Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

May 20, 2025

LLM as Code-Correctness Judge

Unveiling the Causes of Large-Language-Model Failures when Assessing Java Code

POLAD BAKHISHZADE

Abstract

Context. Large Language Models (LLMs) are increasingly being used in software development tasks such as code generation, explanation, and bug fixing. An emerging frontier is the use of LLMs for code review—specifically, assessing whether a given implementation is functionally correct. This requires reasoning over both the specification and the code, making it a strong benchmark for code understanding.

Objective. This study investigates how the quality and quantity of natural-language context affect the ability of LLMs to assess code correctness. We ask: *What kind of descriptive information helps or harms the model's ability to judge whether a function is correct?*

Method. We extend the Java subset of the CoderEval benchmark by systematically enriching it with five structured documentation layers—ranging from brief summaries to formal pre/post-conditions. Each of 362 functions is paired with one correct and one incorrect candidate implementation. We then prompt three open LLMs—Qwen-0.5B, Qwen-1.5B, and DeepSeek-1.3B—to act as zero-shot code judges, under multiple prompt configurations.

Findings. The results reveal that different models respond differently to added context: smaller models benefit from concise behavioral descriptions but often degrade when presented with verbose examples or formal constraints. In contrast, larger models leverage detailed descriptions to reduce mistakes and achieve more reliable judgments. Some layers, such as examples, can introduce confusion and accuracy drop depending on the model.

Outcome. The study highlights the importance of model-specific prompt design and shows that more documentation is not always better. These insights can inform future systems that rely on LLMs for code quality assessment.

Keywords: LLMs; Code Review; Code Correctness; Prompt Engineering; Evaluation; Java; Dataset Enrichment

Advisor:
Gabriele Bavota
Co-advisor:
Giuseppe Crupi

Approved by the advisor on Date:

Contents

1 Introduction

Large Language Models (LLMs) such as ChatGPT [10], GitHub Copilot [3], and StarCoder [7] have become widely adopted in software development workflows. Originally developed for general language understanding, these models have rapidly gained traction in code-related tasks, including code generation, explanation, repair, and synthesis [5, 2, 13]. Researchers are increasingly exploring the use of LLMs for tasks such as debugging, static analysis, and functional correctness evaluation.

Among the many challenges in this space, one particularly important question is whether LLMs can serve as reliable evaluators of automatically generated code. In industrial settings, such as at Google, the proportion of code written or assisted by machine learning systems is steadily increasing. However, verifying the correctness of such code remains an open problem. Traditional techniques like writing unit tests are time-consuming and require substantial human effort, creating a bottleneck in workflows that rely heavily on automated code generation. Existing automatic evaluation metrics—such as BLEU [11], ROUGE [8], METEOR [1], and BERTScore [16]—have shown significant limitations in this setting. These metrics often rely on reference implementations, which may not be available, and struggle to capture functional equivalence between semantically correct but syntactically different solutions. Recent studies also highlight flaws in applying embedding-based metrics to code-related tasks [9]. This motivates the need for oracle-free evaluation mechanisms capable of assessing whether a candidate implementation adheres to a given specification.

Despite promising results in code generation, LLMs' reliability in evaluating correctness remains mostly unexplored. Much of the existing work focuses on generation benchmarks with minimal or vague natural language descriptions, offering limited insight into how models reason about correctness. Furthermore, it is unclear how the structure and depth of the task description impact the model's evaluation ability.

In this work, we investigate how the judging abilities of LLMs vary with the depth and completeness of the instructions provided, to see whether more detailed and structured descriptions help LLMs better judge the correctness of candidate implementations. Specifically, we extend the CoderEval benchmark by enriching each function with five levels of natural language documentation, ranging from a one-line summary to detailed behavioral, signature-level, example-based, and pre/postcondition specifications. We evaluate three instruction-tuned models—Qwen-0.5B, Qwen-1.5B, and DeepSeek-1.3B—across 362 function–candidate pairs, comparing their accuracy under two conditions: *cumulative prompts*, where each level is incrementally added, and *ablation prompts*, where specific levels are removed to test their individual impact.

