CrossPyEval: Enhancing LLM-based Evaluation of Low-Resource Code via Code Translation

Weijie Wu WU-WEIJIE@SJTU.EDU.CN
Ling-I Wu WULIANYI1998@SJTU.EDU.CN
Guoqiang Li* LI.G@SJTU.EDU.CN
Shanghai Jiao Tong University, Shanghai 200240, China

Editors: Hung-yi Lee and Tongliang Liu

Abstract

Large language models (LLMs) have demonstrated remarkable performance in code generation and evaluation tasks, particularly for Python, which dominates the pre-training corpora. However, the evaluation of code in low-resource programming languages remains challenging due to limited data and suboptimal model alignment. In this paper, we propose CrossPyEval, a novel cross-language code evaluation framework that uses an LLM to translate code from other languages into Python, verifies consistency with an SMT solver, and then analyzes the translated code via abstract syntax trees before performing the final evaluation. Experiments on public benchmarks and our custom low-resource datasets demonstrate that CrossPyEval substantially boosts evaluation accuracy for non-Python languages, achieving up to an 8.83% improvement, and significantly enhances alignment with human judgments, with the Kendall correlation rising to as high as 0.689.

Keywords: Code Evaluation; Large Language Models; Code Translation; Low-Resource Programming Languages

1. Introduction

The rapid advancement of large language models (LLMs) has sparked a surge in automated code generation research, ushering in an era where executable programs can be synthesized with minimal human intervention Chowdhery et al. (2023); Nijkamp et al. (2022). This progress has made automatic code generation increasingly viable for industrial-scale applications. As more developers adopt these tools to assist in software development, the evaluation of generated code remains heavily reliant on manual review, creating a bottleneck in the development pipeline. Meanwhile, with the growing popularity of reinforcement learning (RL), a critical challenge has emerged: how to automatically provide robust reward signals for training code generation models. This further highlights the importance of developing reliable and automated code evaluation methods.

Traditional n-gram-based metrics, such as CodeBLEU Ren et al. (2009), focus on syntactic consistency with the reference. However, these metrics often produce false positives for generated code with similar syntax but incorrect execution, and false negatives for code that executes correctly but differs in implementation from the reference. Recent studies have highlighted the significant potential of large language models (LLMs) as autonomous evaluators. Through innovative approaches, such as training-free methods like adversarial testing Liu et al. (2023), self-supervised iteration Zheng et al. (2023a), social simulation Chan

* Corresponding author.

et al. (2023), slow thinking Tong and Zhang (2024) and test-case testing Alshahwan et al. (2024), as well as training-based methods Dong et al. (2025), LLMs have demonstrated evaluation efficacy comparable to human assessments in value alignment tasks.

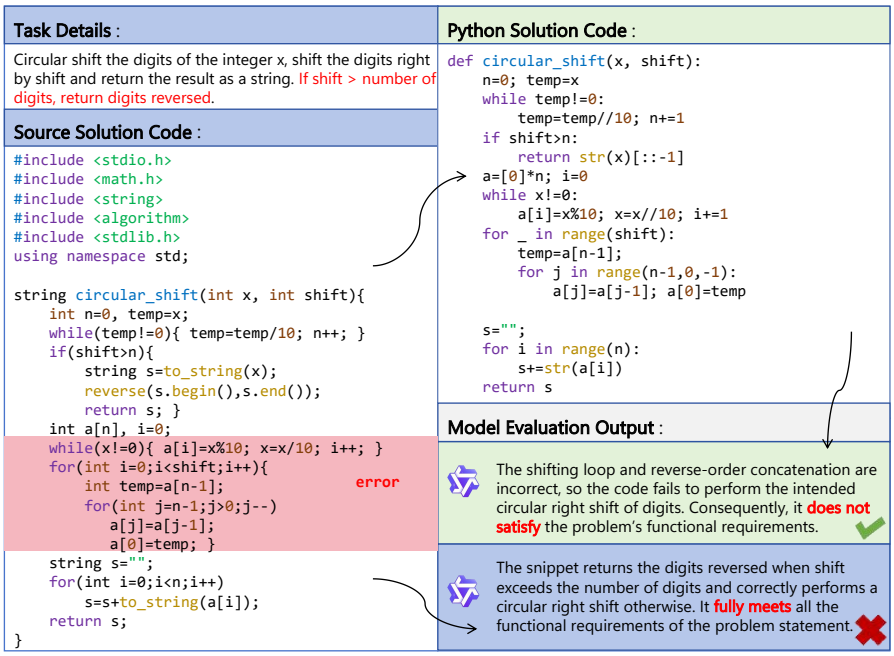


Figure 1: Correcting the Qwen-Coder's Evaluation of a Faulty Cpp Circular Shift Implementation via Python Translation.

Nonetheless, these LLM-based code evaluation approaches face three key limitations: (1) **Insufficient cross-language evaluation**: Syntax and paradigm differences across languages significantly impact scoring quality, with Python and Java often outperforming other languages; (2) **Limited logical reasoning in multi-requirement scenarios**: LLMs struggle to fully capture code logic in complex or deeply branched tasks, leading to degraded evaluation accuracy and missed requirements, resulting in partial assessments; and (3) **Lack of a low-resource code evaluation meta-dataset**: There is a shortage of comprehensive datasets specifically designed to evaluate code in low-resource languages, hindering the effectiveness of LLMs for such tasks.

