Beyond Translation Accuracy: Addressing False Failures in LLM-Based Code Translation

Anonymous Author(s)

Abstract

Large Language Models (LLMs) have achieved remarkable success in automated code translation. While prior work has focused on improving translation accuracy through advanced prompting and iterative repair, the reliability of the underlying evaluation frameworks has received less attention. In this paper, we demonstrate that a significant number of reported failures in code translation are not due to incorrect logic, but rather *evaluation-induced errors* stemming from improper compilation flags, missing library links, and unconfigured runtime environments. We conduct a large-scale empirical study across five programming languages (C, C++, Java, Python, Go) and three benchmarks (Avatar, CodeNet, EvalPlus) using GPT-4o. Our analysis identifies and categorizes common false negatives, revealing that standard evaluation pipelines frequently misclassify semantically correct translations. Our findings highlight the necessity for transparent, configuration-aware evaluation standards to accurately assess progress in LLM-based code translation.

Keywords

Large Language Models, Code Translation, Program Correctness

ACM Reference Format:

Anonymous Author(s). 2026. Beyond Translation Accuracy: Addressing False Failures in LLM-Based Code Translation. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

During code translation, migrating software from one programming language to another while preserving functional correctness, is a long-standing challenge in software engineering. Such migration is critical for maintaining legacy systems [4, 8, 26], improving performance [5, 19], and ensuring long-term maintainability [18], yet it often requires extensive manual effort and expert knowledge. Recent advances in large language models (LLMs) have significantly improved automated code translation [13], making it feasible to translate non-trivial programs across programming languages such as C, C++, Java, Python, and Go.

Despite these advances, fully automated code translation remains unreliable in practice. Prior work has largely evaluated translation quality using compilation success and test case execution as the primary indicators of correctness. While this methodology is reasonable in principle, it implicitly assumes that the evaluation pipeline, including compilation commands, flags, and runtime configurations is itself correct and faithful to the translated program's requirements.

In this paper, we show that this assumption often does not hold. Through an in-depth analysis of existing code translation artifacts and benchmarks, we find numerous cases where translated code is *semantically correct* yet reported as incorrect due to inappropriate settings such as compilation configurations. In particular, missing or incorrect compiler flags (e.g., omitted libraries, language standards, or optimization constraints) can lead to compilation or runtime failures that are unrelated to translation quality.

This issue is especially problematic in large-scale empirical studies, where evaluation pipelines are automated, and errors introduced by incorrect compilation settings propagate silently into reported results. As a consequence, translation systems may be unfairly penalized, and empirical conclusions about model performance may be misleading.

To systematically study this problem, we design an analysis pipeline that iteratively inspects translation and evaluation outcomes. By separating translation errors from evaluation-induced errors, our analysis reveals a significant class of false negatives in existing benchmarks, cases where translations are semantically correct but misclassified due to incorrect compilation flags. Our findings highlight the need for more robust and transparent evaluation practices in code translation research.

2 Related Work

LLM-based code translation has been widely studied at the function and file levels. Prior work improves translation quality through architectural adaptations and contextual augmentation. For example, SteloCoder [12] adapts decoder-only multilingual models for code translation, while Spectra [10] and Saha et al. [17] leverage natural language or structural specifications as intermediate guidance. Retrieval-augmented approaches further improve few-shot translation by injecting relevant examples [2], and pseudocode-based methods translate abstract representations into executable code [24].

Several works focus on correcting translation errors after generation. Rectifier [23] learns from faulty–corrected code pairs, while UniTrans [22] and ExeCoder [6] use execution feedback and semantic information to iteratively repair translations. Lost in Translation [13] categorizes common bugs and failure modes in LLM code translation. Multi-agent systems such as TransAgent [25] and BabelCoder [16] distribute responsibilities across translation, testing, and repair agents. These methods primarily target solving translation-induced errors while translating code.

Repository-level translation has also been explored. K3Trans [11] incorporates dependency and historical knowledge, while AlphaTrans [7] applies static analysis and neuro-symbolic decomposition. RepoTransBench [21] provides a benchmark highlighting challenges in translating real-world repositories with automated tests.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Some recent works on code generation emphasize synthesizing executable code from natural language instructions, docstrings, or functional specifications rather than translating from existing implementations. In single-agent approaches, models like CatCoder [14] and TOOLGEN [20] enhance generation by incorporating structural repository context or mimicking developer-tool interactions. Multi-agent frameworks further distribute these tasks; for instance, Self-collaboration [3] and CodeAgent [27] simulate software team dynamics or integrate external tools to improve reliability. While these methods share architectural similarities with translation—particularly when intermediate specifications are involved—they rely on sparse natural language intent.

Unlike prior work, we address evaluation-induced errors (e.g., incorrect compilation flags) that cause valid translations to be misclassified as failures.