Our results suggest that the ability of LLMs to utilize different types of documentation varies across models. For instance, Qwen-1.5B shows a modest improvement in accuracy from 50% to 55% when provided with more detailed prompts, indicating some benefit from richer context. In contrast, smaller models like Qwen-0.5B and DeepSeek-1.3B exhibit inconsistent performance, sometimes declining with additional information. These findings suggest that prompt design can influence model performance, but its effectiveness varies: while richer prompts help some models slightly, others show inconsistent or even worse results.

Report structure

Section 2 reviews related literature. Section 3 presents our study design and dataset enrichment pipeline. Section 4 covers results from cumulative and ablation experiments. Section 5 concludes and outlines future directions.

2 Related Work

As Large Language Models (LLMs) gain traction in software development, a growing body of research is investigating their potential not only as code generators but also as autonomous evaluators. Yet, assessing whether a model can reliably judge the correctness of code remains an open and challenging question. This section presents a narrative overview of related efforts, focusing on the current limitations of LLM-based evaluation, emerging benchmark strategies, and how these studies relate to and inform our own.

A key concern raised by recent work is that LLMs often lack deep semantic understanding of their own outputs. West et al. [14] formulate this as the "Generative AI Paradox": models that generate impressive solutions may still fail to understand them. Their findings show that even top-tier LLMs frequently make overconfident yet incorrect

predictions, especially on reasoning-intensive tasks. Gu et al. [4] expand this concern in the coding domain. They construct "counterfeit" solutions—plausible-looking but incorrect code—and show that LLMs often fail to flag these as erroneous. Instead, models are easily misled by surface-level cues and rarely recover from their own failures. Together, these studies establish a fundamental challenge: judging correctness is a fundamentally different task from generating code and cannot be guaranteed just because a model generates good-looking solutions.

To directly address this, several studies have designed benchmarks to test LLMs' judgment ability in isolation. Zhao et al. [17] introduce CODEJUDGE-EVAL, a benchmark where LLMs must decide whether a given candidate solution is correct. The dataset includes subtle bugs, edge cases, and misleading code snippets. Across 12 models, including GPT-4 and Claude, performance varied widely—with even the best models making systematic errors. This shows that correctness evaluation is still unreliable and highlights the need for structured prompt design and better supervision.

An influential direction comes from ICE-Score, introduced by Zhuo [18], which proposes that instead of relying on test cases or ground-truth outputs, models can be guided to score code implementations directly through natural language. The ICE-Score metric uses detailed prompts that instruct LLMs to evaluate correctness, readability, and structure in a manner aligned with human expectations. Evaluations across multiple programming tasks show strong correlation with expert judgment, especially when models are explicitly instructed to reflect before answering.

Expanding on this idea, Tong and Zhang [12] develop CODEJUDGE, a framework that applies ICE-Score's principles through a structured, multi-step evaluation process. LLMs are guided through several reasoning stages—reflection, explanation, and final verdict—before issuing a correctness label. Experiments reveal significant improvements in judgment accuracy across five datasets and languages, particularly for smaller models like Llama-3-8B-Instruct. In our study, we build on these insights but take a different angle: instead of structuring the reasoning steps, we structure the task description itself. By layering increasingly rich information—ranging from summaries to behavioral specifications and input/output examples—we investigate whether more detailed descriptions help models act as better judges.

While the studies above focus on correctness, others explore how to align model judgments with broader coding preferences. Weyssow et al. [15] introduce CODEULTRAFEEDBACK, a dataset of 10,000 coding tasks evaluated across five dimensions such as style, readability, and instruction-following. Rather than using test-based metrics, they employ GPT-3.5 as a judge and show that supervised fine-tuning on this feedback improves both stylistic alignment and correctness. Their work illustrates that LLMs can be guided to act as evaluators—though their goals differ from ours, which are rooted in functional validation.

A complementary line of research examines LLMs in educational settings. Koutcheme et al. [6] assess whether models can generate and evaluate feedback on student code submissions. Focusing on introductory Python programming, they show that open-source LLMs like StarCoder2 perform competitively with proprietary systems, especially when prompted with worked examples. Although their setting targets formative feedback rather than formal correctness, the underlying task—evaluating code quality—overlaps conceptually with ours and affirms growing confidence in LLMs' evaluative potential.

In summary, the field is converging on the idea that LLMs can act as judges, but only under carefully crafted conditions. While prior work has focused on benchmarks, multi-step reasoning, or feedback alignment, our study lies in incrementally enriching the prompt descriptions and examining how different model sizes respond to varying levels of contextual detail when judging code correctness.

