Università della Svizzera italiana

Faculty of Informatics

Bachelor Thesis

May 28, 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 [12], GitHub Copilot [4], and StarCoder [8] 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 [6, 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 [14], ROUGE [10], 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 [11]. 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. [5] 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+ [9], 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. [7] 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 large language models (LLMs) of relatively small size: **Qwen2.5-Coder-0.5B-Instruct**, **Qwen2.5-Coder-1.5B-Instruct** [15], and **DeepSeek-Coder-1.3B-Instruct** [1]. These models feature approximately 0.5, 1.5, and 1.3 billion trainable parameters respectively, which is three orders of magnitude smaller than the latest foundation models such as GPT-4, estimated to exceed one trillion parameters. We chose these smaller models to balance performance and computational feasibility, allowing us to run multiple evaluations on local academic hardware without relying on large-scale infrastructure. Although computational cost was not a formal constraint of the study, using smaller models made experimentation more feasible and reproducible in standard academic settings.

All three models are optimized for code-related tasks and released with instruction-following tuning, meaning they have been fine-tuned to generate appropriate responses to structured natural language instructions. This makes them well-suited for prompt-based evaluation tasks like the one in this study.

The term *"Instruct"* indicates that the model has undergone additional fine-tuning on datasets that include natural language prompts and desired completions, making it more responsive to task-specific queries. The Qwen models are part of Alibaba's Qwen2.5 series, designed for code tasks and instruction-following scenarios. Qwen2.5-Coder models were trained on a large mixture of public and synthetic datasets, including a multilingual corpus of code, technical Q&A, and structured programming tutorials [15]. In particular, Qwen2.5-Coder-1.5B was optimized for multi-task code understanding and generation, and supports fine-grained control over function-level behavior through instruction tuning.

DeepSeek-Coder is a family of decoder-only transformer models specifically built for programming-related tasks [1]. The DeepSeek-Coder-1.3B variant used in our study was pre-trained on a massive 2 trillion token corpus that includes 87.5% code and 12.5% natural language. It was then instruction-tuned on 2 billion tokens of high-quality problem–solution pairs and user queries. The model supports over 80 programming languages and is explicitly optimized for tasks such as code completion, explanation, and functional verification, making it suitable for prompt-based correctness evaluation.

All models were accessed via the Hugging Face Hub[1], an open-source platform that hosts pre-trained machine learning models and provides tools for deployment and evaluation. We used the Transformers library to run the models, which handled prompt tokenization, model loading, and inference (i.e., generating predictions) through a standardized API.

3.2 Context: Dataset

We based our experiments on the Java subset of the CODEREVAL benchmark [13], a code generation dataset containing 230 manually curated Java programming tasks. Each task includes: (i) a natural language specification describing what the function is expected to do, (ii) a reference implementation showcasing a possible correct solution, and (iii) an associated test suite for automated correctness verification. The output of each test suite is a boolean: `pass` indicates that a candidate satisfies all test cases, while `fail` indicates that it fails at least one.

Before adopting all 230 tasks, we performed a thorough quality assurance procedure to ensure the reliability of the test suites. This step was crucial, as test misbehavior could bias our evaluation—for instance, by incorrectly labeling an actually wrong solution as correct.

We discarded 49 problems during quality control. Specifically, 20 tasks had reference implementations that failed their own test suites, indicating internal inconsistencies. Another 9 tasks had such weak test coverage that even

[1] `https://huggingface.co`

empty functions passed all test cases. Additionally, 17 tasks were accepted by trivial dummy implementations, such as `return null;` or `return 0;`, which revealed a lack of discriminative power in the test suites. Finally, 3 tasks yielded inconsistent results for identical implementations across different evaluation runs. After filtering out these cases, we retained a final set of 181 Java tasks that passed all quality criteria and provided reliable correctness labels inferred directly from test outcomes.

3.3 Data Collection

Although each problem in the dataset includes a natural language docstring, these descriptions are often minimal—typically consisting of a single vague sentence. For example: "Performs a right node rotation.", "Skips bytes until the end of the current line.", or "Swap values at indexes i and j in arr."

Since our goal is to evaluate the extent to which the quality and depth of natural language documentation affect the ability of LLMs to judge the correctness of a given Java function, the one-line documentation featured in the CODEREVAL dataset is insufficient to answer this question. For this reason, we augmented the description of each code generation problem with a five-layer natural language description covering different aspects of code documentation, such as parameter explanations and examples of input–expected output pairs.

