Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

May 18, 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

1.1 LLMs and the promise of autonomous code review

Tools such as GitHub Copilot, ChatGPT Code Interpreter, and StarCoder have moved LLMs from research prototypes into the daily workflow of developers. Beyond generating or explaining code, a natural next milestone is *autonomous code review*—having the model assess whether an implementation correctly matches a given specification. At the core of this task is the LLM's ability to judge **functional correctness**.

1.2 Why correctness judgement matters

Failure to judge correctness accurately can cause critical issues in two major settings:

S1 **Pull-request triage**: Models that approve buggy code or reject valid patches mislead developers and reduce trust in automation.

S2 **Self-refinement loops**: Recent LLM-based code generators incorporate feedback loops where the model critiques and iteratively improves its own output. These loops are only effective if the internal judge is reliable.

1.3 Gaps in current understanding

While LLMs have shown impressive results in generation and repair tasks, their reliability in correctness judgment remains underexplored. Moreover, existing benchmarks often include only short or vague natural-language descriptions, potentially limiting the model's reasoning capability.

1.4 Our hypothesis: prompt complexity vs. model capacity

Intuitively, richer natural-language context should help LLMs reason more accurately. But overly verbose prompts may contain redundancy, ambiguity, or distractions. We hypothesize that:

Smaller models benefit from concise, focused descriptions, while larger models can exploit more detailed and structured context.

1.5 Approach overview

To test this hypothesis, we extended the CoderEval benchmark with five structured documentation layers:

- L1: one-line summary,
- L2: behavioral description,
- L3: signature semantics,
- L4: examples (input/output),
- L5: pre- and post-conditions.

We ran three instruction-tuned LLMs (Qwen-0.5B, Qwen-1.5B, DeepSeek-1.3B) on 362 function–candidate pairs, testing both *cumulative* prompts (gradually adding more documentation layers) and *ablation* prompts (removing one specific layer at a time to isolate its impact, if any). A unified evaluation pipeline was used to measure accuracy and confusion metrics.

1.6 Key insights

Our findings confirm the importance of tailoring prompts to model capacity:

- Small models improve with behavior descriptions but are easily misled by verbose context.
- Larger models leverage all five layers to reduce misclassifications and false negatives.
- Some documentation elements (e.g., examples) can hurt performance if not well-integrated.

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 application of Large Language Models (LLMs) as evaluators in software engineering tasks has garnered significant attention. This section reviews pivotal studies that explore LLMs' capabilities in assessing code correctness, understanding, and alignment with coding preferences, providing context and contrast to our investigation into LLM-based code correctness judgment.

2.1 LLMs as Judges: Datasets and Benchmarks

CodeUltraFeedback by Weyssow et al. [5] introduces a dataset comprising 10,000 coding instructions, each annotated with responses from 14 diverse LLMs. These responses are evaluated based on five coding preferences: instruction following, code explanation, code complexity and efficiency, code readability, and coding style. GPT-3.5 serves as the judge to rank these responses. The study demonstrates that fine-tuning models like CodeLlama-7B-Instruct using this dataset can outperform larger models in aligning with coding preferences and improving functional correctness on benchmarks like HumanEval+. This work underscores the potential of LLMs not only in code generation but also in nuanced evaluation tasks, aligning closely with our focus on LLMs as code correctness judges.

CodeJudge-Eval by Zhao et al. [6] presents a benchmark designed to evaluate LLMs' code understanding abilities from the perspective of code judging rather than code generation. The benchmark includes tasks where models must determine the correctness of provided code solutions, including those with subtle errors or compilation issues. Evaluations of 12 prominent LLMs reveal that even state-of-the-art models struggle with this task, highlighting the need for improved code understanding in LLMs. This benchmark provides a direct comparison point for our study, emphasizing the challenges LLMs face in accurately assessing code correctness.

ICE-Score introduced by Zhuo [7] proposes an evaluation metric that instructs LLMs to assess code quality, addressing the limitations of traditional token-matching metrics like BLEU. ICE-Score correlates more strongly with functional correctness and human preferences, eliminating the need for test oracles or reference implementations. Evaluations across four programming languages demonstrate its effectiveness in aligning with human judgments. This approach aligns with our methodology of leveraging LLMs for code evaluation, reinforcing the viability of instruction-based assessment metrics.