3 Study Design

Research question

RQ – *To what extent the information reported in the code description impact the ability of the LLMs to judge the correctness of a given code?*

3.1 Models evaluated

- **Qwen-0.5B Instruct** (0.5 B params).

- **Qwen-1.5B Instruct** (1.5 B).
- **DeepSeek-Coder-1.3B Instruct**.

3.2 Dataset

From the Java subset of CODEREVAL:

a) Removed noisy IDs.

b) For each remaining ID, selected one correct and one incorrect candidate (existing model outputs).

Final size: **362** rows (balanced).

3.3 Documentation layers

L1
Summary (1 sentence).

L2
Behaviour narrative (detailed summary).

L3
Signature description (argument/return semantics).

L4
Examples (I/O pairs).

L5
Pre/post-conditions.

ChatGPT-4o generated all layers for each of 181 Java functions.

3.4 Prompt template

Constant preamble: task, label scheme (‘0/1‘, no extra text).
Variable payload: selected layers + *first line of candidate code* (argument/return).
Generation: temperature 0.2, max_new_tokens 20.

3.5 Experimental conditions

Incremental (5 runs):

L1; L1+L2; L1+L2+L3; L1+L2+L3+L4; L1+L2+L3+L4+L5

Ablations:

No L1; No L2; No L4; No L5

3.6 Metrics

TP, TN, FP, FN; Accuracy = (TP+TN)/362.

3.7 Project Repository

`github.com/poladbachs/Bachelor-Thesis`.

4 Results & Discussion

4.1 Incremental enrichment (L1 → L5)

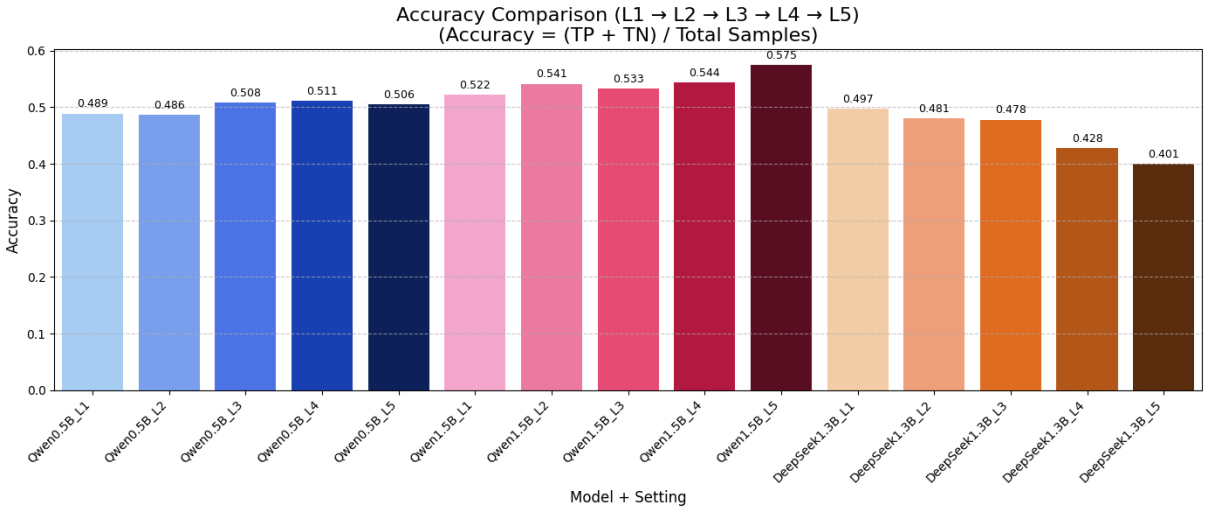

Figure 1. Accuracy per model and documentation level.

Accuracy trends. We observe clear divergence in how each model responds to additional natural-language context:

- **Qwen-0.5B**: After stagnating on L1–L2, the model improves through L3 and L4, peaking at 0.511. L5 slightly degrades performance, suggesting diminishing returns or confusion from formal logic.
- **Qwen-1.5B**: Displays consistent benefit from each additional layer, rising steadily to 0.575 at L5. This model appears to handle verbosity well and leverages the full specification.
- **DeepSeek-1.3B**: Begins strong at 0.497 with minimal input, but drops with every added level—falling below 0.41 at L5. More context leads to worse performance, implying overload or poor instruction tuning for interpretive tasks.

