Evaluation and Benchmarking of Large Language Models for Code Generation Tasks

Prepared to:

Prof. Phuong Thanh Nguyen
Assistant Professor, Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila

Prepared for:

Machine Learning For Software Engineering Course Examination

Prepared by:

Pragati Manandhar
Matriculation number: 297913

Submission Date:

January 19, 2026

Table of Contents

1. Introduction

Large Language Models (LLMs) have demonstrated strong capabilities in program synthesis and code generation. However, evaluating their performance remains a challenging problem. Traditional evaluation approaches rely on **execution-based correctness**, while recent research has explored **LLM-as-a-Judge** paradigms, where a strong language model evaluates generated code based on semantic understanding.

In this project, a **systematic evaluation of multiple open-source LLMs** on Python code generation tasks is performed using two complementary approaches:

1. **Execution-based evaluation** using the HumanEval and HumanEvalNext benchmarks.
2. **LLM-as-a-Judge evaluation**, where a powerful code-focused LLM (Qwen2.5-Coder-7B-Instruct) scores generated solutions of the other models.

The goal is to analyze:

- Code correctness (pass@k),
- Diversity of generated solutions,
- And alignment between execution-based and judge-based evaluation.

2. Datasets

2.1 HumanEval

HumanEval is a widely-used benchmark for evaluating code generation models. It consists of 164 Python programming problems, each with:

- A function signature and docstring prompt
- Hidden unit tests used for execution-based validation

2.2 HumanEvalNext

HumanEvalNext is a more challenging extension of HumanEval that introduces:

- More complex test cases
- Higher sensitivity to edge cases

3. Models Evaluated

A diverse set of 7B-scale language models was evaluated. The evaluated models were categorized into five groups:

- **Base models** (StarCoder2, Mistral), which are primarily pretrained with minimal instruction alignment.
- **General instruction-tuned models** (DeepSeek, Qwen2.5, CodeLlama-Instruct), optimized for following natural language instructions across a broad range of tasks.
- **Python-specialized models** (CodeLlama-Python), trained with a strong emphasis on Python-centric code corpora.
- **Synthetic-instruction models** (Magicoder), which leverage large volumes of automatically generated instruction–response pairs.
- **Industrial models** (aiXcoder) are developed with enterprise-oriented training pipelines and real-world deployment considerations. This categorization enables a comparative analysis of how different models trained with different training strategies influence code generation behavior
- **Judge Model Selection:** For the *LLM-as-a-judge* component, the Qwen-coder-7B model is used, which is widely regarded in research as a robust instruction-following model. Its high instruction density enables it to reliably interpret complex evaluation criteria, provide coherent explanations, and identify logical errors that rule-based or simpler evaluation methods may overlook.

4. Evaluation Methodology

4.1 Execution-Based Evaluation

Execution-based evaluation was performed using the **Hugging Face code_eval metric**, which executes generated code against hidden unit tests.

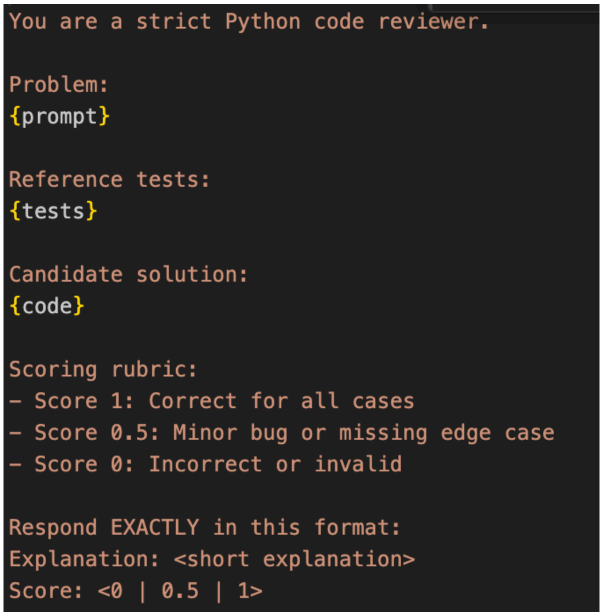
Metrics:

- **pass@5**: Probability that at least one of five generated solutions is correct
- **Unique Solution Rate**: Measures diversity among generated samples

For each problem, five samples were generated using stochastic decoding (temperature = 0.7).

