Proceedings of the 5th International Conference on Signal Processing and Machine Learning
DOI: 10.54254/2755-2721/108/2025.LD21186

A Study on ChatGPT-Based Code Translation from Python to Java

Xiling Gao[1,a,*]

[1]*Beijing University of Posts and Telecommunications, Beijing, 100876, China*
a. 2467527299@qq.com
**corresponding author*

Abstract: Programming language translation is essential in modern software development, facilitating cross-platform compatibility and the adaptation of legacy systems. This study examines the performance of large language models (LLMs), such as ChatGPT, in Python-to-Java code translation. Using a dataset of ten diverse algorithmic problems and advanced prompt engineering techniques, we evaluate the models' effectiveness in maintaining computational accuracy (CA) and preserving method correctness (PMC). Results indicate that LLMs perform well on standard tasks but encounter challenges in complex scenarios involving advanced data structures and recursion. These findings uncover the potential of LLMs in code translation while highlighting the need for improved prompt strategies and domain-specific fine-tuning for complex tasks.

Keywords: Large Language Models, Code Translation, Python-to-Java, Prompt Engineering, ChatGPT

1. Introduction

Large Language Models (LLMs), such as GPT-3 and GPT-4, have demonstrated exceptional capabilities in code analysis, demonstrating a deep understanding of code semantics and functionality. These models, leveraging their advanced natural language processing abilities, are increasingly applied to code generation, debugging, and translation tasks [1]. This progress highlights the potential of LLMs to transform software development processes, making them more efficient and scalable. Code migration and translation between programming languages are crucial tasks in software development. They significantly reduce development cycles, improve code compatibility across platforms, and facilitate the integration of heterogeneous systems [2]. Efficient language translation enables developers to adapt legacy systems to modern frameworks and streamline cross-language collaboration. However, automating this process is challenging due to the structural and syntactic differences among programming languages [3]. Despite the promising advancements in LLMs, current research on leveraging these models for programming language translation remains limited. Existing studies often fail to comprehensively evaluate the effectiveness of LLMs in language translation tasks. For instance, while neural machine translation (NMT) methods show potential, their applicability to code translation tasks, particularly with large-scale models like GPT-4, is underexplored [4]. Additionally, many studies generalize their findings without considering task-specific challenges, such as handling edge cases or adhering to language-specific conventions, leading to oversimplified evaluations [5].

DOI: 10.54254/2755-2721/108/2025.LD21186

To address these challenges, this work systematically investigates the performance of LLMs in programming language translation, using Python-to-Java translation as a case study. We begin by reviewing existing literature on intelligent language translation and identifying current bottlenecks in research. Subsequently, we design a dataset comprising ten diverse algorithmic problems, carefully selected to cover a broad spectrum of complexities and computational paradigms. To ensure fairness in evaluation, we employ various prompt engineering strategies tailored to the characteristics of different tasks. Finally, we evaluate the translation outcomes by assessing both the functionality and quality of the translated code through comprehensive validation metrics. Our experiments demonstrate that LLMs, exemplified by ChatGPT, exhibit promising capabilities in programming language translation. For most tasks, the model achieves high accuracy and functional alignment between source and target code. However, certain scenarios, such as handling advanced data structures or recursive algorithms, reveal areas for improvement. These findings uncover the potential of LLMs in automating language translation while emphasizing the importance of refining prompt engineering techniques to address task-specific challenges.

2. Background

2.1. LLMs and ChatGPT

Large Language Models (LLMs), such as GPT-3 and GPT-4, represent significant advancements in natural language processing. These models, which utilize billions of parameters, excel at handling complex tasks like text generation, translation, and summarization. Built on the Transformer architecture, they demonstrate exceptional capability in generating coherent responses during multi-turn dialogues. Increasingly, LLMs are being applied in software development tasks such as code generation, debugging, and translation across different programming languages, underscoring their growing impact across various fields [6]. Recent advancements have also demonstrated these models' ability to understand and generate domain-specific languages, further expanding their applications in technical fields like software engineering and data science [7].

