Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

May 25, 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 [13], GitHub Copilot [5], and StarCoder [9] 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 [7, 3, 17]. 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 [15], ROUGE [11], METEOR [2], and BERTScore [20]—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 [12]. This motivates the need for oracle-free evaluation mechanisms capable of assessing whether a candidate implementation adheres to a given specification.

While LLMs have shown strong performance in code generation, an open question remains: can they also reason about the correctness of code? This question has received relatively little attention so far, as most benchmarks are designed for code generation. Moreover, adapting code generation benchmarks for correctness evaluation is not trivial. Many such benchmarks provide only minimal or vague natural language descriptions, which are insufficient to test whether a model understands the functionality of the implementation it is evaluating.

In this work, we investigate how the judging abilities of LLMs vary with the depth and completeness of the instructions provided. Specifically, we create a correctness evaluation benchmark by extending the CoderEval dataset with five levels of natural language documentation. These range from a simple one-line summary of the function's purpose (L1), to a high-level description of the expected behavior and edge cases (L2), an explanation of the function signature and its parameters (L3), concrete input/output examples (L4), and formal pre- and postconditions specifying what must hold before and after execution (L5). We evaluate three instruction-tuned models—Qwen-0.5B, Qwen-1.5B, and DeepSeek-1.3B—across 362 function–candidate pairs. Each pair is tested under two conditions: *cumulative prompts*, where the model sees all levels from L1 up to a given level, and *ablation prompts*, where a specific level (e.g., L4) is removed from the full prompt to isolate its 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 an overview of related efforts, focusing on the current limitations of LLM-based evaluation and recent benchmark development.

A key concern raised by recent work is that LLMs often lack deep semantic understanding of their own outputs. West et al. [18] 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. [6] 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. Collectively, these studies highlight a core insight: evaluating code correctness is fundamentally distinct from generating code. Therefore, it is not safe to assume that a model which produces plausible-looking solutions is also a reliable evaluator of such completions.

Several studies have designed benchmarks to assess LLMs' judgment ability in isolation. Zhao et al. [21] 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, accuracy remained low—rarely exceeding 60%, with GPT-4 reaching at most 65%. These results suggest that current models struggle with code correctness as a task, regardless of their size. Zhao et al. highlight that even advanced models like GPT-4 frequently misclassify incorrect solutions that contain subtle bugs or misleading logic. They suggest that reformulating the task—by giving more precise and complete descriptions of the expected behavior—and training models using datasets that include both correct and incorrect implementations could help improve judgment accuracy.

An influential approach is introduced by ICE-Score from Zhuo [22], which suggests using GPT-3.5-turbo to evaluate candidate implementations using precisely crafted prompts that focus on properties such as correctness and readability. The study shows that prompting the model to explain its reasoning before delivering a final judgment enhances alignment with human evaluations.

Expanding on this idea, Tong and Zhang [16] develop CODEJUDGE, a framework that builds on ICE-Score's principles through a structured, multi-step evaluation process. In this setup, two instruction-tuned models are guided through reasoning stages—including explanation and justification—before returning a binary decision on correctness. Experiments reveal significant improvements in judgment accuracy across five datasets and programming 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 human coding preferences. Weyssow et al. [19] introduce CODEULTRAFEEDBACK, a dataset of 10,000 coding problems, each answered by 14 different LLMs. These outputs are then ranked by GPT-3.5 across five dimensions: style, readability, instruction-following, complexity, and explanation. The authors use these rankings as training data to fine-tune CodeLlama-7B-Instruct, teaching it to prefer responses that GPT-3.5 rated more highly. The fine-tuned model is evaluated on HumanEval+ [10], where it outperforms the original base model in both preference alignment and functional accuracy. Although the task focuses on stylistic and qualitative assessment rather than functional correctness, the results suggest that training with ranked outputs can guide LLMs to better evaluate and generate code along desired dimensions such as style and readability.

A complementary line of research examines LLMs in educational settings. Koutcheme et al. [8] assess whether LLMs can generate and evaluate feedback on student code submissions in introductory Python assignments. Their goal is to explore how models can assist in automated grading systems that support student learning. They find that open-source models like StarCoder2 perform competitively with proprietary models such as GPT-4, particularly when provided with annotated student submissions that include instructor-written feedback, which serve as exemplars for what good feedback should contain. While not directly focused on correctness classification, this work supports the broader feasibility of using LLMs as evaluators of code quality.