To address these shortcomings, we propose a novel cross-language code evaluation framework, **CrossPyEval**, designed to enhance the performance of large language models (LLMs) on code evaluation tasks, particularly for low-resource languages. First, to bridge the gap between widely-used languages and low-resource languages, we leverage well-established methods for code translation using large language models (LLMs) to convert diverse source languages into Python without losing semantic meaning, thereby creating a unified evaluation environment. Next, to further enhance the LLM's logical reasoning capabilities in complex, multi-requirement scenarios, we extract path conditions, data-flow dependencies and function calls from Python Abstract Syntax Trees (ASTs). We then annotate key vari-

ables along each execution path using def-use chains, merging path conditions with variable-evolution data to produce a comprehensive logical summary. This summary captures all possible branches and highlights variable transformations, providing rich semantic context for subsequent LLM evaluations. Finally, we construct and release a new multilingual code evaluation dataset covering 810 problems in three low-resource languages (Kotlin, PHP, and Scala), each annotated with LLM-generated solutions and correctness labels, further validating the generalizability and practical value of our method.

Our major contributions can be summarized as follows:

- We propose a novel cross-language code evaluation framework that leverages large language models to translate code from low-resource programming languages into Python, enabling more accurate and robust assessment by utilizing well-established Python-based evaluation tools.
- We empirically demonstrate the effectiveness of our approach on the HumanEval-X benchmark, showing that evaluating translated Python code yields significantly higher accuracy compared to direct evaluation on the original code in various programming languages.
- We construct a low-resource language code evaluation benchmark to further assess the effectiveness of CrossPyEval and address the current lack of a low-resource code evaluation meta-dataset in the community.

We validate our code translation-based evaluation method across multiple public code evaluation benchmarks and self-constructed dataset. Experimental results show that: (i) in cross-language code evaluation scenarios, compared to direct evaluation on the original language code, our method achieves significant improvements, with accuracy gains of up to 8.83% and Kendall correlation reaching as high as 0.689; and (ii) across all three low-resource languages, CrossPyEval also significantly outperforms direct source-language evaluation, achieving its largest accuracy gain of 11% on Kotlin.

2. Related Work

2.1. Match-based Evaluation

Match-based evaluation methods include BLEU Papineni et al. (2002), ROUGE-L Chin-Yew (2004), METEOR Denkowski and Lavie (2014), and others. In the domain of code, specialized metrics such as CodeBLEU Ren et al. (2009) and RUBY Tran et al. (2019) have also been proposed. These match-based metrics mainly capture surface-level similarities and often fail to reflect the functional correctness or semantic equivalence of code, thereby limiting their effectiveness in practical code evaluation scenarios.

2.2. Model-based Evaluation

Model-based evaluation methods assess code quality by computing the similarity between generated and reference code in a semantic space. For example, CodeBERTScore Zhou et al. (2023) is inspired by BERTScore Zhang et al. (2019) from the machine translation

field. Compared with traditional match-based methods, model-based approaches can capture richer semantic information, but their evaluation results still depend on the quality and diversity of the reference code.

2.3. LLM-based Evaluation

In recent years, with the advent of large language models (LLMs), LLM-based code evaluation methods have attracted increasing attention. ICE-Score Zhuo (2023) employs predefined evaluation criteria for multidimensional assessment, CodeScore Dong et al. (2025) introduces a novel paradigm by evaluating code through its execution behavior, and CodeJudge Tong and Zhang (2024) adopts a "slow thinking" mechanism, guiding the LLM through step-by-step analysis before making its judgment.

3. Method

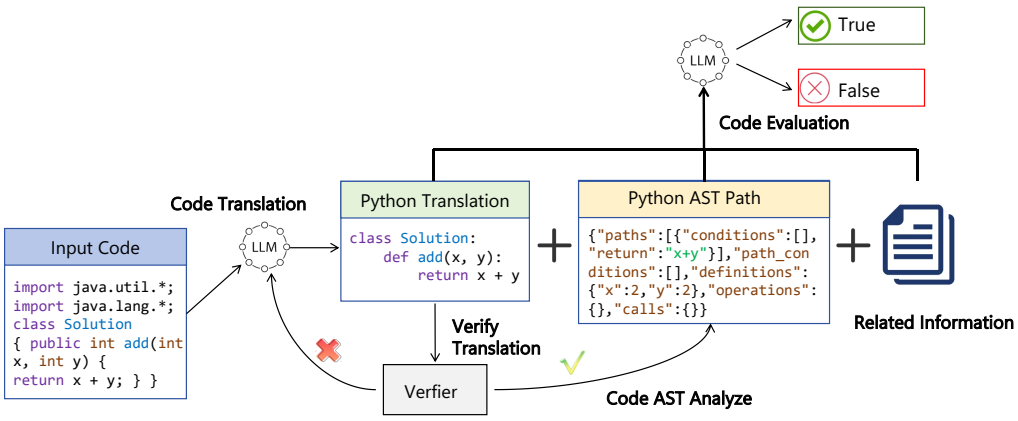


Figure 2: The CrossPyEval Framework Workflow Overview

Figure 2 illustrates the overall workflow of the CrossPyEval evaluation framework proposed in this study. The framework first translates source code written in other programming languages (e.g. Java) into semantically equivalent Python versions. It then verifies functional equivalence and logical consistency using a verifier. Next, it analyzes Python code with abstract syntax trees (AST). Finally, the large language model integrates the problem description, Python code, and analysis results to evaluate the code.