4.2 LLM-as-a-Judge Evaluation

In the second phase, generated solutions were evaluated using **Qwen2.5-Coder-7B-Instruct as a judge model**. The following judge prompt is used as shown in the image below:



```
You are a strict Python code reviewer.

Problem:
{prompt}

Reference tests:
{tests}

Candidate solution:
{code}

Scoring rubric:
- Score 1: Correct for all cases
- Score 0.5: Minor bug or missing edge case
- Score 0: Incorrect or invalid

Respond EXACTLY in this format:
Explanation: <short explanation>
Score: <0 | 0.5 | 1>
```

Figure: Judge Prompt

The scoring rubric is as given below:

- **1.0** – Correct solution for all cases
- **0.5** – Minor bug or missing edge case
- **0.0** – Incorrect, incomplete, or invalid code

For each problem:

- 5 candidate solutions were generated
- The judge model provided:
    - Individual scores
    - Natural language explanations

- Aggregated metrics were computed for all the average scores of the five candidate solutions of each problem:
    - Mean score
    - Max score
    - Diversity score

This approach allows evaluation without executing code, simulating human-like reasoning.

5. Execution-Based Evaluation Results

5.1 Pass@k Performance

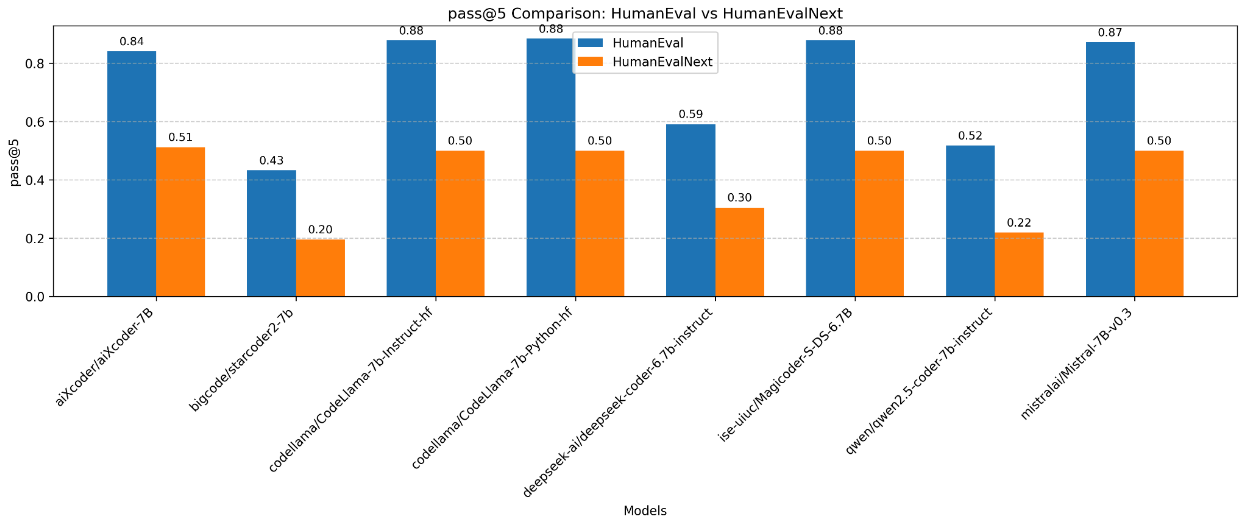


Figure 2: Pass@5 Results

Observation:

- CodeLlama and Magicoder achieve the highest execution correctness.
- Performance drops significantly on **HumanEvalNext**, indicating sensitivity to harder test cases.