3.3.1 Augmenting the Dataset

Our goal is to study the extent to which the natural language documentation affect the ability of LLMs to judge the correctness of Java functions. To enable this evaluation, we enriched the CODEREVAL dataset with additional structured documentation, since the original format was designed for code generation, not understanding.

We used GPT-4o to produce this documentation because it was both scalable and accurate. Manual annotation of all 181 Java functions across multiple levels would have been impractical, and GPT-4o consistently followed structured formatting and stylistic constraints when prompted correctly.

Each prompt to GPT-4o included the full Java function body and its name. The model was asked to return a five-part structured description covering different semantic aspects of the function:

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 outputs followed a fixed format and were manually verified for quality and correctness. We ensured that L1 remained high-level, L2 captured the full logic and edge cases, and L3–L5 remained accurate, relevant, and specific to each function.

This enrichment gave us modular "building blocks" that could be flexibly combined into prompts with different levels of information. For example, combining L1 and L2 produces a compact explanation of the function's purpose and behavior, while L1 and L3 give purpose plus interface-level understanding without relying on examples or full logic. This modularity was essential for constructing the incremental and layer-removal settings used in our evaluation.

To make this enrichment structure more concrete, we present a full example based on one of the tasks from the dataset.

Listing 1. Reference implementation of `trimArrayElements`

```
public static String[] trimArrayElements(String[] array){
  if (Objects.isEmpty(array)) {
    return new String[0];
```

```
  }
  String[] result = new String[array.length];
  for (int i = 0; i < array.length; i++) {
    String element = array[i];
    result[i] = (element != null ? element.trim() : null);
  }
  return result;
}
```

The original CODEREVAL docstring was:

```
/*
Trim the elements of the given String array, calling String.trim() on each of them.
@param array the original String array
@return the resulting array (of the same size) with trimmed elements
* /
```

Below is the five-layer enrichment produced using GPT-4o:

L1 – Summary: Trims whitespace from each element of an array.

L2 – Behavioral description: Trims leading and trailing whitespace from each string in the input array. If the array is null, returns an empty array. If an element is null, it remains null in the output.

L3 – Signature explanation:

```
@param array String[]:  The input array of strings to be trimmed.
@return String[]:  A new array with trimmed strings, or nulls where applicable.
```

L4 – Examples:

`[" hello ", " world "] → ["hello", "world"]`	(trims spaces)
`["foo", "bar"] → ["foo", "bar"]`	(no trimming needed)
`null → []`	(handles null input)

L5 – Preconditions and postconditions: Input may be null or contain null elements. Output array has the same length as input; each non-null string is trimmed, nulls are preserved.

3.3.2 Building Candidate Pairs

To evaluate whether LLMs can judge the functional correctness of code based on natural language descriptions, we paired each of the 181 Java problems with two candidate implementations: one correct and one incorrect. These implementations were not drawn from CODEREVAL itself, but were generated by us using several large language models, including DeepSeek-Coder-6.7B [1] and Qwen2.5-Coder-7B [15]. We generated multiple candidate solutions for each problem and compiled them into a custom knowledge base of over 5,000 Java implementations.[2]

We applied a filtering process to remove trivial, duplicate, or unusable code. First, we discarded placeholder code such as `/* implementation goes here */` or empty function bodies, which clearly lacked any executable logic. Second, we excluded logic-less shells like `return;` or `return 0;` that performed no computation and could not plausibly be evaluated for correctness. Third, we removed print-only stubs such as `System.out.println("TODO");`, as these were clearly placeholders and not real attempts at implementation. Fourth, we eliminated duplicate implementations to ensure that each candidate pair provided a distinct comparison case. Finally, we removed candidates that relied on methods, classes, or functionality not defined within the task scope. For example, some generated implementations contained comments such as:

```
// Assuming there's a getDefaultValue method for the class
// Assuming IsomorphicGraphMapping is a custom class
// Update heights (assuming that updateHeight() is defined)
```

Such phrases indicate that the code presumes the existence of external dependencies that are not guaranteed to be available within the function's context. Including these candidates would introduce ambiguity, as correctness could no longer be determined purely based on the prompt and provided code. We therefore excluded all such cases to

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

maintain a closed-world evaluation setup, where all necessary information is explicitly present.

After filtering, we finalized the dataset by selecting one correct and one incorrect implementation per problem. When no valid correct candidate was found, we fell back to using the original reference implementation from CODEREVAL. In two exceptional cases where no usable incorrect candidates were available, we manually inserted a faulty variant into the dataset to complete the correct–incorrect pairing.