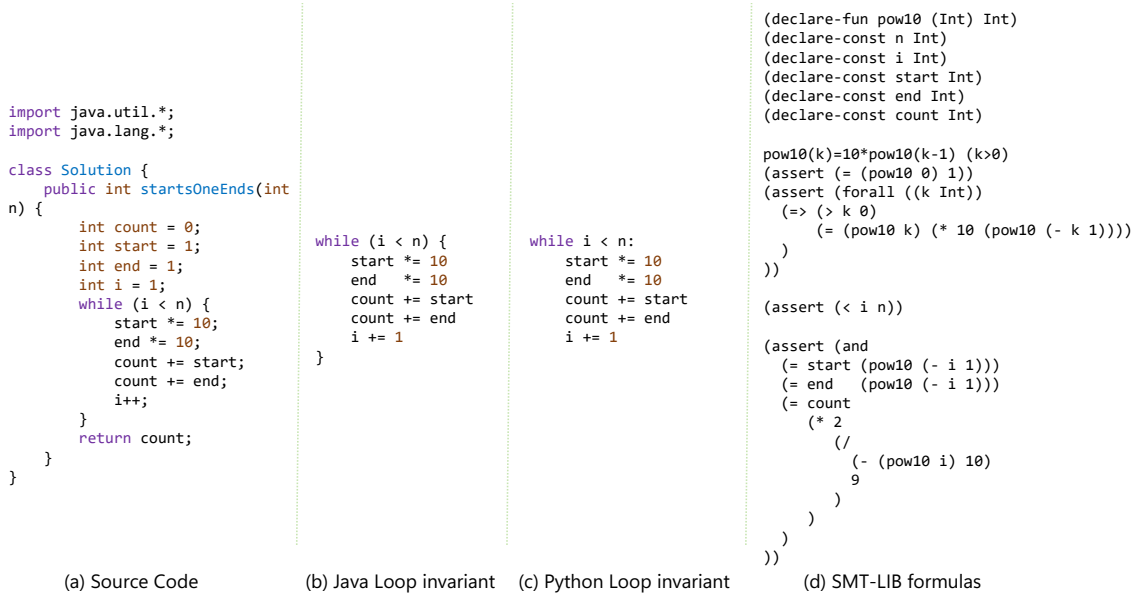
3.1. Cross-Language Python Translation

Previous research Tong and Zhang (2024) has shown that LLM-based evaluators perform significantly better on Python code compared to other programming languages. This advantage is largely due to the abundance of Python training data available during the pretraining phase of LLMs. We believe that, even when the same functionality is expressed,

the LLM's deeper understanding of Python allows it to better simulate code execution. Therefore, instead of asking the LLM to evaluate code written in other languages directly, we first prompt it to translate the code into Python.

The key objective of the code translation stage is to ensure that the translated code behaves identically to the original. One approach is to generate test cases for both versions and compare their execution results. However, this method requires a full runtime environment, which can be resource-intensive. Alternatively, static analysis can be used, but building a robust cross-language verifier is inherently challenging.

Inspired by the framework proposed in "Verified Code Transpilation with LLMs" Bhatia et al. (2024), we argue that invariant-based consistency verification offers a lightweight means to partially verify translation correctness without relying on runtime execution or complex static analysis. Many studies have pointed out that handling loops remains a frequent source of errors for LLMs. As code scenarios grow increasingly complex, loops also become more prevalent. Therefore, we consider invariant verification to be a reasonable lower bound for ensuring translation consistency. It avoids the need for both heavy runtime environments and sophisticated static analysis.



(a) Source Code (b) Java Loop invariant (c) Python Loop invariant (d) SMT-LIB formulas

Figure 3: Original java loop code convert to python code and generate loop invariants, then convert to SMT-LIB formulas.

Specifically, our framework prompts the LLM to extract formal invariants from each function containing loops and convert them into SMT-LIB formulas. An SMT solver is then used to verify semantic entailment between the invariants of the original and translated code. If any discrepancies or missing invariants are detected, the solver provides counterexamples and diagnostic information, which the LLM uses to generate localized fixes in the form of diff patches. The revised code is then subjected to the same extraction and verification process.

Figure 4: Original java loop-free code convert to python code and generate post-state conditions, then convert to SMT-LIB formulas.

As a complementary approach, for functions without loops, we apply Relational Bounded Symbolic Execution (RBSE). In this case, the LLM is prompted to symbolically compute the post-state conditions for each execution path and encode them as SMT-LIB formulas. This technique is not suitable for loop-containing functions, as the symbolic path conditions would need to be extended up to n times, where n is the number of loop iterations, leading to combinatorial explosion. Similar to the loop-invariant pipeline, an SMT solver checks for semantic consistency between the original and translated post-conditions. If inconsistencies arise, counterexamples and diagnostics are fed back to the LLM, which then generates localized fixes. These updates are reprocessed through the same verification loop.