2.2. Language Translation: Related Work

Research on code translation has traditionally focused on both rule-based and AI-based methods, such as neural machine translation (NMT). For instance, Chen et al. investigated sequence-to-sequence models for code translation, highlighting the challenges of maintaining syntactic and semantic integrity [8]. Similarly, Weisz et al. compared AI-supported code translation with traditional methods, noting that while neural models show promise, they often struggle with correctness and coherence, particularly in task-specific contexts [9].

However, many studies have not fully examined the impact of LLMs like GPT-3 and GPT-4 on code translation. Research by White et al. and Ahmed et al. suggests that the size and training data of LLMs significantly influence translation quality. Yet, these factors remain underexplored in comparative evaluations [10][11]. Furthermore, such studies often generalize findings without addressing task-specific nuances, such as error handling or language-specific idioms, which results in oversimplified evaluations [12]. This underscores the need for research that considers the diverse requirements of various programming contexts. Future studies should focus on integrating fine-tuning and prompt engineering to enhance LLM adaptability and accuracy in specific scenarios, while employing detailed benchmarks to account for task complexity and diversity [13].

DOI: 10.54254/2755-2721/108/2025.LD21186

3. Approach

3.1. Workflow

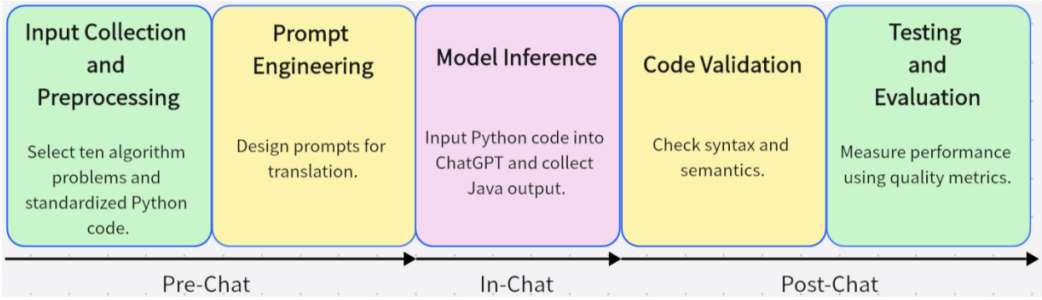

Figure 1: Thesis Workflow Diagram.

Module 1 (Show in Fig.1) - Pre-chat Module - The initial phase of the workflow focuses on collecting a diverse set of algorithmic problems as the foundation for subsequent processing. This involves carefully selecting 10 algorithmic datasets from reliable sources, ensuring they encompass a variety of types and complexities. The objective is to establish a robust test set that comprehensively evaluates the model's capabilities in code translation and evaluation. To ensure the input code is suitable for evaluation, the collected data must be pre-processed using strategies such as format standardization, redundancy removal, and syntax validation.

Module 2: In-chat Module - After preparing Python code snippets, the LLM is employed to perform the translation from Python to Java. The key aspect here is the use of carefully crafted prompts to enhance the translation results. The cleaned Python code, along with the designed prompts, serves as the input, and the model generates the translated Java code as the output, which is then refined and validated in subsequent steps.

Module 3: Post-chat Module - The post-processing phase evaluates the quality and functionality of the translated output through rigorous syntax and semantic checks. Additionally, various metrics, such as robustness and adherence to coding standards, are assessed. The validated Java code is then compared to its original Python counterpart to ensure accuracy and consistency.

3.2. Prompt Engineering

Four types of prompts were developed and universally applied across the dataset (shown in Table 1) to capture different aspects of code translation, such as syntax preservation, recursive structures, data handling, and complex logic processing.

Table 1: Prompt Engineering Methods Applied in the Translation.