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 models with different sizes respond to varying levels of contextual detail when judging code correctness.

3 Study Design

Research question

RQ – *To what extent does the quality and depth of natural language documentation affect the ability of LLMs to judge the correctness of a given Java function?*

3.1 Context: LLMs

We selected three instruction-tuned LLMs of relatively small size to simulate cost-efficient evaluation scenarios: **Qwen2.5-Coder-0.5B-Instruct** and **Qwen2.5-Coder-1.5B-Instruct** [1], both from Alibaba's Qwen family, and **DeepSeek-Coder-1.3B-Instruct** [4]. These models are available on Hugging Face and optimized for multi-turn interaction with structured task prompts. Each model was prompted in identical conditions, across five enrichment levels and multiple ablation runs, to evaluate how their judgment accuracy evolved as more semantic information was added—or selectively removed—from the prompt.

3.2 Context: Dataset

Our base dataset is the publicly available Java subset of the CoderEval benchmark [14], a manually verified corpus of real-world Java functions with associated candidate implementations. We started with 230 function IDs and removed 49 based on prior TA-provided exclusion criteria (e.g., functions that were duplicates, ambiguous, or noisy), ending with 181 valid functions. Each function includes the original code, a natural language docstring, and a human-assigned correctness label. However, in its raw form, CoderEval is generation-focused and lacks structured documentation layers. We therefore augmented each instance with a richer, five-part description to enable code understanding evaluation.

3.3 Data Collection

3.3.1 Augmenting the code generation dataset

To transform CoderEval from a code generation benchmark into a code understanding benchmark, we enriched each Java function with a multi-level natural language description. This transformation was performed using GPT-4o, guided by a custom prompt that forced the model to output structured information in five distinct documentation layers:

L1 **Summary**: a concise one-sentence description of the function's high-level purpose, strictly excluding edge cases.

L2 **Behavioral description**: a 1–2 sentence narrative detailing the function's logic, including how edge cases and special conditions are handled.

L3 **Signature explanation**: structured tags describing the meaning of each parameter and the return value (`@param`, `@return`, `@throws`), if applicable.

L4 **Examples**: several one-line input–output pairs with explanatory notes clarifying expected behavior.

L5 **Preconditions and postconditions**: short logical statements capturing required input constraints and expected output guarantees.

The input to GPT-4o consisted of the full function body and its name, and the output was strictly constrained to follow the five-part format with no extra commentary or filler tokens. During development, multiple iterations of the prompt were tested to reduce verbosity, ensure syntactic consistency, and avoid contamination of L1 with behavioral detail.

Once the final prompt format was locked, we generated all five layers for the full set of 181 function IDs. We manually inspected and corrected the outputs for all rows, initially spot-checking 20–30 examples, then proceeding to a full sweep. Particular attention was paid to ensuring:

- L1 never included edge cases or technical terms.
- L2 correctly expanded the logic and covered edge behavior.
- L3 tags were accurate and omitted irrelevant fields.

- L4 examples reflected both standard and boundary behavior.
- L5 constraints were logically sound and matched the function logic.

After parsing the output into structured columns, the enriched dataset was stored in CSV format, ready for prompt composition.

3.3.2 Generating the LLM's judgments

To evaluate whether LLMs can judge functional correctness from natural language descriptions, we paired each of the 181 functions with two candidate implementations: one correct and one incorrect. These implementations were selected from the official CoderEval knowledge base[1], which contains over 5,000 Java methods previously generated by various models.

We applied a strict filtering process to clean this pool and remove trivial or placeholder code. The cleaning was done using a custom script that rejected candidates based on several heuristics:

- Contained known placeholder phrases such as "`TODO`", "`not implemented`", "`assuming`", etc.
- Contained only print statements or placeholder bodies, such as `System.out.println("To be implemented");`.
- Had extremely low information density, for example: `return;` or `return 0;`.
- Consisted of vague pseudocode or empty shells without logic, such as `/* implementation goes here */`.

For each function, we then attempted to find one correct and one incorrect candidate from the filtered KB. When multiple valid candidates existed, a random sample was selected. If no valid correct candidate was found in the KB, we used the official CoderEval reference code as a fallback. If no valid incorrect candidate was found, we either skipped the entry or, in two edge cases, manually injected incorrect candidates by rewriting a correct one into a subtly incorrect form. This process yielded a total of 362 candidate pairs (181 functions $\times$ 2), forming a balanced dataset.