3.4 Prompting and Evaluation Setup

Each model was tasked with a binary classification problem: given a natural language description, a function signature, and a candidate implementation, it must decide whether the implementation correctly satisfies the described behavior. The model outputs `1` if the implementation is correct, and `0` otherwise. The exact prompt given to the models was formatted as follows, using placeholders to insert task-specific information:

```
You will be provided with the description ("Description") and the signature ("Signature")
of a Java function to implement.  You will also see a candidate implementation ("Candidate").
Your role is to evaluate the correctness of the Candidate, providing as output either 0
or 1, and no other text:

0.  Wrong Implementation:  The implementation does not correctly implement the described
function.
1.  Correct Implementation:  The implementation correctly implements the described function.

# Description:
- Summary:  {L1:  one-line summary}
- Behavior:  {L2:  behavioral explanation}
- Signature Description:  {L3:  @param/@return tags}
- Examples:  {L4:  input-output pairs with notes}
- Preconditions and Postconditions:  {L5:  logical constraints}

# Signature:
{First line of candidate code}

# Candidate:
{Full candidate implementation}

# Output:
```

All prompts were encoded and passed to the models using Hugging Face Transformers with temperature set to 0.2. This low temperature was chosen to reduce randomness, since we could not afford to run multiple evaluations on the same instance. A limit of `max_new_tokens = 20` was enforced to constrain output length and prevent verbose completions, as the task only required a single-digit response (0 or 1). Model outputs were parsed to extract a binary classification, and any non-conforming output was treated as invalid and recorded as `None`.

We evaluated model performance under two experimental settings. In both, the baseline was the minimal prompt containing only the L1 summary layer.

In the first setting, prompt enrichment, we incrementally increased the amount of documentation information provided to the model. Starting from L1 alone, we progressively added layers in the following order: L2 (behavioral explanation), L3 (signature-level tags), L4 (input–output examples), and finally L5 (logical constraints). This yielded five prompt configurations per model, corresponding to L1, L1+L2, L1+L2+L3, L1+L2+L3+L4, and the full L1–L5 stack. We applied this setup to three LLMs—Qwen-0.5B, Qwen-1.5B, and DeepSeek-1.3B—resulting in 15 enrichment experiments in total.

In the second setting, layer removal, we started from the full prompt containing all five documentation layers (L1–L5) and selectively removed individual or combined layers to assess their individual effect on model performance. Rather than exhaustively testing all combinations, we defined a limited set of configurations per model to observe the contribution of each layer. This experimental approach helped isolate which layers had the greatest impact on accuracy and whether some components could be omitted without harming performance. The detailed discussion of results is presented in Section 4.

3.5 Data Analysis

Each model was evaluated on a binary classification task: given a natural language description of a Java function and a candidate implementation, decide whether the implementation correctly satisfies the described behavior. Ground truth correctness labels were drawn from the dataset: each candidate was pre-labeled as either correct or incorrect. Separately, the model was instructed to output `1` if it judged the implementation to be correct, and `0` if it judged it to be incorrect.

Using these two independent sources—predefined correctness labels and model predictions—we computed standard confusion matrix metrics. A true positive (TP) occurs when the model correctly identifies a correct implementation. A true negative (TN) is when the model correctly identifies an incorrect implementation. A false positive (FP) occurs when the model mistakenly labels an incorrect implementation as correct, and a false negative (FN) when the model fails to recognize a correct implementation.

Because our dataset was balanced—containing 181 correct and 181 incorrect implementations—we report accuracy as the primary performance metric. Accuracy reflects the overall proportion of correct classifications (TP and TN) over all evaluated samples:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Model outputs that did not include a valid binary prediction (`0` or `1`) were marked as invalid and excluded from metric calculations.

3.6 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, accuracy plots, confusion matrices, and detailed README instructions for full optional replication. Environment dependencies are specified in `requirements.txt`.