2.2 Understanding LLM Limitations in Code Evaluation

The Generative AI Paradox by West et al. [4] explores the discrepancy between the generative capabilities of LLMs and their understanding of the generated content. The study finds that while models can produce outputs that surpass human performance, they often lack a deep understanding of the content, leading to brittleness and susceptibility to adversarial inputs. This paradox underscores the challenges in relying on LLMs for tasks that require genuine comprehension, a concern directly relevant to our investigation into LLM-based code correctness judgment.

The Counterfeit Conundrum by Gu et al. [1] investigates whether code language models can discern the nuances of their incorrect generations, termed "counterfeits." The study identifies three primary failure modes: models misclassify incorrect code as correct, struggle to reason about the execution behavior of counterfeits, and often fail to repair them effectively. These findings highlight the shallow understanding LLMs have of their erroneous outputs, emphasizing the need for improved evaluation mechanisms, as pursued in our research.

2.3 LLMs in Educational Contexts and Evaluation Frameworks

Evaluating Language Models for Generating and Judging Programming Feedback by Koutcheme et al. [2] assesses the efficiency of open-source LLMs in generating high-quality feedback for programming assignments and judging the quality of programming feedback. Their evaluations on a dataset of students' submissions to introductory Python programming exercises suggest that state-of-the-art open-source LLMs are nearly at the same level with proprietary models in both generating and assessing programming feedback. This finding underscores the potential of open-source models in educational contexts and the importance of further research in this area, complementing our focus on LLMs' evaluative capabilities.

CodeJudge by Tong and Zhang [3] proposes a framework that leverages LLMs to evaluate the semantic correctness of generated code without relying on test cases. The approach involves guiding LLMs through a "slow thinking" process to perform in-depth evaluations. Experiments across four code generation datasets and five programming languages show that CodeJudge outperforms existing methods, achieving better results even with smaller models like Llama-3-8B-Instruct. This framework aligns with our objective of enhancing LLM-based code evaluation, offering insights into effective evaluation strategies.

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):

$$\text{L1; L1+L2; L1+L2+L3; L1+L2+L3+L4; L1+L2+L3+L4+L5}$$

Ablations:

$$\text{No L1; No L2; No L4; No L5}$$

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)

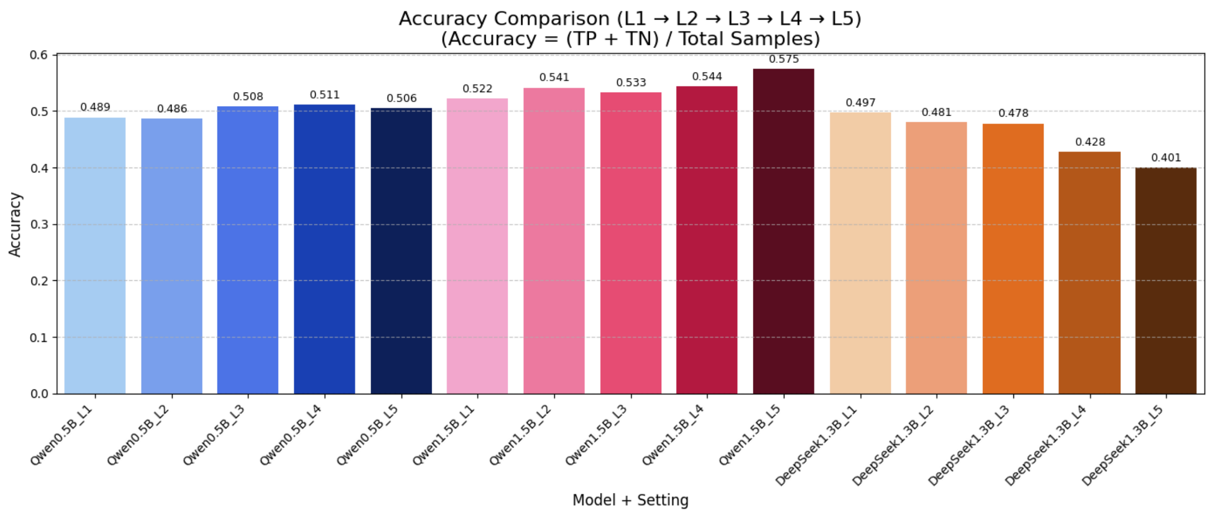


Figure 1. Accuracy per model and documentation level.