These trends support our core hypothesis: smaller models benefit from concise cues, while larger models thrive on richer context. However, more detail is not universally helpful—its usefulness depends on model capacity and internal robustness to verbosity.

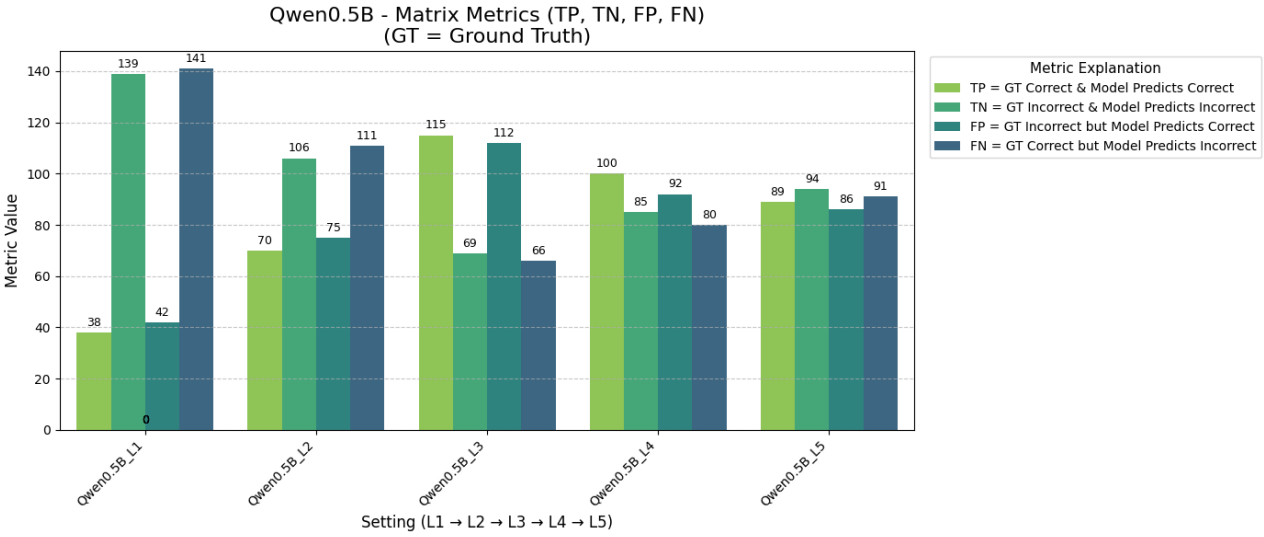

Figure 2. Confusion-matrix metrics — Qwen-0.5B.

Interpretation — Qwen-0.5B. Early stages (L1–L2) show a struggle to distinguish correct from incorrect implementations, with high false negatives. L3 and L4 reduce FN significantly, converting many correct cases into true positives. However, at L5 false positives creep back in, suggesting that formal conditions (pre/post) mislead the model or increase ambiguity. Overall, Qwen-0.5B needs concise clarity—too much structure destabilizes its judgment.