3.2. Python AST analysis

To further enhance the logical reasoning capabilities of LLM-based code evaluation in multi-requirement scenarios, we propose a logic summarization method based on Abstract Syntax Trees (ASTs). By constructing ASTs from source code, our approach generates high-fidelity symbolic representations within a single-language environment (Python).

Specifically, our method performs a depth-first traversal of the AST to accurately capture all execution paths involving control structures such as if/elif/else, for, while, and try/except. Each path is annotated with the corresponding sequence of branch conditions and return expressions, resulting in a comprehensive representation of the program's branching logic, which is crucial for understanding behavioral semantics in complex functions.

At the data flow level, we introduce lexical reverse parsing to precisely decompose assignment and augmented assignment statements. This includes marking variable definition points, retaining operator and operand details, and embedding the evolution history of variables into the path context via def-use chains, thereby enabling fine-grained tracking of state changes throughout the code. In parallel, we traverse function call nodes in the AST to record calling expressions and their frequencies. This allows us to capture external

dependencies, supporting further analyses such as code complexity evaluation and library usage risk assessment.

The resulting multidimensional logic summary unifies control flow, data flow, and call relationships into a structured format, significantly enhancing both the information density and interpretability of the code. This summary is then fed into the LLM to produce the final evaluation output, enabling more informed and context-aware model judgments.

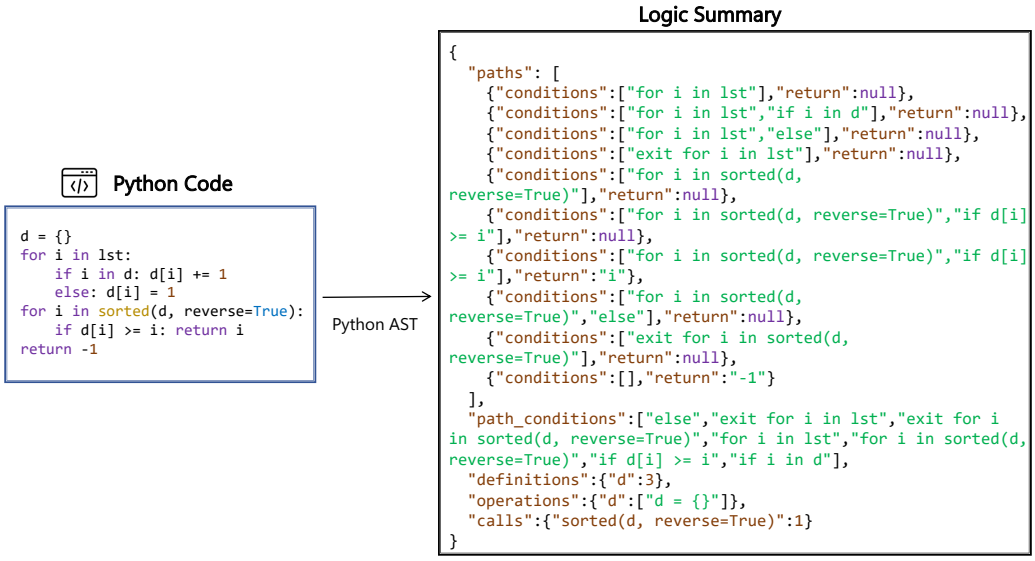


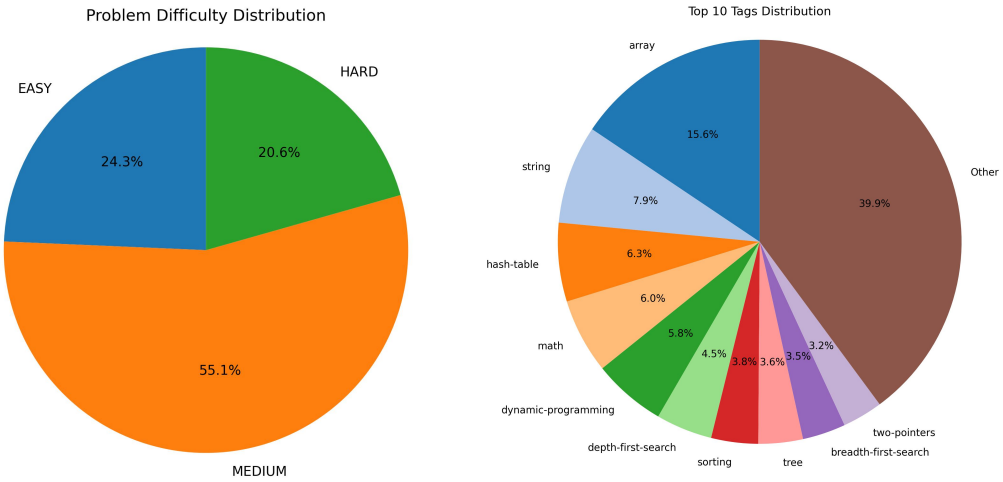
Figure 5: An illustrative example of the AST-based logic summarization method applied to a Python function.

Our approach adopts a unified extraction pipeline that abstracts control structures and data dependencies above the function level into concise, human-readable summaries. This eliminates the need for tedious manual feature engineering and provides a plug-and-play input format for downstream LLM tasks.