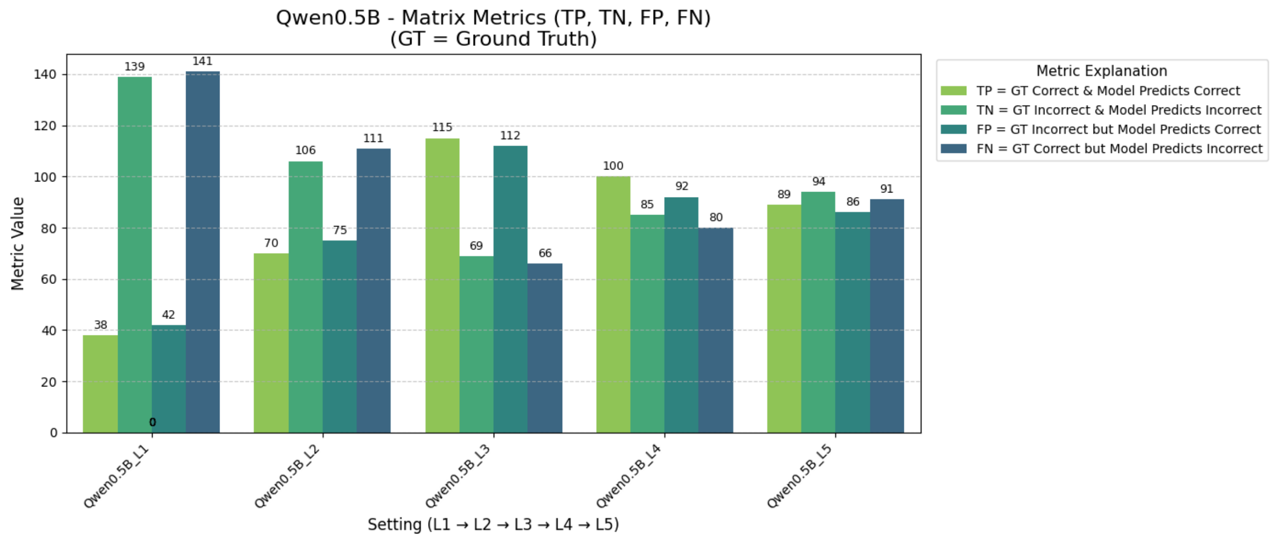
Accuracy trends. We observe clear divergence in how each model responds to additional natural-language context:

- **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.

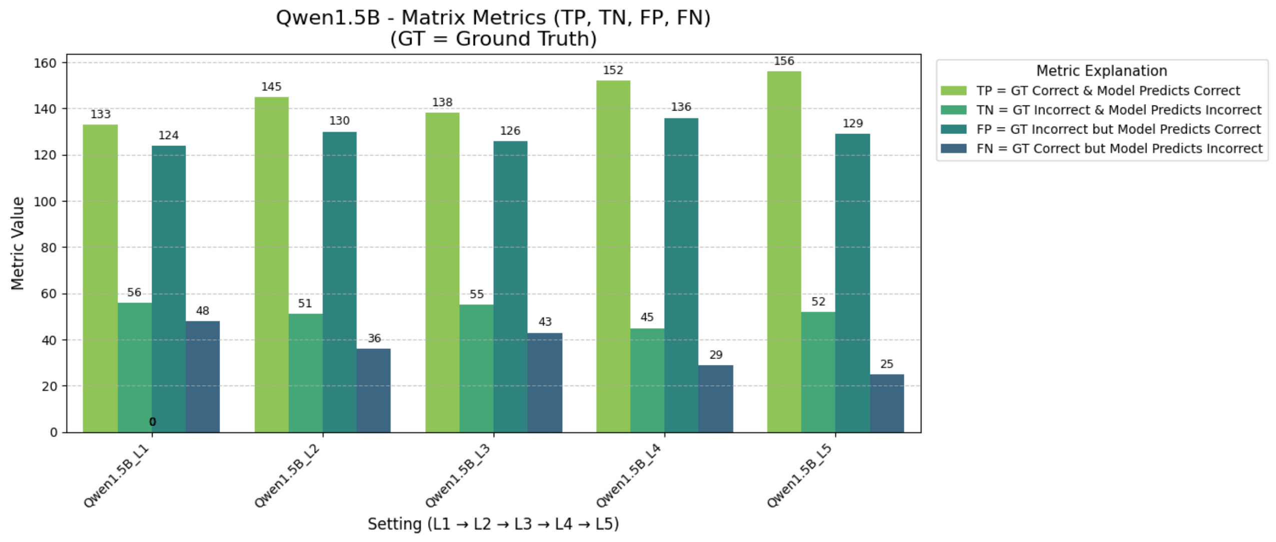
These trends support our core hypothesis: smaller models benefit from concise cues, while larger models thrive on richer context. However, more detail is not universally helpful—its usefulness depends on model capacity and internal robustness to verbosity.