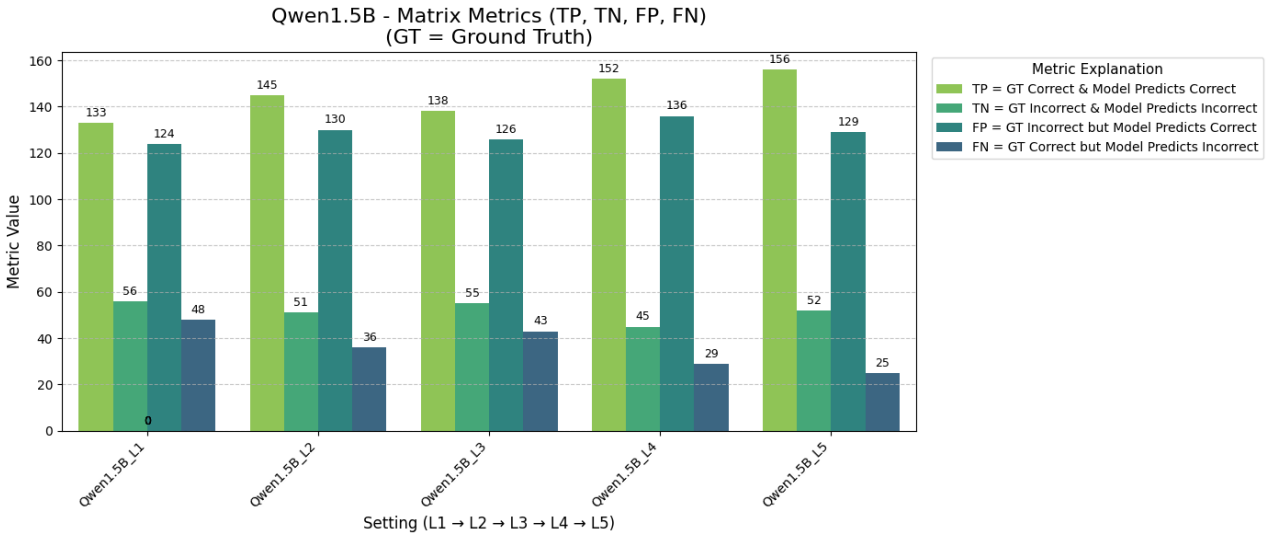

Figure 3. Confusion-matrix metrics — Qwen-1.5B.

Interpretation — Qwen-1.5B. The larger Qwen model handles incremental context with ease. As layers are added, false negatives drop sharply while true positives grow, especially after L4–L5. Pre/post-conditions help the model rule out wrong implementations without over-predicting correctness (FP stays stable). The balance between increased TP and maintained TN implies that Qwen-1.5B is the most context-aware model among the three.

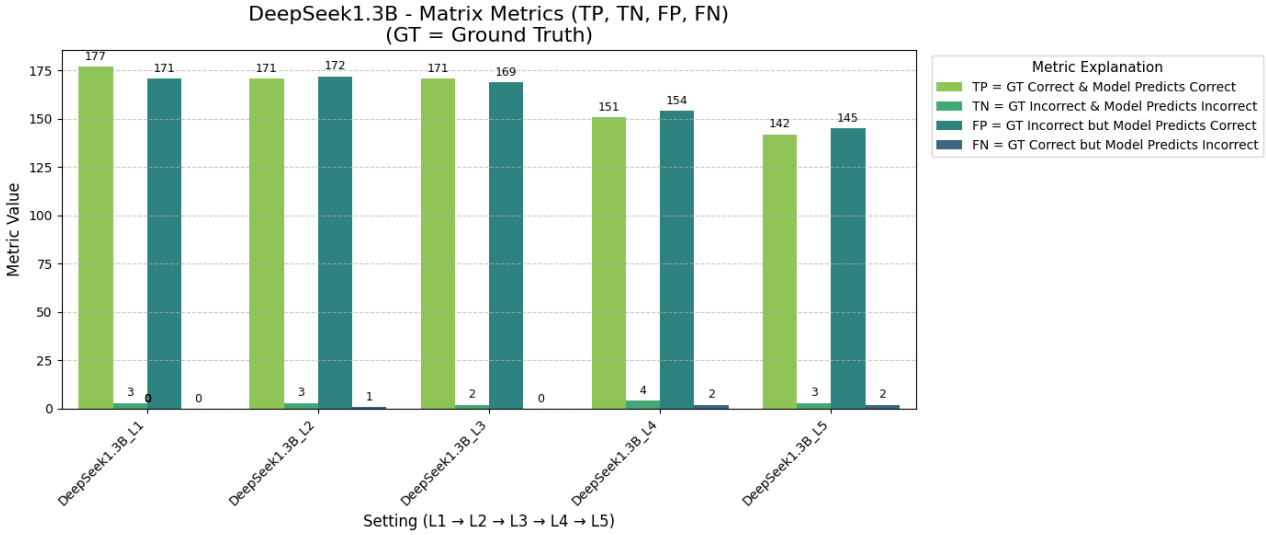

Figure 4. Confusion-matrix metrics — DeepSeek-1.3B.

Interpretation — DeepSeek-1.3B. Despite its comparable size, DeepSeek behaves like a brittle small model in judgment tasks. At L1 and L2, it achieves high TP with minimal FN. But starting at L3, false positives spike and true negatives collapse. This suggests that the added details—examples, constraints—shift attention away from core logic and confuse decision boundaries. It likely overfocuses on surface patterns in examples, falsely validating buggy code.

