Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

May 19, 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, GitHub Copilot, and StarCoder 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. Researchers are increasingly exploring the use of LLMs for more advanced code understanding 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. This motivates the need for automatic, 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 underexplored. Much of the existing work focuses on generation benchmarks with minimal or vague natural language descriptions, which offer 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 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 cumulative and ablation prompt settings.

Our results suggest that model capacity plays a significant role in how different types of documentation are utilized. Smaller models, such as Qwen-0.5B, benefit from concise behavioral context but are negatively impacted by verbose or redundant input. In contrast, Qwen-1.5B leverages detailed structure effectively, showing consistent accuracy gains as more layers are added. Meanwhile, DeepSeek-1.3B performs best with minimal input and degrades when faced with richer prompts—suggesting sensitivity to verbosity and limited ability to exploit complex descriptions. These findings highlight the importance of prompt design in aligning LLM reasoning with software correctness.

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

The use of Large Language Models (LLMs) for code understanding and evaluation has become an active area of research. While LLMs have demonstrated impressive capabilities in code generation tasks, recent work highlights a critical gap in their ability to evaluate or judge the correctness of code—a task that requires genuine semantic understanding rather than pattern memorization.

Several studies have investigated this limitation. West et al. [4] introduce the "Generative AI Paradox", noting that models often produce high-quality outputs without fully understanding them. Gu et al. [1] further expose this gap through the "Counterfeit Conundrum", showing that LLMs struggle to recognize subtle flaws in their own generations. These findings suggest that correctness judgment is a non-trivial challenge, especially in the absence of explicit oracles such as test cases.

In response to this, researchers have proposed new benchmarks and evaluation strategies. Zhao et al. [6] present CodeJudge-Eval, a benchmark specifically designed to test whether LLMs can distinguish correct from incorrect implementations. The results reveal that even strong models frequently misclassify buggy or misleading solutions, reinforcing that functional correctness evaluation remains an open question. Similarly, Zhuo [7] introduces ICE-Score,

an instruction-based metric where LLMs are explicitly guided to assess code correctness and quality. This approach avoids traditional test-based evaluation and aligns with the growing interest in oracle-free evaluation pipelines.

Building on ICE-Score, Tong and Zhang [3] propose CodeJudge, a more structured framework where LLMs are prompted to follow a multi-step "slow thinking" process before rendering a correctness decision. Their results demonstrate that structured prompting can significantly improve evaluation quality, even when using relatively small models. Our study builds on these insights by also employing layered prompt structures but focuses more explicitly on how the level of detail and structure in descriptions interacts with model capacity.

Other efforts have focused on aligning LLM judgments with human preferences or pedagogical goals. Weyssow et al. [5] introduce CodeUltraFeedback, a dataset where 14 LLMs respond to coding instructions and are judged on dimensions such as readability, style, and instruction-following. Their findings suggest that LLMs can be aligned to human-like evaluation standards through supervised fine-tuning. While our work also investigates LLMs-as-judges, our focus is on functional correctness rather than stylistic alignment.

The educational domain provides additional context. Koutcheme et al. [2] explore whether LLMs can generate and assess programming feedback in student assignments. Their findings suggest that LLMs—both open-source and proprietary—are approaching the reliability of human graders in introductory programming contexts. Though focused on feedback generation rather than correctness verification, this line of work highlights the growing confidence in LLM-based evaluation systems.

In summary, prior work has identified the limits of current LLMs in judging code, proposed structured prompting techniques and evaluation metrics, and explored domain-specific use cases. Our study complements these directions by 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)