4. Experiments Setup

4.1. Self-constructed Dataset: LeetCodeEval-LR

To address the lack of meta-datasets for low-resource programming languages and further validate the effectiveness of our proposed framework, we construct a new benchmark derived from the LeetCode platform. We randomly sampled 810 problems across three low-resource languages: Kotlin, PHP, and Scala, and used the CodeLlama-13B-Instruct model to generate solutions based on the provided problem descriptions and predefined function signatures. The generated code was then uploaded back to the LeetCode platform, where we leveraged its built-in test cases to evaluate and manually label the correctness of each solution. Figure 6 presents the summary statistics of our benchmark, LeetCodeEval-LR.



(a) Problem Difficulty Distribution

(b) Algorithm Types Distribution

Figure 6: (a) and (b) together provide a comprehensive overview of the LeetCodeEval-LR benchmark. Specifically, (a) illustrates the relative proportions of Easy, Medium, and Hard problems, giving insight into the overall difficulty balance of the dataset; while (b) breaks down the types of algorithms tested—such as Sorting, Dynamic Programming, Graph Traversal, and others—highlighting the benchmark's coverage of core algorithmic topics.

4.2. Public Datasets

HumanEval-X Zheng et al. (2023b), is a multilingual extension of the classic HumanEval benchmark, specifically designed to assess code-generation capabilities across different programming languages. The original HumanEval dataset, released by OpenAI in their research on Codex Chen et al. (2021), comprises a series of natural language task descriptions, human-authored references and test cases. In our experiments, five widely-used programming languages—Java, C++, JavaScript, Go and Python are selected for evaluation.

BigCodeBench Zhuo et al. (2024), is a benchmark dataset for evaluating Python code generation. It comprises a total of 1,140 moderately complex programming tasks, which are designed to closely reflect real-world programming scenarios. These tasks involve multi-library interactions, API calls, and function compositions, aiming to assess the capability of large language models (LLMs) to invoke and effectively integrate multiple library functions.

APPS Hendrycks et al. (2021), is a benchmark designed for evaluating Python code generation. It features a diverse set of programming tasks with progressively increasing difficulty, including entry-level, interview-level, and competition-level problems. These tasks are collected from various mainstream programming competition platforms, providing realistic and challenging scenarios to assess the code generation capabilities of models. In this study, we adopt 100 high-difficulty tasks from the dataset.

4.3. Evaluation Metrics

We employ the standard statistical correlation metric Kendall's Tau (τ) Kendall (1938) to quantify the consistency between our method's scores and human judgments, a measure that has been widely used in recent studies. Meanwhile, we treat the large model's judgment of code correctness as a binary classification task and use accuracy as the primary evaluation metric. Since our dataset has a balanced distribution of positive and negative samples, accuracy provides an intuitive and effective measure of the model's overall performance.

4.4. Baselines

We adopt the evaluation methods introduced in Section 2, which are divided into two categories: six token-based metrics (BLEU, ROUGE-L, METEOR, CodeBLEU, chrF and RUBY), the model-based metric (CodeBERT) and LLM-based metric CodeJudge.

4.5. Implementation details

To demonstrate the effectiveness of our method across different models and analyze how model performance affects this approach, we deploy the instruction-tuned code models CodeLlama-13B-Instruct and Qwen2.5-Coder-14B-Instruct, as well as the general-purpose language model Llama3.1-Instruct-8B, on a local inference cluster equipped with two NVIDIA RTX 4090-24 GB GPUs. To assess the impact of stochasticity during inference, we set the temperature to 0.4 and the nucleus sampling parameter top_p to 0.9. Each experimental configuration is run three times, and the results are averaged to produce the final score. All models employ an identical prompt template to ensure comparability and reproducibility.

5. Result

We validate the effectiveness of the proposed CrossPyEval evaluation framework through extensive experiments. First, we report experimental results on four commonly used programming languages in HumanEval-X. Then, we further evaluate the performance of the Python AST method on the HumanEval, BigCodeBench, and APPS datasets. Finally, we demonstrate that the framework achieves consistent results on our self-constructed low-resource test set, in line with previous experiments. In addition, we conduct ablation studies to verify the effectiveness of the Python AST method for cross-language code evaluation.

Results on Cross-Language Scenarios. Table 1 presents the evaluation results of our CrossPyEval framework compared with existing methods and the CODEJUDGE baseline across four popular programming languages (C++, Java, JavaScript, and Go) on the HumanEval-X dataset. From the results, we can draw the following conclusions: (1)CrossPyEval outperforms existing automatic metrics and CODEJUDGE across most languages and models. For example, when using the Qwen2.5-Coder-14b model, CrossPyEval achieves the highest Kendall's τ scores of 0.689, 0.648, and 0.675 on C++, Java, and JavaScript respectively, significantly surpassing baseline. This demonstrates the superior ability of our framework to capture semantic correctness in cross-language code evaluation. (2)The advantage of CrossPyEval is more pronounced with stronger LLMs. While improvements over CODEJUDGE are visible for CodeLlama-Instruct-13b and Llama3.1-Instruct-8b, the gains become substantial with Qwen2.5-Coder-14b, indicating that the framework

Table 1: The Kendall (τ) correlations and Accuracy (%) of different models with semantic correctness on HumanEval-X in four languages. "—" for accuracy indicates that traditional n-gram or embedding-based metrics do not produce binary scores and thus accuracy is not applicable.