4 Results & Discussion

4.1 Incremental enrichment (L1 → L5)

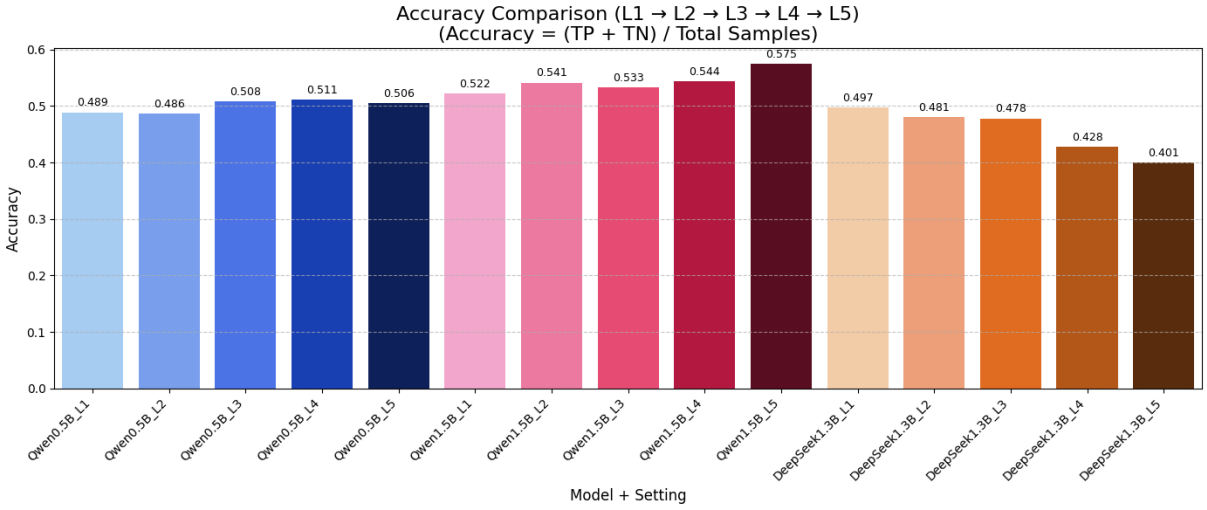

Figure 1. Accuracy per model and documentation level.

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.

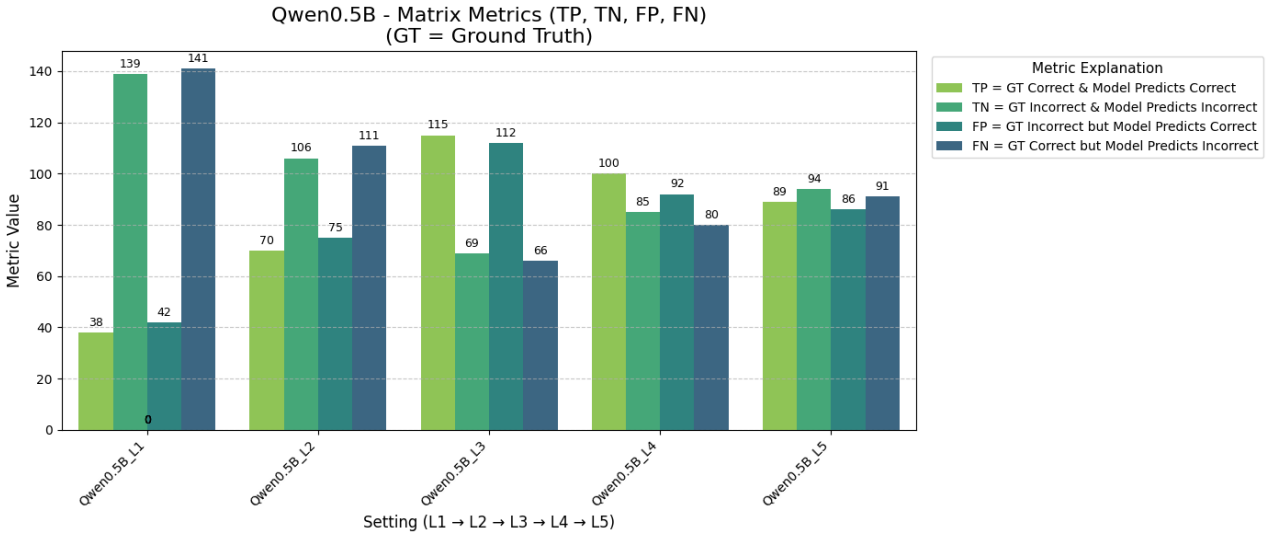

Figure 2. Confusion-matrix metrics — Qwen-0.5B.

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.