5.2 Diversity and BLEU

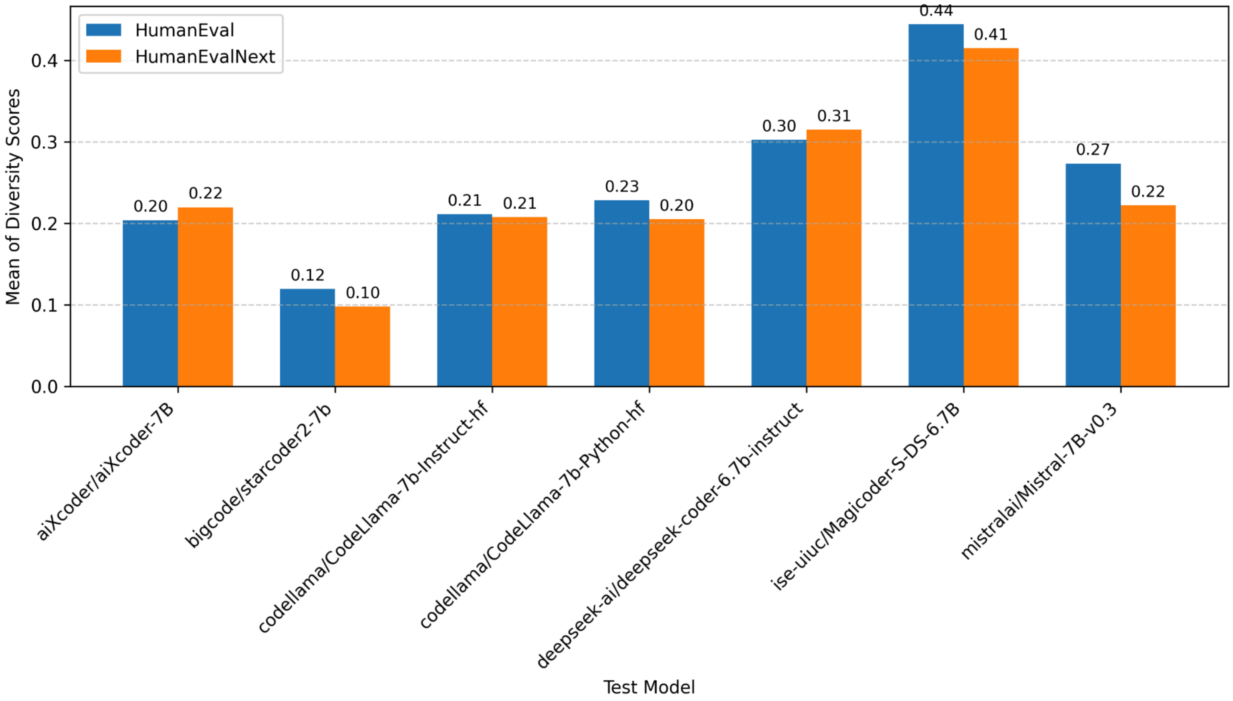


Figure 3: Diversity Results

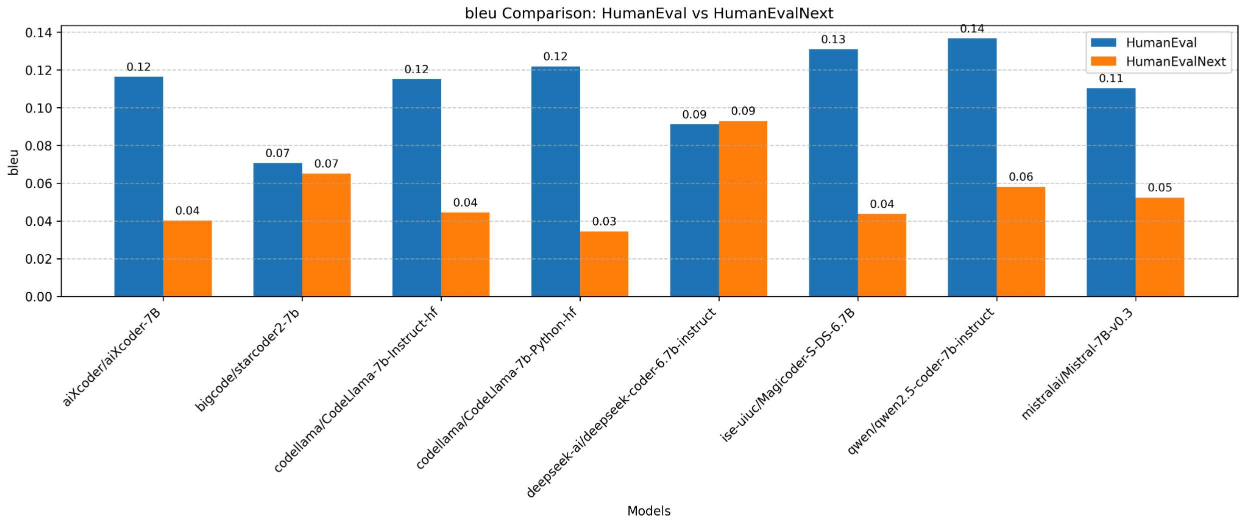


Figure 4: Blue comparison

- **BLEU scores are consistently low**, confirming that:
  - Correct solutions may be structurally different
  - BLEU is poorly correlated with functional correctness