Metric	C++		Java		JavaScript		Go	
	τ	*Acc*	τ	*Acc*	τ	*Acc*	τ	*Acc*
Existing Methods								
BLEU	0.306	—	0.230	—	0.288	—	0.261	—
ROUGE-L	0.305	—	0.249	—	0.329	—	0.260	—
METEOR	0.338	—	0.299	—	0.379	—	0.284	—
CodeBLEU	0.341	—	0.318	—	0.384	—	0.268	—
chrf	0.314	—	0.267	—	0.368	—	0.242	—
RUBY	0.284	—	0.260	—	0.329	—	0.245	—
CodeBERTScore f1	0.334	—	0.282	—	0.318	—	0.308	—
CodeBERTScore f3	0.375	—	0.303	—	0.363	—	0.324	—
CodeLlama-Instruct-13b								
CODEJUDGE	0.311	63.13	0.296	56.82	0.285	59.60	0.294	59.35
CrossPyEval	**0.373**	**65.41**	**0.325**	**62.62**	**0.407**	**64.14**	0.259	**59.60**
- Python Ast	0.276	62.63	0.306	59.09	0.284	62.37	0.227	58.84
Llama3.1-Instruct-8b								
CODEJUDGE	0.188	58.34	0.304	61.36	0.304	61.36	0.204	51.77
CrossPyEval	**0.373**	**67.17**	**0.337**	**63.40**	**0.340**	**65.15**	**0.301**	**60.01**
- Python Ast	0.287	61.36	0.314	60.60	0.285	62.88	0.277	58.33
Qwen2.5-Coder-14b								
CODEJUDGE	0.581	79.30	0.610	81.57	0.605	80.05	0.620	80.30
CrossPyEval	**0.689**	**85.10**	**0.648**	**82.33**	**0.675**	**83.33**	0.557	**82.83**
- Python Ast	0.588	80.30	0.625	81.82	0.650	82.58	0.578	79.55

effectively leverages the reasoning capabilities of larger, more capable models. (3)Incorporating the Python AST analysis module consistently improves evaluation accuracy. The ablation results show a clear drop in accuracy when the AST-based logic summarization is removed, across all languages and models. This confirms that the AST analysis enriches the model's understanding of program structure and control/data flow, leading to more precise assessments. (4)Performance varies across languages, reflecting inherent language characteristics and dataset complexity. For instance, Go shows relatively lower Kendall's τ scores compared to other languages, which may be due to its syntactic and semantic differences or fewer training examples in the LLM pretraining data. Nevertheless, CrossPyEval still achieves competitive accuracy on Go, demonstrating its generalizability.

Results on Multi-Requirement Scenarios. To support this, we further analyzed the number of loops present in several widely-used public benchmarks. The results, shown in Table 5, indicate that loops are commonly distributed across these benchmarks.

Table 2: Average number of loops per case in each benchmark

	HumanEval	**BigCodeBench**	**APPS**
Average Loops per Case	0.488	0.491	2.485

To validate the effectiveness of our AST-based logic summarization method in multi-requirement scenarios, we compared our approach with the mainstream evaluation baseline CodeJudge on three representative Python benchmarks: HumanEval, BigCodeBench, and APPS. Table 3 summarizes the experimental results across different large language models (LLMs). It can be observed that our method achieves comparable or better performance than CodeJudge on all benchmarks, with the most significant improvement observed on BigCodeBench. For example, with the Qwen2.5-Coder-14B-Instruct model, our method outperforms CodeJudge by 7.3 percentage points on BigCodeBench, while the improvement on APPS is relatively limited.

Table 3: Performance of Different Large Language Models on Various Python Benchmarks Using AST Analysis

Method	HumanEval	BigCodeBench	APPS
CodeLlama-13B-Instruct			
CodeJudge	72.22	51.17	53.33
Ours	70.71	**54.97**	**55.67**
Llama3.1-8B-Instruct			
CodeJudge	70.96	52.05	44.00
Ours	**71.59**	50.88	43.33
Qwen2.5-Coder-14B-Instruct			
CodeJudge	87.88	64.92	68.00
Ours	**88.64**	**72.22**	66.00

We believe the main reason for this result is that BigCodeBench focuses on practical, moderately complex programming tasks that emphasize multi-library function calls, API integration, and function composition—scenarios that closely reflect real-world software development. Although these tasks are less dense in loops compared to APPS, they require a stronger grasp of global logic, data flow, and function call relationships. Our AST-based logic summarization method systematically extracts multidimensional information such as control flow, data flow, and function calls, providing the LLM with richer context and reasoning cues. As a result, our method performs better in scenarios like BigCodeBench, which emphasize multi-module integration and complex dependencies.

In contrast, although the APPS dataset features more complex loop structures, its problems are mostly algorithmic tasks from programming competitions, where the main challenge for models lies in the correctness of algorithm implementation and handling of edge cases. Since the high-difficulty problems in APPS often involve extreme inputs, special boundaries, and deeply nested loops, relying solely on AST summarization and static analysis cannot fully cover all potential errors, thus limiting the overall improvement.

Additionally, the HumanEval dataset mainly consists of basic programming tasks with relatively simple control flow and dependencies. Therefore, both CodeJudge and our method achieve high and similar accuracy on this benchmark.

Results on Low-Resource Languages Scenarios. To further evaluate the generalizability and robustness of our CrossPyEval framework in low-resource language settings, we conduct experiments on the self-constructed LeetCodeEval-LR benchmark. Table 4 reports the accuracy of various models and evaluation methods on this dataset.