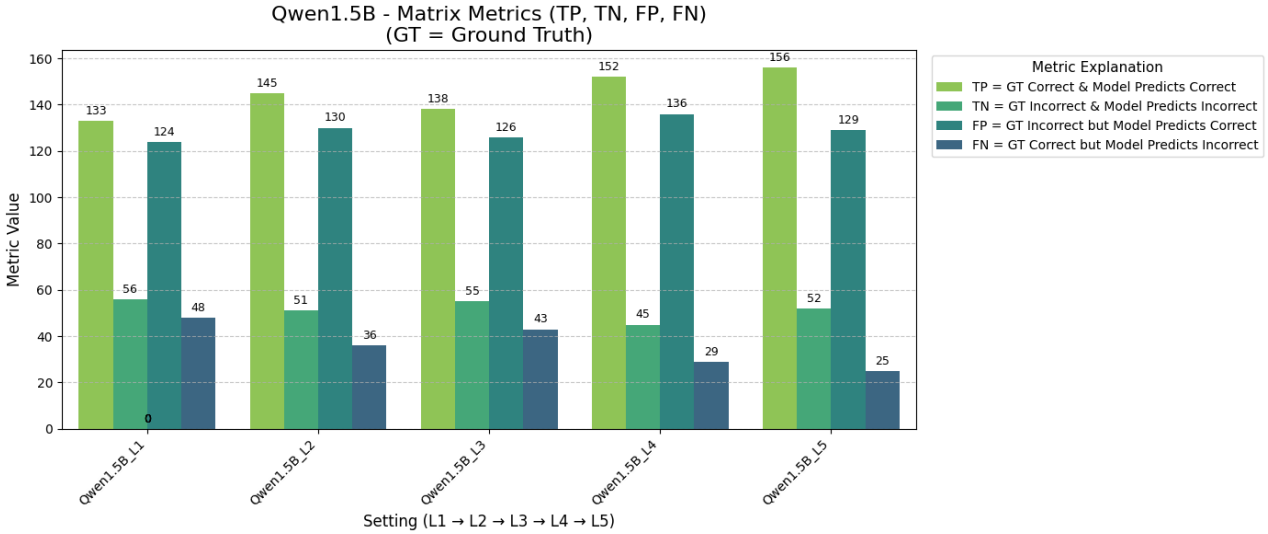

Figure 3. Confusion-matrix metrics — Qwen-1.5B.

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.

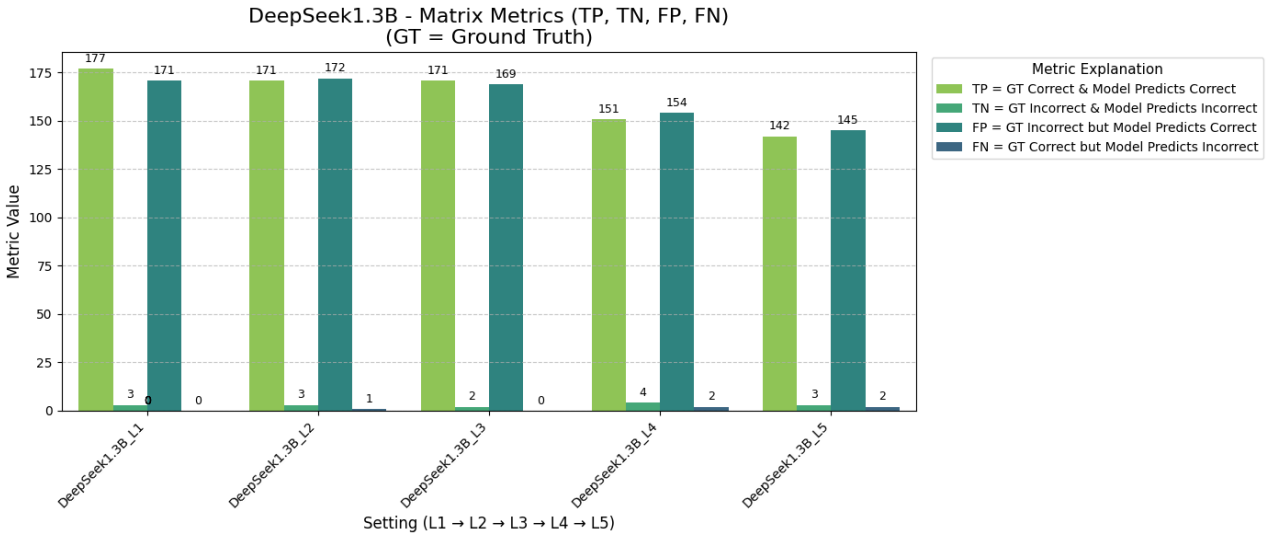

Figure 4. Confusion-matrix metrics — DeepSeek-1.3B.

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] D. AI. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *arXiv preprint arXiv:2401.16670*, 2024.

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

[5] 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.

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

[7] 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.

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

[9] 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.

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

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

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

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

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

[15] Q. Team and A. D. Academy. Qwen2.5: Scaling up language models with data and instruction tuning. *arXiv preprint arXiv:2403.16880*, 2024.

[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.