6. LLM-as-a-Judge Results

6.1 Mean Judge Scores

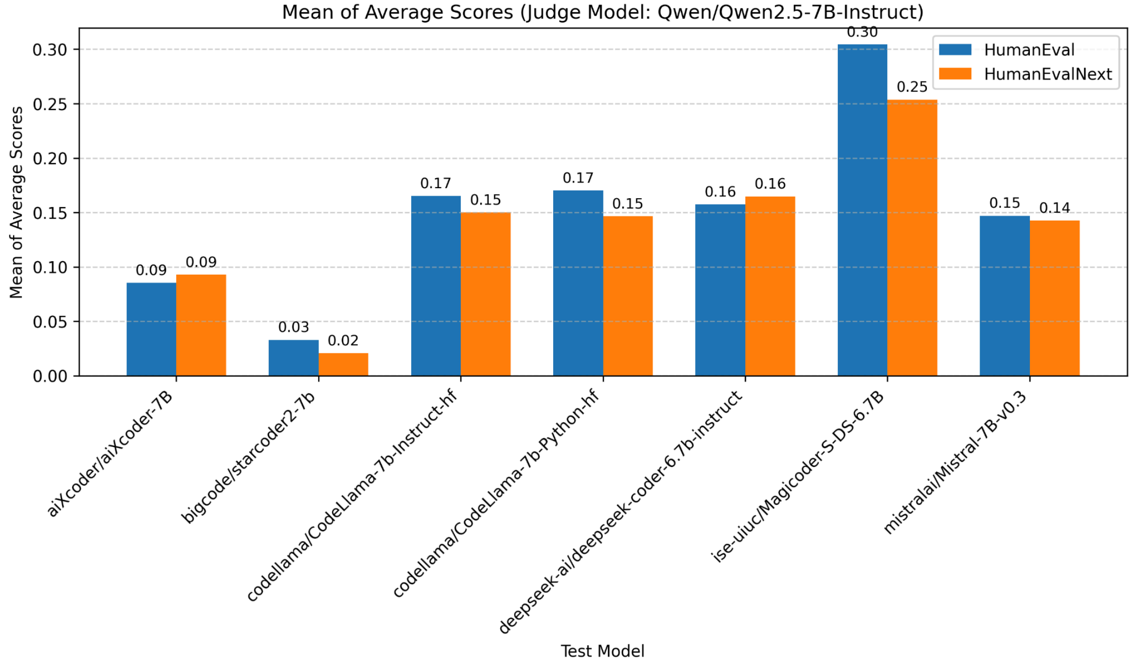


Figure 5: Mean Average Scores

Observation:
Judge scores are much lower in absolute value compared to pass@k, indicating:

- Stricter semantic evaluation
- Penalization of partial correctness and edge case failures