Table 4: Accuracy (%) of Various Models on the Self-constructed Dataset LeetCodeEval-LR

Method	Kotlin	PHP	Scala
CodeLlama-13B-Instruct			
CodeJudge	45.00	54.33	41.67
Ours	**56.00**	**56.00**	**46.00**
- Python Ast	51.00	52.00	44.33
Llama3.1-8B-Instruct			
CodeJudge	58.00	50.00	45.67
Ours	**61.00**	58.67	**47.33**
- Python Ast	57.00	**59.00**	46.00
Qwen2.5-Coder-14B-Instruct			
CodeJudge	67.33	67.67	63.00
Ours	**70.00**	**73.00**	**65.67**
- Python Ast	66.67	72.00	63.66

Experimental results on three programming languages and multiple LLMs show that our method consistently outperforms the baseline. For example, on the CodeLlama-13B-Instruct model, our method improves accuracy over CodeJudge by 11.00 and 4.33 percentage points on Kotlin and Scala, respectively. Similar trends are observed with other LLM models, where our method achieves the highest accuracy in Kotlin, PHP, and Scala. Notably, the ablation study demonstrates that the Python AST-based logic summarization module contributes a significant portion of the performance gain, highlighting its effectiveness in enhancing code reasoning and evaluation, even in low-resource language scenarios.

The superior performance of our framework can be attributed to several key factors. First, by translating code from low-resource languages into Python, we effectively alleviate the data scarcity and domain shift issues that typically hinder LLM-based code evaluation. Second, cross-language verification leverages SMT solvers to perform invariant and post-condition checks, ensuring that the translated Python code preserves the semantics of the original implementation. This reduces the risk of evaluation errors caused by translation artifacts or semantic mismatches. Third, the AST-based logic summarization provides the LLM with a structured and comprehensive view of the program's control flow, data flow, and function call relationships, enabling more accurate assessment of code correctness.

It is also worth noting that, while the overall accuracy on low-resource languages is lower than that on high-resource languages such as Python, the relative improvement brought by our method is substantial. This demonstrates the potential of CrossPyEval as a practical and extensible solution for multilingual code evaluation, especially in scenarios where direct LLM evaluation is unreliable due to limited language-specific training data.

Costs Analysis. Although our method outperforms the baseline in terms of accuracy, it inevitably incurs additional computational and engineering overhead. Specifically, we conducted a comparative experiment on 132 randomly selected samples from the low-resource Kotlin dataset, using CodeLlama-13B as the backbone model. Compared with CodeJudge, our approach increases token consumption by an average of 27.79% and latency by 20.19%. This demonstrates that relying on large language model generation for code translation and verification introduces non-negligible computational and time costs. Nevertheless, we consider these costs acceptable, as the improvement in accuracy provides greater value for

Table 5: Average number of loops per case in each benchmark

Method	Token Costs	Time Costs	Efficiency
Codejudge	1276.47	9.7992	4.592
Ours	1631.37	11.7781	4.755

ensuring fair evaluation in low-resource language scenarios. In future work, we will explore more efficient generation and verification mechanisms to further optimize computational costs while maintaining or even enhancing translation accuracy.

6. Conclusion

In this paper, we introduce CrossPyEval, an LLM-based evaluation framework designed to improve code evaluation accuracy in low-resource languages. CrossPyEval employs code translation and SMT solver verification, using the translated Python code alongside AST-based analysis as the evaluation target. Compared to existing evaluation methods, it significantly boosts accuracy for other languages and demonstrates strong adaptability in low-resource programming environments. This work thus offers a promising direction for code evaluation in multilingual, low-resource settings.

References

Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 185–196, 2024.

Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Verified code transpilation with llms, 2024. URL https://arxiv.org/abs/2406.03003.

Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. Chateval: Towards better llm-based evaluators through multi-agent debate. *arXiv preprint arXiv:2308.07201*, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Lin Chin-Yew. Rouge: A package for automatic evaluation of summaries. In *Proceedings of the Workshop on Text Summarization Branches Out, 2004*, 2004.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

Michael Denkowski and Alon Lavie. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the ninth workshop on statistical machine translation*, pages 376–380, 2014.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–22, 2025.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1-2):81–93, 1938.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634*, 2023.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis (2020). *arXiv preprint arXiv:2009.10297*, 2009.

Weixi Tong and Tianyi Zhang. Codejudge: Evaluating code generation with large language models. *arXiv preprint arXiv:2410.02184*, 2024.

Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. Does bleu score work for code migration? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 165–176. IEEE, 2019.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36: 46595–46623, 2023a.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684, 2023b.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code. *arXiv preprint arXiv:2302.05527*, 2023.

Terry Yue Zhuo. Ice-score: Instructing large language models to evaluate code. *arXiv preprint arXiv:2304.14317*, 2023.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

Appendix A. Prompt Design

Figures 7, 8 and 9 illustrate three prompt templates: (1) Cross-Language → Python Translation, preserving original structure and logic; (2) Loop Invariants → SMT-LIB, extracting invariants and generating assertions; and (3) Loop-Free Code RBSE → SMT-LIB, symbolically exploring branches and outputting SMT-LIB assertions.

