF	0.117	0.118
CODEJUDGE	**0.392**	**0.392**
CODEJUDGE w/o REF	0.317	0.317

Table 15: The Kendall-Tau (τ) and Spearman (r_s) correlations of each method with semantic correctness on BigCodeBench. "w/ REF" indicates that this method contains the reference code in the prompt. The correlation coefficients are reported across three runs using open-source models, along with the standard deviation.

```
Determine the correctness of the code snippet. Output Yes or No.

Problem Statement: {PROBLEM}

Code Snippet: {CODE}

Answer(Yes or No only): Yes
```

Table 16: Full prompt of VANILLA baseline for binary assessment task. Blue text is an example of model output. Brown text is the problem, reference, and code we provide to LLMs.

```
Determine the helpfulness of the code snippet. Output a score from 0 to 4 where 0 means the code
snippet is not helpful at all and 4 means the code snippet is very helpful.

Problem Statement: {PROBLEM}

Code Snippet: {CODE}

Helpfulness (0-4): 4
```

Table 17: Full prompt of VANILLA baseline for code deviation assessment task. Blue text is an example of model output. Brown text is the problem, reference, and code we provide to LLMs.

```
Analysis Subtask

You will be provided with a problem statement and a code snippet that supposedly addresses the
problem in {LANGUAGE}.
Your task is to check if the code snippet covers the required functionalities. Do not provide a
corrected version.
Evaluation Steps:
1. Read the problem statement carefully and identify the required functionalities of the
implementation. You can refer to the example to understand the problem better.
2. Read the code snippet and analyze its logic. Check if the code snippet covers all the required
functionalities of the problem.
3. Finally, conclude your evaluation.

Problem Statement: {PROBLEM}
Code Snippet: {CODE}

<ANALYSIS>
Summarization Subtask

You will be provided with an analysis result of a code snippet.
If the analysis believes that the code snippet is correct, output: "Yes". Otherwise, output: "No".

Analysis Result: {ANALYSIS}

Yes
```

Table 18: Full prompt of ANALYZE THEN SUMMARIZE method. Blue text is an example of model output. Brown text is the problem and code we provide to LLMs.

```
You will be provided with a problem statement, a code snippet that supposedly addresses the problem,
and a catalog of code inconsistencies.

Evaluation Steps:
1. Read the problem statement carefully to identify the functionalities required for the
implementation.
2. Read the code snippet and compare it to the problem statement. Check if the code snippet covers
the required functionalities.
3. Output your answer in a JSON format list.
  a) If the code snippet is correct, output: ["inconsistency": "None", "severity": "Negligible"].
  b) If the code snippet is incorrect, output the identified inconsistencies and their severity
according to the catalog of code inconsistencies. For example: ["inconsistency": "<inconsistency1>",
"severity": "<severity1>", "inconsistency": "<inconsistency2>", "severity": "<severity2>", ...]

Problem: {PROBLEM}

Code Snippet: {CODE}

Taxonomy of Common Inconsistencies:
1. Missing dependency declarations: Negligible
2. No error messages for unexpected input cases: Negligible
3. Inefficiency, unnecessary statements: Negligible
4. Edge case not handled: Small
5. Logic error: Major
6. Function or variable not defined: Fatal
7. Code not completed: Fatal
Evaluation Form:
JSON output (a JSON list only):

[{"inconsistency": "None", "severity": "Negligible"}]
```

Table 19: Full prompt of FAULT LOCALIZATION method. Blue text is an example of model output. Brown text is the problem and code we provide to LLMs.