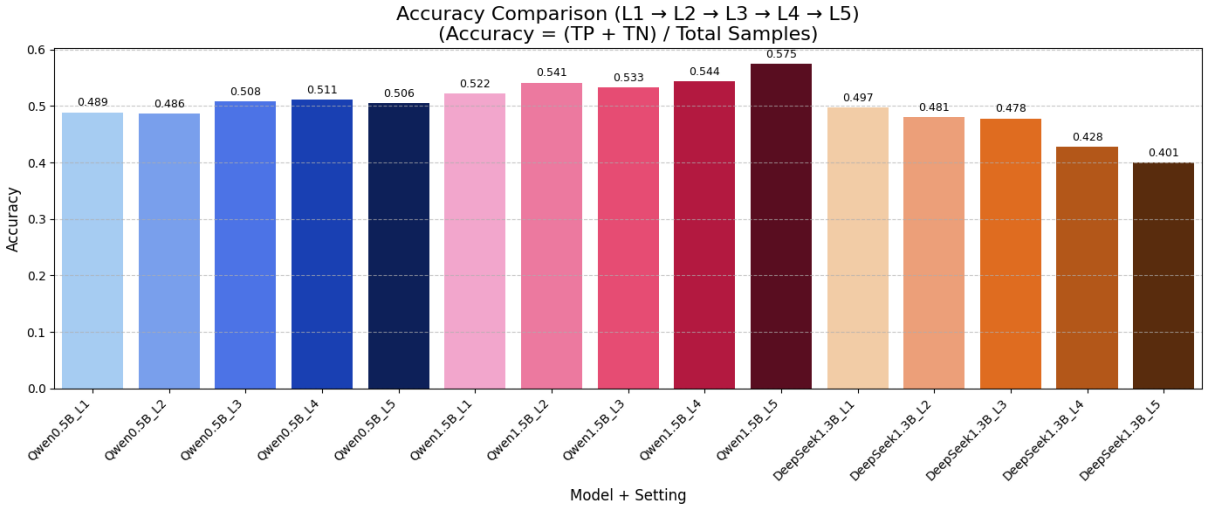

Figure 1. Accuracy per model and documentation level.

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.

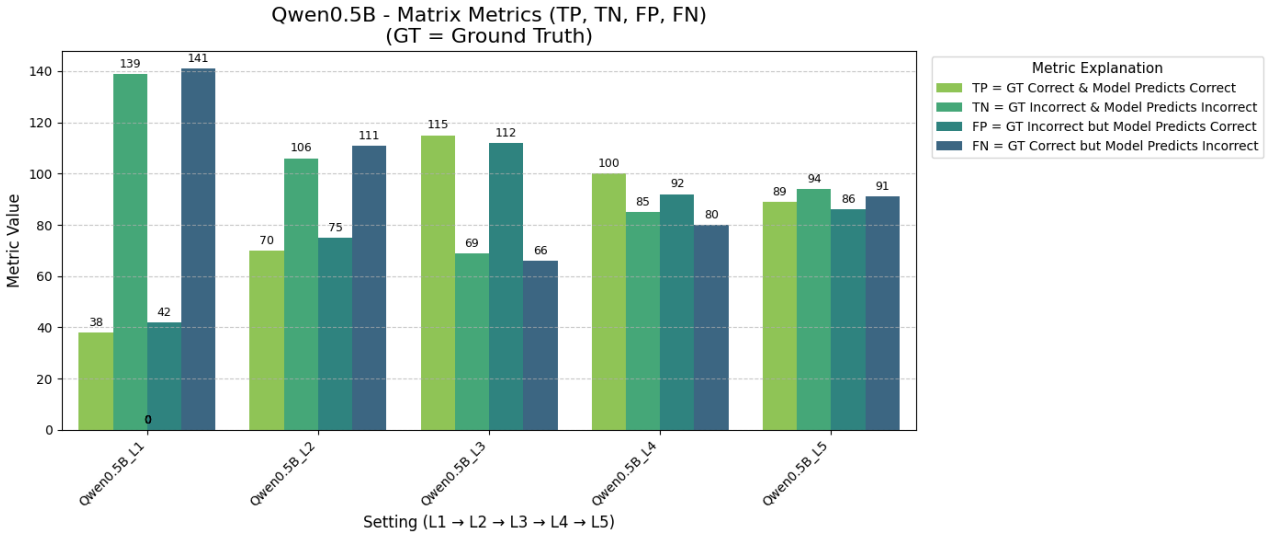

Figure 2. Confusion-matrix metrics — Qwen-0.5B.

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.