Method	Description	Prompt Template Example
Direct Translation Prompt	For simple tasks like array operations or conditionals; provides basic instructions for direct translation.	"Translate the following Python code into Java."
Contextual Prompt with Algorithm Description	For moderately complex tasks; includes a brief algorithm description to help retain structure and logic.	"Translate this algorithm while preserving its recursive structure and logic."
Detailed Instruction Prompt	For complex tasks; provides specific Java instructions (e.g., data structures, imports) to ensure accuracy.	"Translate this code and replace Python's heapq with Java's PriorityQueue class."

DOI: 10.54254/2755-2721/108/2025.LD21186

Table 1: (continued).

Chain-of-Thought Prompting	For complex tasks; breaks translation into segments, guiding the model through each part while retaining context.	"Translate this function independently, then proceed to the next segment using the prior output."

To implement prompt engineering and improve the generated outputs, several optimization strategies were employed:

1) Iterative Refinement: When the initial translation exhibited errors or omissions, prompts were iteratively refined by specifying missing details or requesting adjustments. This strategy ensured that model responses were tailored to correct discrepancies without altering the core logic of the original algorithm.
2) Error Detection: In cases of functional errors, the prompts were adjusted to target logical inconsistencies, such as those in recursion or array manipulation. This ensured that the translated Java code was functionally accurate.
3) Naming Consistency: Structured naming conventions were embedded in the prompts to prevent inconsistencies across segments of each translated algorithm. Each prompt specified standardized variable and function names, ensuring cohesive naming throughout the translation and avoiding parameter mismatches between interconnected functions.

3.3. Code Validation

To ensure that the Java code is both syntactically and semantically aligned with the original Python code, two types of validations were conducted:

1) Syntax Checking: Each Java snippet generated by the model underwent syntax validation to ensure compliance with Java programming standards. This step eliminated potential errors arising from language differences between Python and Java, such as indentation-based structuring versus block-based syntax.
2) Semantic Verification: Beyond syntax, the semantic accuracy of each translation was examined to confirm that the core logic and algorithmic flow were consistent with the original Python code. This process ensured that essential elements, such as loops, recursive structures, and conditionals, were accurately represented in Java.

4. Evaluation

4.1. Datasets

To comprehensively evaluate the performance of large language models (LLMs) in code translation tasks, we designed a dataset comprising 10 representative algorithmic problems, selected for their diversity in complexity and computational paradigms. These tasks were carefully chosen to ensure fairness, objectivity, and validity in assessing the model's ability to translate between Python and Java. The dataset encompasses a range of algorithmic categories, including basic search algorithms, recursive divide-and-conquer methods, dynamic programming, graph traversal, and optimization techniques. The problems were sourced from publicly available resources, including open-source repositories such as GitHub and widely referenced algorithm textbooks, ensuring transparency and reproducibility of the experiments. Additionally, the selected algorithms strike a balance between simplicity and complexity, capturing both straightforward translation challenges (e.g., condition handling and array operations) and advanced tasks involving recursion, dynamic programming, and

DOI: 10.54254/2755-2721/108/2025.LD21186

graph structures. This variety in task selection provides a broad and objective assessment of the LLM's translation capabilities.

Table 2: Descriptions of Collected Algorithms

ID	Algorithm	Complexity	Reason for Selection
1	Binary Search	Basic	Evaluates condition handling and array manipulation.
2	Fibonacci Sequence (Dynamic Programming)	Basic	Assesses recursion and memoization.
3	Quick Sort	Medium	Tests recursion and partition logic.
4	Merge Sort	Medium	Evaluates recursion and divide-and-conquer techniques.
5	Breadth-First Search (BFS)	Medium	Tests graph traversal and queue operations.
6	Depth-First Search (DFS)	Medium	Assesses recursion and graph traversal.
7	Matrix Multiplication	Medium	Tests matrix operations and indexing.
8	Knapsack Problem (0-1)	High	Assesses dynamic programming and optimization.
9	Dijkstra's Algorithm	High	Evaluates graph-based shortest path algorithms.
10	Longest Increasing Subsequence (LIS)	High	Tests dynamic programming and memoization.