6.3 Percentage of Maximum Scores

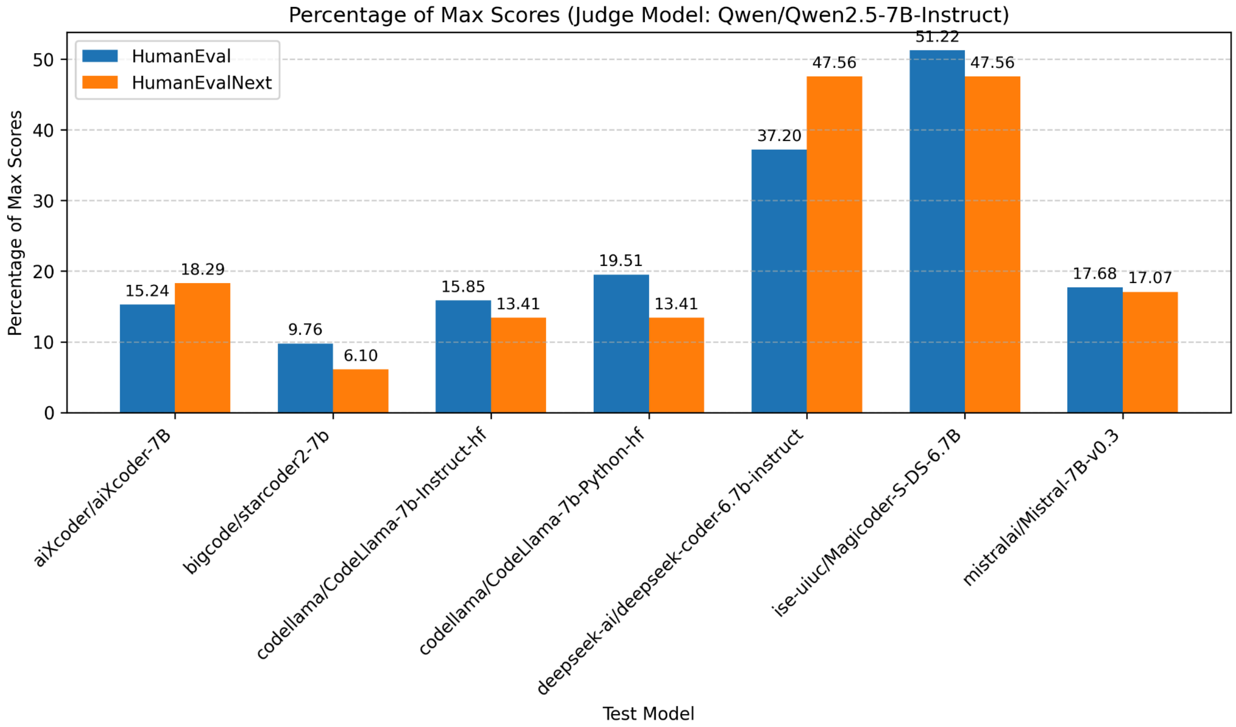


Figure 6: Percentage of max scores

7. Analysis and Discussion

This section analyzes the observed trends across execution-based and LLM-as-a-judge evaluations, highlighting strengths, weaknesses, and consistency between metrics.

7.1 Execution-Based vs. Judge-Based Performance

Across both HumanEval and HumanEvalNext benchmarks, code-specialized and instruction-tuned models consistently outperformed general-purpose models. Magicoder-S-DS-6.7B and CodeLlama variants achieved the highest pass@5 scores, indicating strong functional correctness when multiple samples are generated. However, absolute performance dropped noticeably on HumanEvalNext, demonstrating increased sensitivity to edge cases and reduced reliance on memorization.

Judge-based evaluation produced significantly lower average scores compared to pass@5 results. This difference reflects the stricter semantic evaluation of the LLM-as-a-judge, which penalizes partial correctness, missing edge cases, and logical flaws that may still pass limited unit tests. Despite the difference in scale, the relative ranking of models remained largely consistent across both evaluation paradigms, suggesting alignment between execution-based correctness and semantic reasoning.

7.2 Impact of Model Specialization

Models trained explicitly on code data, particularly Magicoder and CodeLlama, showed superior robustness across both benchmarks. These models not only achieved higher correctness but also maintained higher maximum judge scores, indicating their ability to produce at least one fully correct solution per problem. In contrast, general-purpose models such as Mistral-7B and aiXcoder-7B lagged, especially on HumanEvalNext, highlighting the importance of domain-specific training for reliable code generation.

7.3 Diversity, BLEU, and Correctness Relationship

Diversity scores were generally high across most models, confirming that stochastic sampling produces varied solution strategies. However, high diversity did not necessarily correlate with correctness. Some weaker models generated diverse but largely incorrect solutions. Conversely, stronger models achieved both high diversity and correctness, suggesting deeper problem understanding.

BLEU scores remained consistently low across all models, reinforcing that textual similarity is poorly correlated with functional correctness in code generation. Correct solutions often differ significantly in structure while maintaining equivalent behavior, limiting BLEU's usefulness as a primary evaluation metric.