Setting	TP	TN	FP	FN
Qwen0.5B_L1	38	139	42	141
Qwen0.5B_L2	70	106	75	111
Qwen0.5B_L3	115	69	112	66
Qwen0.5B_L4	100	85	92	80
Qwen0.5B_L5	89	94	86	91

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.



Setting	TP	TN	FP	FN
Qwen1.5B_L1	133	56	124	48
Qwen1.5B_L2	145	51	130	36
Qwen1.5B_L3	138	55	126	43
Qwen1.5B_L4	152	45	136	29
Qwen1.5B_L5	156	52	129	25

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.

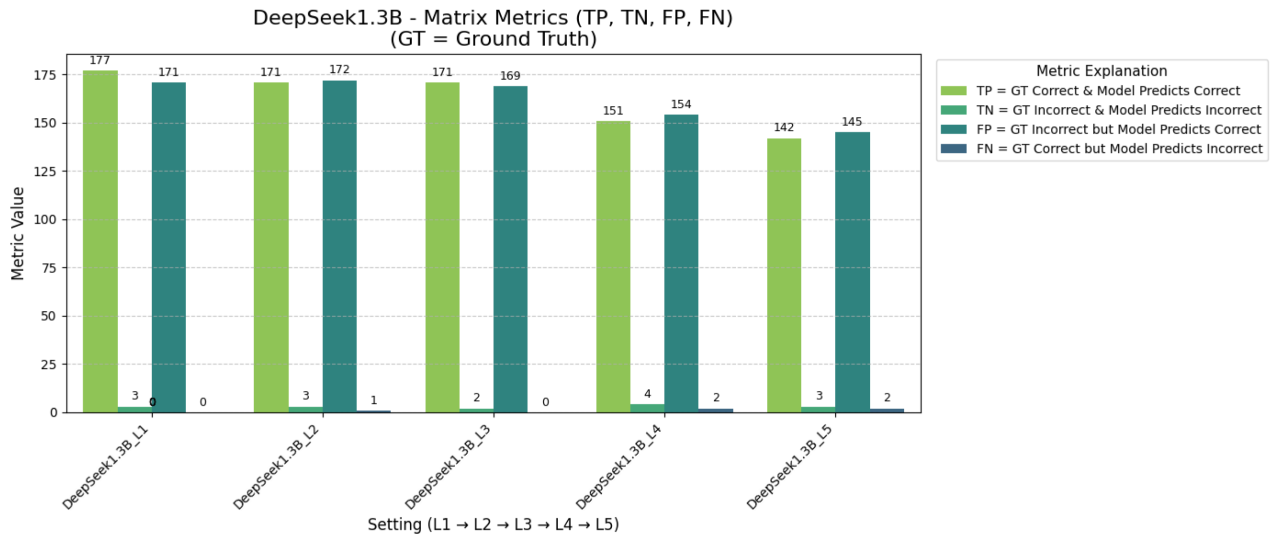


Figure 4. Confusion-matrix metrics — DeepSeek-1.3B.

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 (vs. full 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) small model needs that layer.
- Summary (L1) is useless or noisy for DeepSeek (+9.9pp when removed).
- Qwen-1.5B tolerates all removals (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] 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.