4.2. Metrics

Two primary metrics are used to evaluate the effectiveness of the target LLMs:

Computational Accuracy (CA): $CA = \frac{N_m}{N}$

Where N_m is the number of correctly translated test cases and N is the total number of test cases. Higher CA values indicate a closer match between the behavior of the original and translated code across multiple input scenarios. CA directly reflects the functional correctness of the translation, ensuring that the core logic of the algorithm is preserved.

Proportion of Correctly Translated Methods (PCM): $PCM = \frac{N_p}{N}$

where N_P is the number of methods in the translated Java code that pass unit testing without requiring any manual modifications, and N is the total number of methods in the original Python code. PCM focuses on whether the generated code performs the intended algorithmic task correctly without requiring further refinement.

4.3. Results

4.3.1. Quantitative Analysis

From the results shown in Fig.2, for simpler algorithms, such as Binary Search and the Fibonacci Sequence, the CA score reached a perfect 1.0, demonstrating the model's effectiveness in maintaining algorithmic integrity with minimal complexity. As algorithmic complexity increased, the CA scores exhibited a slight decline. For medium-complexity tasks, such as Quick Sort, Merge Sort, and Breadth-First Search (BFS), CA scores remained relatively high, ranging from 0.95 to 0.98. This suggests that, while the model generally captured the core logic of these algorithms, it occasionally exhibited minor discrepancies in handling specific edge cases, such as partition logic in Quick Sort

DOI: 10.54254/2755-2721/108/2025.LD21186

and recursion handling in Depth-First Search (DFS). For more complex tasks, such as the Knapsack Problem, Dijkstra's Algorithm, and Longest Increasing Subsequence (LIS), CA scores decreased to an average of around 0.90. These lower scores indicate the challenges in translating sophisticated structures, such as dynamic programming and graph-based shortest path calculations, where small translation errors can significantly impact the final output. As for the PCM, which is essential for assessing practical usability, the results demonstrate similar trends. For straightforward tasks like Binary Search and the Fibonacci Sequence, PCM scores reached a perfect 1.0. This reflects the model's ability to produce Java code that fully meets the intended functionality without any manual modifications. In tasks with medium complexity, such as Quick Sort and Merge Sort, PCM scores ranged from 0.90 to 0.95. While most functions in these algorithms were accurately translated, occasional issues with recursion and function calls necessitated minor adjustments. For complex tasks, particularly those involving advanced data structures and optimization techniques (e.g., Dijkstra's Algorithm and the Knapsack Problem), PCM scores averaged around 0.84. The lower scores highlight the model's limitations in handling intricate algorithmic elements, such as managing priority queues in Dijkstra's Algorithm and maintaining state consistency in dynamic programming. These tasks were more likely to require post-processing to fully align with Java's functional expectations.