Each of the three LLMs—Qwen-0.5B, Qwen-1.5B, and DeepSeek-1.3B—was evaluated using a consistent prompt template. The prompt contained:

- A fixed task instruction: "Judge whether the candidate implementation is correct given the function description."
- A binary output schema: 0 = incorrect, 1 = correct.
- One or more of the five documentation layers (L1–L5), depending on the experimental condition.
- The first line of the candidate function (its signature).
- The full candidate code.

The models were loaded via Hugging Face Transformers with the following inference settings: temperature 0.2, max_new_tokens 20. Prompts were fed in via the tokenizer, and the generated output was parsed to extract either `0` or `1`. Any deviations were treated as invalid predictions, outputting None. This pipeline was run across 15 incremental configurations (3 models $\times$ 5 levels), plus 8 additional ablation runs where selected layers were removed to analyze their individual contribution to accuracy. All results were logged, converted to structured CSVs, and stored for downstream analysis and visualization.

3.4 Data Analysis

We analyzed the model predictions along two experimental axes: prompt enrichment and prompt ablation.

In the **prompt enrichment setting**, each model was evaluated across five runs:

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

This resulted in 15 total runs (3 models $\times$ 5 levels). We tracked confusion matrix components—true positives, true negatives, false positives, and false negatives—for each run, and computed accuracy as the primary metric.

[1]https://github.com/poladbachs/Bachelor-Thesis/blob/main/CoderEval/knowlbase_codereval.csv

We then conducted an **ablation study** with 8 additional runs. Here, we removed selected documentation levels from the full prompt, isolating their individual contribution. Importantly, the ablations were not exhaustive. Instead, we focused on levels that appeared least influential based on prior analysis—e.g., removing L4 (Examples) or L5 (Pre/postconditions). In some cases, notably with DeepSeek-1.3B, performance improved when L5 was excluded, suggesting that some prompts introduced noise or confusion rather than clarity.

All outputs and metrics were saved as CSV files, and visualization scripts generated comparative bar plots per model. These allowed us to contrast how models responded to incremental detail versus selective removal of context.

3.5 Replication Package

All source code, enriched datasets, model evaluation scripts, and plotting tools are available in the following GitHub repository:

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

The repository includes all CSVs, ablation results, accuracy plots, and confusion matrix visualizations. Dependencies are listed in a 'requirements.txt', and a step-by-step guide for full replication is provided in the README.

4 Results & Discussion

4.1 Incremental enrichment (L1 → L5)

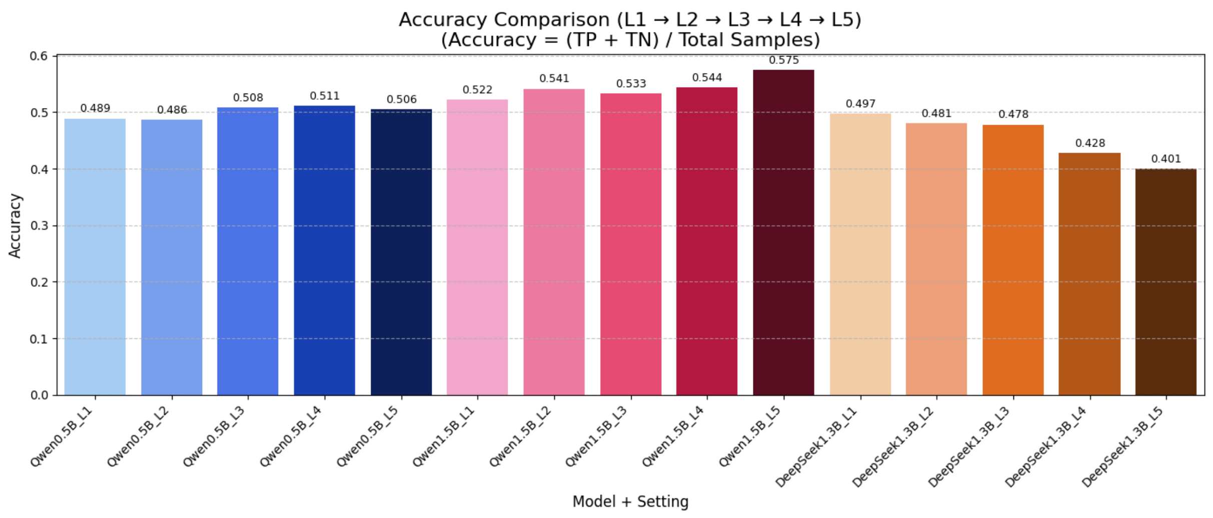

Figure 1. Accuracy per model and documentation level.