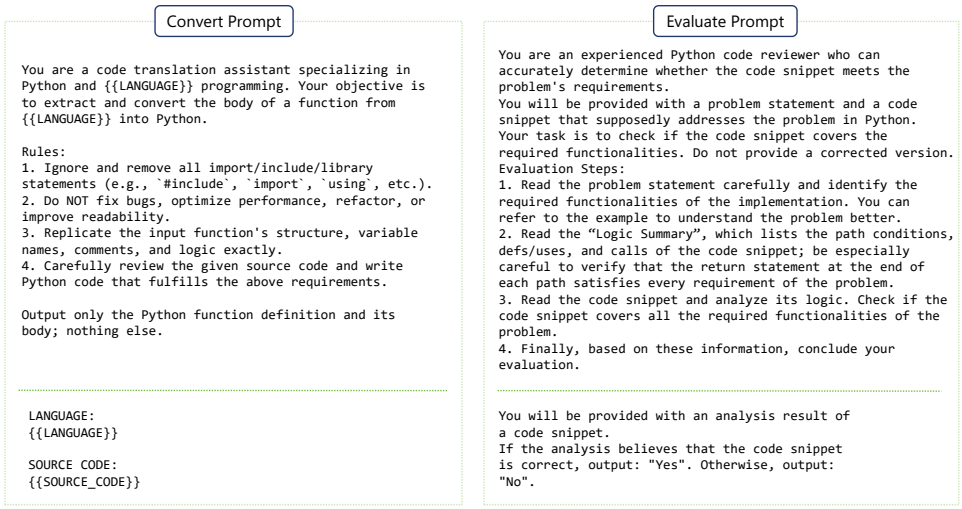


Figure 7: The complete prompt template for converting code from other languages into Python and evaluating the Python code.

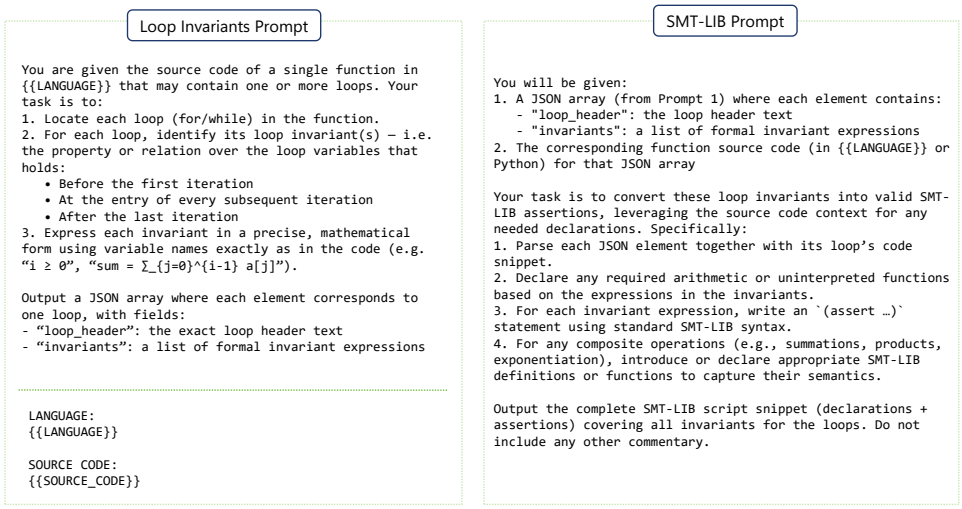


Figure 8: Loop Invariant Extraction Prompt and Generate language-loop SMT-LIB Formula Prompt.

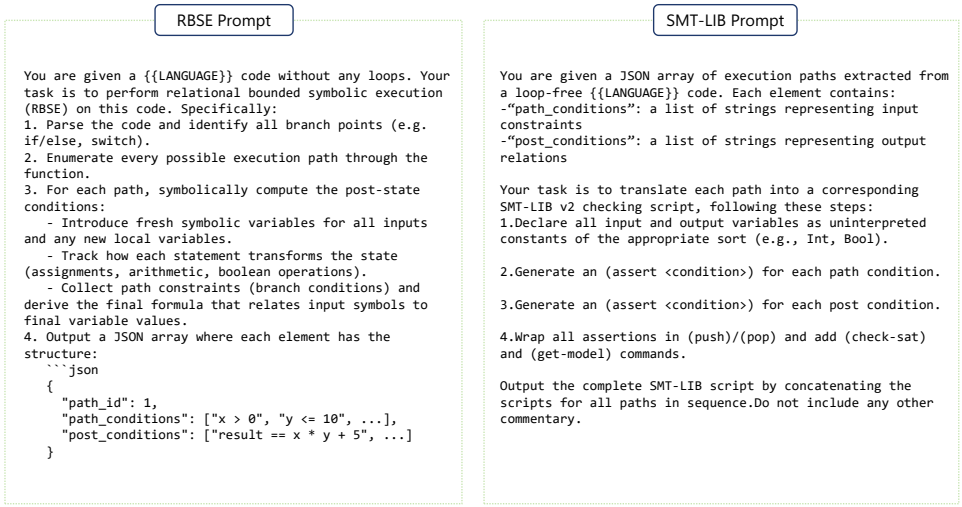


Figure 9: The complete prompt template for loop-free code and generating language-free SMT-LIB Formula.