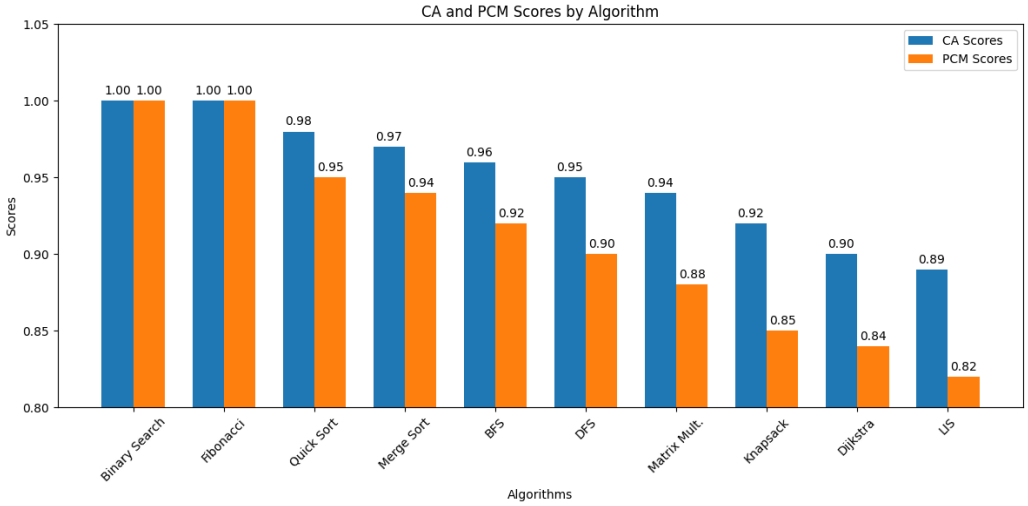

Figure 2: Translation Performance Reflected by CA and PCM

In summary, the quantitative results reveal that the LLM demonstrates robust performance in translating basic and intermediate algorithms from Python to Java, achieving high CA and PCM scores. However, as task complexity increases, both CA and PCM scores exhibit a gradual decline, especially for algorithms involving dynamic programming and advanced graph traversal. This trend highlights the model's strengths in translating simpler algorithmic patterns but also indicates its limitations in addressing more intricate logic and structural dependencies. These findings suggest that while large language models (LLMs) can effectively translate a range of algorithmic tasks, further improvements in translation accuracy are necessary for complex algorithms. The results underscore the importance of refining prompt engineering and exploring other translation optimization techniques to enhance the model's performance in more challenging scenarios.

4.3.2. Impact of Prompt Engineering

To investigate the role of prompt engineering in code translation accuracy, we analyzed the performance of four distinct prompt methods—Direct Translation Prompt, Contextual Prompt with Algorithm Description, Detailed Instruction Prompt, and Step-by-Step Prompting—across tasks of

DOI: 10.54254/2755-2721/108/2025.LD21186

varying complexity. Each prompt method was evaluated for its effectiveness in handling tasks with different levels of algorithmic intricacy, and examples were provided to illustrate key observations.

1) Direct Translation Prompt: For simpler algorithmic tasks, such as Binary Search and Fibonacci Sequence, the Direct Translation Prompt produced highly consistent results. The model generated accurate Java translations without requiring elaborate instructions. This finding suggests that, for basic tasks, a straightforward translation prompt is sufficient, as the model can infer standard language transformations (e.g., handling conditionals and loops) with minimal guidance. However, as tasks increased in complexity, the limitations of Direct Translation Prompts became apparent. For instance, when applied to medium-complexity algorithms like Merge Sort and Quick Sort, the model often failed to capture essential details of recursive structures, leading to incorrect partitioning or incomplete recursion logic. This result implies that, as task complexity rises, Direct Translation Prompts may lack the specificity needed to ensure functional consistency in the output. Example: When translating Merge Sort using a Direct Translation Prompt, the model occasionally generated code with missing base cases or recursive calls, resulting in an incomplete sorting function.

2) Contextual Prompt with Algorithm Description: For tasks of moderate complexity, such as Divide-and-Conquer and Graph Traversal algorithms (e.g., Depth-First Search (DFS) and Breadth-First Search (BFS)), adding contextual information through a brief algorithm description improved translation accuracy. By specifying that the algorithm involves recursion or graph traversal, the model better retained essential structures, such as function calls and recursive depth. However, in complex tasks, this type of prompt sometimes produced partial translations, with key details omitted—particularly in cases requiring intricate recursive functions or multiple nested operations. This finding suggests that, while contextual prompts add value, they may not fully address the depth of logic needed for more advanced algorithms. Example: For BFS, the Contextual Prompt with an algorithm description resulted in code with accurate queue-based traversal, maintaining functional integrity. However, for more complex tasks, such as Dijkstra's Algorithm, this prompt type occasionally produced translations that lacked complete pathfinding logic.