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.

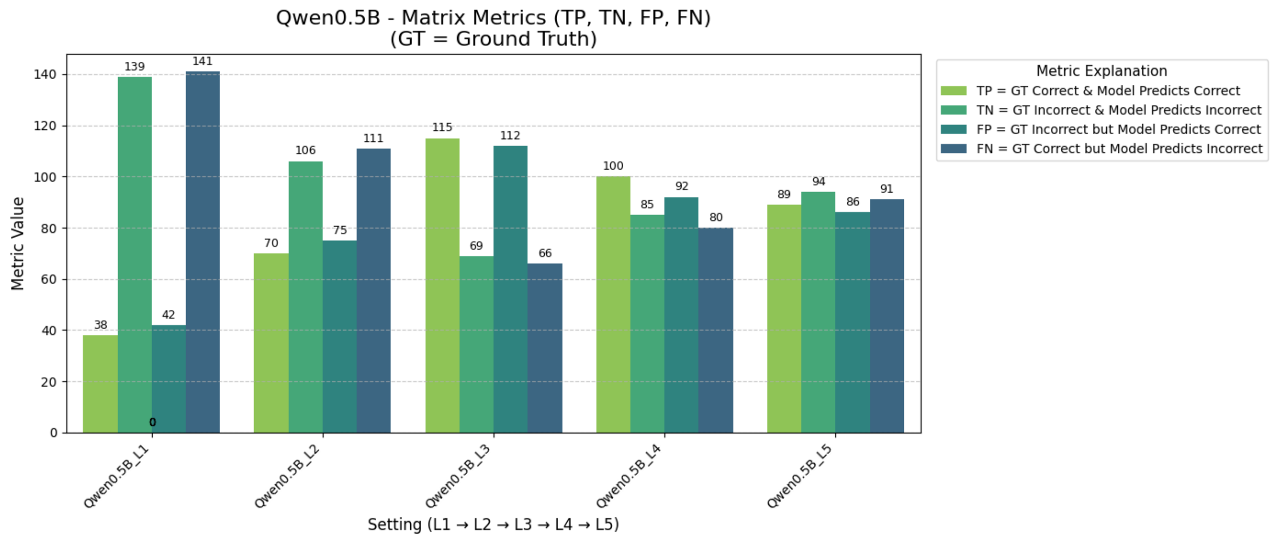

Figure 2. Confusion-matrix metrics — Qwen-0.5B.

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.