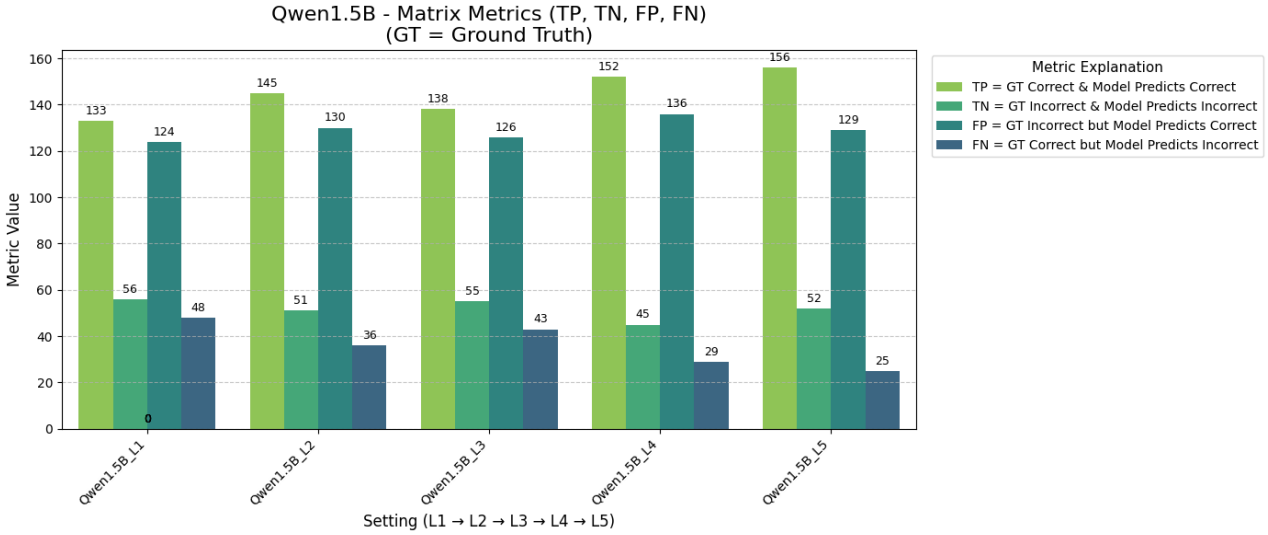

Figure 3. Confusion-matrix metrics — Qwen-1.5B.

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.

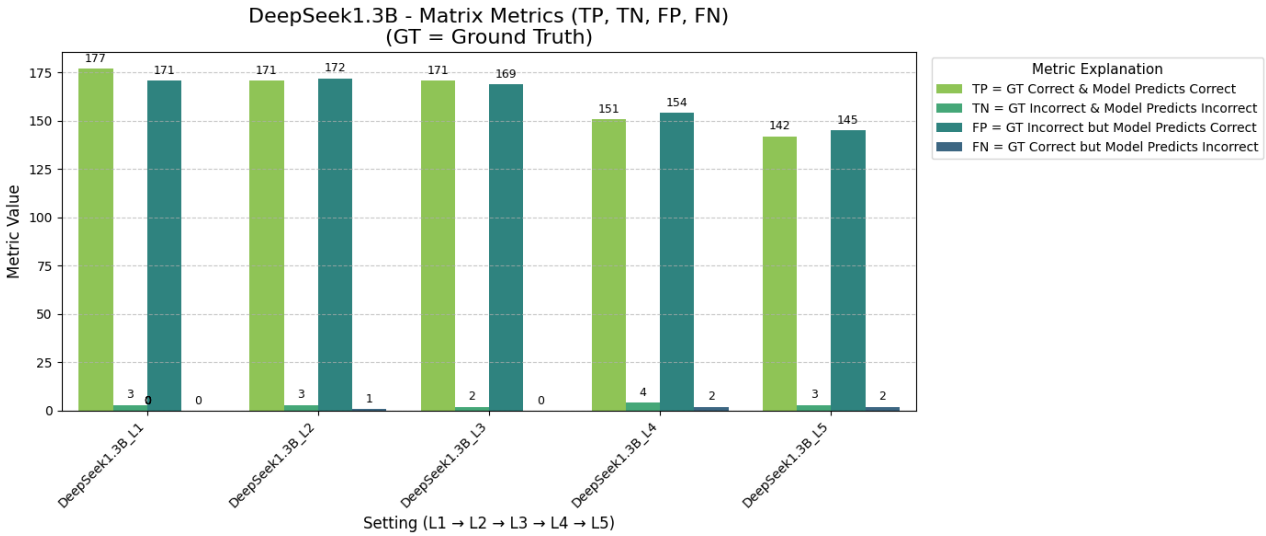

Figure 4. Confusion-matrix metrics — DeepSeek-1.3B.

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 → 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 → 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] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

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