3 Methodology

We analyze false failure cases in LLM-based code translation using an automated translation and evaluation pipeline. Given a source program and target language, the pipeline (1) translates the code using GPT-4, (2) compiles and executes the translated program using dataset-provided test cases, and (3) records compilation errors, runtime failures, and test mismatches.

To distinguish translation errors from evaluation-induced failures, we manually inspect evaluation artifacts, including compilation commands, error logs, and runtime configurations, without modifying the translated code. Cases where the code is semantically correct but fails due to incorrect configurations are classified as evaluation-induced errors.

This pipeline is used solely as an analysis tool to diagnose failure modes in existing benchmarks, rather than as a new translation or repair system.

4 Experimental Setup

All findings in this work are derived from the code translation task. We focus on five programming languages: C, C++, Java, Python, and Go, performing translations between each language pair. Our analysis utilizes three widely used benchmarks: Avatar [1], CodeNet [15], and EvalPlus [9], which provide parallel corpora and corresponding test cases across these languages.

We use GPT-4o to translate source programs into target languages. The translated programs are compiled and executed using the dataset-provided test cases via command-line execution. For each run, we record compilation outcomes, runtime behavior, and test results. The entire translation and evaluation process is fully automated and serves as the basis for identifying translation errors versus evaluation-induced failures.

5 Findings

Our extensive analysis reveals that a significant portion of reported failures in code translation benchmarks stems from artifacts in the evaluation pipeline rather than fundamental errors in the translated logic. We categorize and detail these key findings below.

5.1 Evaluation-Induced Compilation Errors

Natural language contamination. Large Language Models frequently intersperse natural language explanations or conversational fillers within the generated source code. Additionally, they often wrap code blocks in Markdown formatting (e.g., "‘cpp). If these artifacts are not rigorously sanitized before compilation, standard compilers interpret them as syntax errors. Consequently, perfectly functional code is rejected solely due to formatting issues, inflating the reported failure rate.

Missing library linking. Automated evaluation scripts typically rely on generic, minimal compilation commands (e.g., `gcc file.c -o file`) which are often insufficient for complex programs. Translated C programs frequently require explicit linker flags to resolve external dependencies. Common examples include `-lm` for mathematical functions in `math.h`, `-lgmp` for arbitrary-precision arithmetic in `gmp.h`, and `-lssl` / `-lcrypto` for cryptographic operations in `openssl/ssl.h`. The omission of these flags leads to linker errors that are incorrectly classified as translation failures.

Excessive global memory usage. We observed instances where C translations attempted to allocate extremely large uninitialized global arrays or variables, sometimes exceeding 16 GB of memory. While such declarations may be syntactically valid in the source language (or on systems with vast resources), they exceed the standard stack or heap limits of the evaluation environment. We classify these as genuine code errors, as the translation failed to adapt memory management to standard constraints.

5.2 Language-Specific Compilation Behavior

The nature of compilation failures exhibited distinct patterns depending on the target language's strictness and paradigm:

- **Python**: We observed virtually no compilation-stage errors after applying our iterative fix pipeline. As an interpreted language, Python's failures are predominantly runtime-based, simplifying the initial validation step.
- **Java/C/C++**: These statically typed languages suffered mostly from syntax violations and missing import statements. In C++, specifically, template instantiation errors and header dependency chains were frequent sources of build failure.
- **Go**: The Go compiler enforces strict discipline regarding unused imports and variables. LLMs often translate logic directly from looser languages without cleaning up redundant variable declarations, causing frequent build failures unique to Go's strict tooling.

5.3 Runtime and Test Mismatch Errors

Faulty PyTest - JUnit conversion. The automated conversion of test suites from Python (PyTest) to Java (JUnit) proved to be a source of error. In several cases, the LLM hallucinates incorrect assertions or fails to port the test logic accurately. This leads to correct Java translations failing tests because the ground-truth test case itself is flawed.

Unrealistic Timeout Constraints. We identified a systemic bias where correctly translated code failed due to rigid timeout limits inherited from the source language. Benchmarks often enforce the execution time limits of the source implementation (e.g., highly optimized C++). Consequently, valid translations in languages with

higher runtime overhead—such as Python (due to interpretation) or Java (due to JVM startup time)—are unfairly penalized, even when they implement the correct asymptotic algorithm.

Missing native APIs. Certain source-language features lack direct equivalents in the target ecosystem. For example, Python's dynamic `eval()` function has no native counterpart in Java or C++. LLMs often attempt to reimplement these complex features from scratch, resulting in custom implementations that are incomplete, buggy, or insecure, leading to runtime crashes.

Divergent API semantics. Subtle semantic differences in operators across languages cause silent failures. A prominent example is the modulo operator (%) applied to negative operands: Python retains the sign of the divisor, whereas Java and C retain the sign of the dividend. Such discrepancies result in semantic mismatches where the code runs without error but produces incorrect numerical outputs.