7.4: Key Takeaways

The combined use of execution-based metrics and LLM-as-a-judge evaluation provides a more comprehensive assessment of code generation quality. While execution-based evaluation remains the gold standard for functional correctness, judge-based scoring adds interpretability, partial credit, and semantic insight, particularly in scenarios where unit tests are unavailable or incomplete.

8. Limitations

Despite its contributions, this study has several limitations. First, the LLM-as-a-judge framework relies on a single judge model, which may introduce bias toward the judge's own training distribution or coding style preferences. Second, judge-based scoring is computationally expensive, as it requires running an additional large model for every generated candidate. Third, although care was taken to standardize prompts and scoring rubrics, natural language explanations generated by the judge model may occasionally be inconsistent or hallucinated. Finally, judge-based scores are not directly comparable to pass@k metrics, which limits their interpretability as absolute performance measures.

9. Future Work

Future work could extend this study in several directions. One promising direction is the use of multiple judge models or ensemble-based judging to reduce single-model bias and improve robustness. Another avenue is calibrating judge scores against execution-based metrics to better align semantic judgments with functional correctness. Additionally, scaling the evaluation to larger benchmarks and more recent models could provide deeper insights into emerging trends in code generation. Finally, incorporating human evaluation or hybrid human–LLM judging could further improve reliability and provide richer qualitative feedback on generated code.

10. Conclusion

This project presented a comprehensive evaluation and benchmarking study of multiple open-source large language models for Python code generation. Two complementary evaluation paradigms were explored: execution-based evaluation using established benchmarks (HumanEval and HumanEvalNext) and an LLM-as-a-judge framework leveraging a strong code-oriented model. The execution-based results demonstrated that code-specialized and instruction-tuned models such as Magicoder, CodeLlama, and DeepSeek-Coder consistently achieved higher functional correctness, particularly on the original HumanEval benchmark. Performance degradation across all models on HumanEvalNext highlighted the increased difficulty of the benchmark and underscored limitations in generalization and edge-case handling.

The LLM-as-a-judge evaluation provided additional insights beyond binary correctness. Judge-based scores were generally lower in magnitude, reflecting stricter semantic scrutiny and sensitivity to partial correctness, logical flaws, and missing corner cases. Importantly, the relative ranking of models under judge-based evaluation largely aligned with execution-based results, suggesting that LLM judges can serve as a meaningful proxy for correctness when unit tests are unavailable. Furthermore, diversity analysis revealed that stronger models not only produced more correct solutions but also generated a broader range of distinct solution strategies, indicating deeper problem understanding.

Overall, this work demonstrates that execution-based and judge-based evaluations capture complementary aspects of code generation quality. While execution-based metrics remain the gold standard for correctness, LLM-as-a-judge methods offer interpretability, flexibility, and applicability in scenarios where automated testing is infeasible.

References

[1] R. Li *et al.*, *StarCoder2: The Next Generation of Open Code Large Language Models*, BigCode Project, 2024.

[2] D. Guo *et al.*, *DeepSeek-Coder: Advancing Code Intelligence with Large Language Models*, DeepSeek AI, 2023.

[3] Alibaba Cloud, *Qwen2.5: Large Language Models for Code Generation and Reasoning*, Alibaba Cloud Research, 2024.

[4] Meta AI, *Code Llama: Open Foundation Models for Code*, Meta AI Research, 2023.

[5] Y. Wei *et al.*, *Magicoder: Instruction-Tuned Code Generation with Diverse Synthetic Data*, University of Illinois Urbana–Champaign, 2023.

[6] aiXcoder, *aiXcoder-7B: Large Language Model for Intelligent Programming Assistance*, aiXcoder Research, 2023.

[7] A. Jiang *et al.*, *Mistral 7B*, Mistral AI, 2023.

[8] S. Koohestani *et al.*, "Benchmarking AI Models in Software Engineering," *IEEE Transactions on Software Engineering*, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:282401197

[9] Arize AI, "LLM as a Judge – Best Practices and Basis in Research," Arize.com, 2026. [Online]. Available: https://arize.com

[10] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A Survey on Large Language Models for Code Generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, Jul. 2025, doi: 10.1145/3747588.