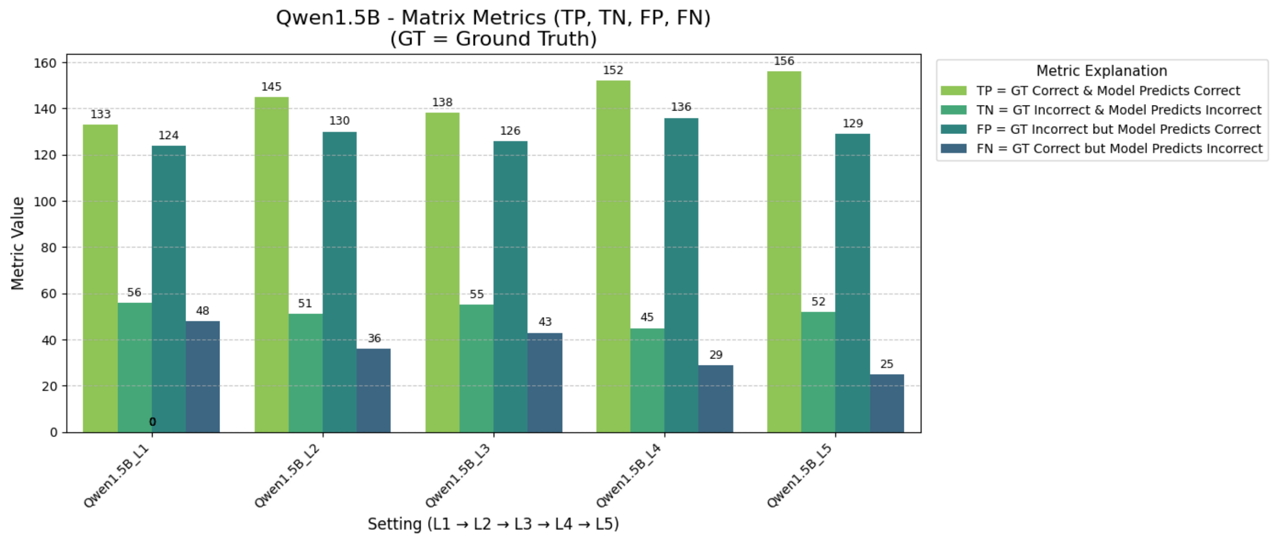

Figure 3. Confusion-matrix metrics — Qwen-1.5B.

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.

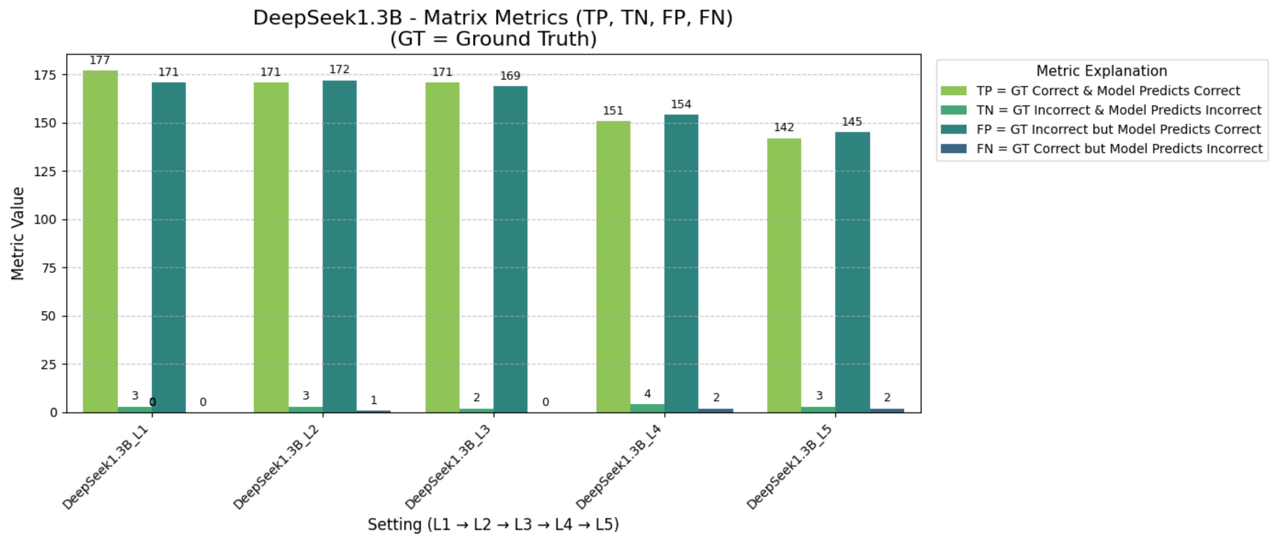

Figure 4. Confusion-matrix metrics — DeepSeek-1.3B.

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 (≤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] Alibaba Group. Qwen2.5-coder models on hugging face. `https://huggingface.co/Qwen/Qwen2.5-Coder-1.5B-Instruct`, 2024. Accessed May 2025.

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

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

[4] DeepSeek AI. Deepseek coder 1.3b instruct model. `https://huggingface.co/deepseek-ai/deepseek-coder-1.3b-instruct`, 2024. Accessed May 2025.

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

[6] 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.

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

[8] 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.

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

[10] Z. Li, Y. Zhao, Y. Zhao, Y. Liu, Z. Zeng, Y. Liu, and M. Sun. Humaneval+: A more thorough benchmark for evaluating the generalizability of code generation models. *arXiv preprint arXiv:2307.13859*, 2023.

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

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

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

[14] OpenBMB. Codereval benchmark. `https://github.com/CoderEval/CoderEval`, 2023. Java subset used as base for this study.

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

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

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

[18] 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.

[19] 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.

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

[21] 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.

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