Logical and corner-case errors. Some failures stem from missing corner-case handling that cannot be inferred without deep knowledge of the source context. For instance, Python's `round()` function employs "banker's rounding" (rounding to the nearest even number, e.g., `round(2.5)` $\longrightarrow$ `2`), whereas C, C++, and Java typically round away from zero or truncate. These nuanced differences reflect a fundamental limitation of context-free translation.

Uninvoked entry-point logic. In datasets like Avatar and CodeNet, evaluation relies on capturing standard output (stdout). We found numerous cases where the LLM correctly implemented the required logic within a function or class but failed to generate the main execution block to invoke it. As a result, the program compiles and runs successfully but produces no output, leading to a test mismatch.

6 Conclusion

This work investigates the causes of false failure verdicts in code translation across five languages. We demonstrate that even semantically correct translations are often rejected due to evaluation-induced errors, specifically regarding compilation flags and runtime configurations. Future work will extend this analysis beyond code translation to broader software engineering tasks, incorporating more languages and datasets.

References

[1] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*, 2021.

[2] M. Bhattarai, J. E. Santos, S. Jones, A. Biswas, B. Alexandrov, and D. O'Malley. Enhancing code translation in language models with few-shot learning via retrieval-augmented generation. *arXiv preprint arXiv:2407.19619*, 2024.

[3] Y. Dong, X. Jiang, Z. Jin, and G. Li. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38, 2024.

[4] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf. Translating c to safer rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29, 2021.

[5] S. Gandhi, M. Patwardhan, J. Khatri, L. Vig, and R. K. Medicherla. Translation of low-resource cobol to logically correct and readable java leveraging high-resource java refinement. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 46–53, 2024.

[6] M. He, F. Yang, P. Zhao, W. Yin, Y. Kang, Q. Lin, S. Rajmohan, D. Zhang, and Q. Zhang. Execoder: Empowering large language models with executability representation for code translation. *arXiv preprint arXiv:2501.18460*, 2025.

[7] A. R. Ibrahimzada, K. Ke, M. Pawagi, M. S. Abid, R. Pan, S. Sinha, and R. Jabbarvand. Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation. *arXiv preprint arXiv:2410.24117*, 2024.

[8] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan. In rust we trust: a transpiler from unsafe c to safer rust. In *Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings*, pages 354–355, 2022.

[9] J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[10] V. Nitin, R. Krishna, and B. Ray. Spectra: Enhancing the code translation ability of language models by generating multi-modal specifications, 2024.

[11] G. Ou, M. Liu, Y. Chen, X. Du, S. Wang, Z. Zhang, X. Peng, and Z. Zheng. Enhancing llm-based code translation in repository context via triple knowledge-augmented. *arXiv preprint arXiv:2503.18305*, 2025.

[12] J. Pan, A. Sadé, J. Kim, E. Soriano, G. Sole, and S. Flamant. Stelocoder: a decoder-only llm for multi-language to python code translation, 2023.

[13] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[14] Z. Pan, X. Hu, X. Xia, and X. Yang. Enhancing repository-level code generation with integrated contextual information. *arXiv preprint arXiv:2406.03283*, 2024.

[15] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

[16] F. Rabbi, S. K. Saha, T. M. T. Pham, S. Wang, and J. Yang. Babelcoder: Agentic code translation with specification alignment. *arXiv preprint arXiv:2512.06902*, 2025.

[17] S. K. Saha, F. Rabbi, S. Wang, and J. Yang. Specification-driven code translation powered by large language models: How far are we? *arXiv preprint arXiv:2412.04590*, 2024.

[18] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional, 2003.

[19] H. M. Sneed. Migrating from cobol to java. In *2010 IEEE International Conference on Software Maintenance*, pages 1–7. IEEE, 2010.

[20] C. Wang, J. Zhang, Y. Feng, T. Li, W. Sun, Y. Liu, and X. Peng. Teaching code llms to use autocompletion tools in repository-level code generation. *arXiv preprint arXiv:2401.06391*, 2024.

[21] Y. Wang, Y. Wang, S. Wang, D. Guo, J. Chen, J. Grundy, X. Liu, Y. Ma, M. Mao, H. Zhang, et al. Repotransbench: A real-world benchmark for repository-level code translation. *arXiv preprint arXiv:2412.17744*, 2024.

[22] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1585–1608, 2024.

[23] X. Yin, C. Ni, T. N. Nguyen, S. Wang, and X. Yang. Rectifier: Code translation with corrector via llms, 2024.

[24] Q. Yu, Z. Huang, and N. Gu. Pseudocode to code based on adaptive global and local information. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 61–72. IEEE, 2023.

[25] Z. Yuan, W. Chen, H. Wang, K. Yu, X. Peng, and Y. Lou. Transagent: An llm-based multi-agent system for code translation, 2024.

[26] H. Zhang, C. David, Y. Yu, and M. Wang. Ownership guided c to rust translation. In *International Conference on Computer Aided Verification*, pages 459–482. Springer, 2023.

[27] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*, 2024.