4.2 Single-layer ablations

Table 1. Accuracy after removing one layer. Values in parentheses show the change relative to the full prompt (L1–L5).

Model	No L1	No L2	No L4
Qwen-0.5B	0.481 (–0.025)	**0.450 (–0.056)**	0.494 (–0.012)
Qwen-1.5B	0.572 (–0.003)	0.558 (–0.017)	0.552 (–0.023)
DeepSeek-1.3B	**0.500 (+0.099)**	0.434 (+0.034)	0.428 (+0.028)

Interpretation.

- Removing behaviour (L2) cripples Qwen-0.5B (–5.6pp) $A \Rightarrow B$ small model needs that layer.
- Summary (L1) is useless or noisy for DeepSeek (+9.9pp when removed).
- Qwen-1.5B tolerates all removals ($\leq$2pp), showing robustness.

4.3 Compound ablations (illustrative)

For space we discuss one illustrative combo per model:

- **Qwen-0.5B No L1 & No L4**: accuracy 0.506 (–0.006 vs. full) — removing summary offsets example noise.
- **Qwen-1.5B No L1 & No L4**: 0.558 (–0.017) — examples help this model.
- **DeepSeek No L2 & No L4**: 0.478 (+0.078) — minimal prompt is best.

4.4 What makes a layer "noisy"?

Manual error analysis highlights three patterns:

a) **Example bias**. L4 lists only happy-path cases $\rightarrow$ model over-predicts "correct".
b) **Narrative redundancy**. L2 restates corner cases differently from L5, confusing alignment.
c) **Generic summaries**. L1 adds little entropy; for DeepSeek it seems to trigger a "looks plausible $\rightarrow$ correct" bias.

5 Conclusions & Future Work

- Moderate enrichment (+behaviour, +signature) boosts small models (+8pp).
- Verbose layers (examples, formal conditions) help mid-size models but harm 0.5 B and DeepSeek.
- Removing noisy layers (summary or examples) can recover performance (DeepSeek +10pp).

Next steps.

1. Fine-tune a 0.5 B checkpoint on the enriched dataset.
2. Auto-select minimal counter-examples to make L4 compact and balanced.
3. Extend to Python and larger open models (e.g. Llama-3-8B).

References

[1] S. Banerjee and A. Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *ACL Workshop*, 2005.

[2] M. Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] GitHub. Github copilot. `https://github.com/features/copilot`, 2023.

[4] A. Gu, W.-D. Li, N. Jain, T. X. Olausson, C. Lee, K. Sen, and A. Solar-Lezama. The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? *arXiv preprint arXiv:2402.19475*, 2024.

[5] J. Jiang et al. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[6] C. Koutcheme, N. Dainese, S. Sarsa, A. Hellas, J. Leinonen, S. Ashraf, and P. Denny. Evaluating language models for generating and judging programming feedback. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, pages 624–630, 2025.

[7] Y. Li et al. Starcoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[8] C.-Y. Lin. Rouge: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, 2004.

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

[10] OpenAI. Chatgpt. `https://openai.com/chatgpt`, 2023.

[11] K. Papineni et al. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of ACL*, 2002.

[12] W. Tong and T. Zhang. Codejudge: Evaluating code generation with large language models. *arXiv preprint arXiv:2410.02184*, 2024.

[13] P. Wang et al. Codet: Code generation with generated tests. *arXiv preprint arXiv:2305.14278*, 2023.

[14] P. West, X. Lu, N. Dziri, F. Brahman, L. Li, J. D. Hwang, L. Jiang, J. Fisher, A. Ravichander, K. R. Chandu, et al. The generative ai paradox: "what it can create, it may not understand". *arXiv preprint arXiv:2311.00059*, 2023.

[15] M. Weyssow, A. Kamanda, X. Zhou, and H. Sahraoui. Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences. *arXiv preprint arXiv:2403.09032*, 2024.

[16] T. Zhang et al. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.

[17] Y. Zhao, Z. Luo, Y. Tian, H. Lin, W. Yan, A. Li, and J. Ma. Codejudge-eval: Can large language models be good judges in code understanding? *arXiv preprint arXiv:2408.10718*, 2024.

[18] T. Y. Zhuo. Ice-score: Instructing large language models to evaluate code. *arXiv preprint arXiv:2304.14317*, 2023.