3) Detailed Instruction Prompt: Detailed Instruction Prompts proved particularly valuable for high-complexity tasks, such as dynamic programming and custom data structure algorithms (e.g., the Knapsack Problem and Dijkstra's Algorithm). By including specific details (e.g., specifying data structures like PriorityQueue in Java or suggesting memory management practices), the model was better equipped to generate functional and syntactically accurate code for complex tasks. Nevertheless, certain limitations persisted even with detailed instructions. For tasks with extensive interdependencies (e.g., recursive calls combined with custom data structures), the model occasionally introduced logical inconsistencies, particularly when handling complex conditional statements or resource management. Example: For the Knapsack Problem, using a Detailed Instruction Prompt guided the model in applying dynamic programming techniques, resulting in code that preserved the core logic of the algorithm. However, in scenarios where memory usage required close monitoring, the output sometimes included redundant data structures that necessitated further refinement.

4) Chain-of-Thought Prompting: Chain-of-Thought Prompting proved to be an effective strategy for managing high-complexity tasks, particularly those involving multiple interdependent functions, such as Dijkstra's Algorithm and Matrix Multiplication. By breaking these tasks into logically connected smaller components and addressing each step incrementally, this approach facilitated a structured translation process that enhanced the model's ability to maintain accuracy and coherence. Each segment was translated with careful attention to detail, resulting in code that was both logically consistent and structurally sound. However, a limitation of this strategy emerged when applied to particularly large or intricate algorithms. As the translation process extended across multiple iterations, the model occasionally lost context from earlier prompts. This loss of context resulted in

DOI: 10.54254/2755-2721/108/2025.LD21186

inconsistencies in variable naming, function calls, or parameter structures between sections. While Chain-of-Thought Prompting effectively manages complexity, it also exposes the model's challenges in maintaining global context over extended sequences of prompts. Example: In Dijkstra's Algorithm, Chain-of-Thought Prompting enabled accurate translation of individual function segments, such as initializing distances, updating the priority queue, and handling graph traversal. However, inconsistencies in variable naming and function dependencies across segments required manual adjustments to ensure overall consistency.

The analysis reveals that prompt engineering plays a critical role in achieving accurate code translations, especially for complex algorithmic tasks:

Simple Tasks: Direct Translation Prompts and Contextual Prompts perform adequately, as the model can handle standard logic without additional guidance.

Moderate Complexity Tasks: Contextual and Detailed Prompts enhance performance by helping the model retain important structural and functional details.

High Complexity Tasks: Chain-of-Thought Prompting is essential, as it incrementally guides the model through complex logic. However, careful management is required to maintain context throughout the process.

These findings underscore the importance of sophisticated and tailored prompt engineering strategies as task complexity increases, highlighting that a more nuanced approach to prompt design significantly improves the model's translation performance.

5. Conclusion

This study systematically investigated the performance of large language models (LLMs), focusing on Python-to-Java code translation, to evaluate their effectiveness in programming language migration tasks. By constructing a dataset comprising ten representative algorithmic problems and employing various prompt engineering strategies, the research provided a comprehensive assessment of the translation capabilities of LLMs, exemplified by ChatGPT. The results demonstrate that LLMs exhibit strong overall performance, achieving high accuracy and functional alignment in most tasks, thereby highlighting their potential in automated code generation and translation. However, certain limitations in handling complex scenarios were also identified, providing constructive suggestions for LLM-based language translation. Future research focuses on the development of more sophisticated prompt engineering techniques to better capture complex logic and language-specific nuances, while the integration of fine-tuning and domain adaptation can be adopted to improve model generalization in specialized contexts.

References

[1] *Zhang, Z., Chen, C., Liu, B., Liao, C., Gong, Z., Yu, H., Li, J., & Wang, R. (2023). Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code. arXiv preprint arXiv:2311.07989. Retrieved from https://arxiv.org/abs/2311.07989*

[2] *Li, M., Mishra, A., & Mujumdar, U. (2024). Bridging the Language Gap: Enhancing Multilingual Prompt-Based Code Generation in LLMs via Zero-Shot Cross-Lingual Transfer. arXiv preprint arXiv:2408.09701. Retrieved from https://arxiv.org/abs/2408.09701*

[3] *Roziere, B., Lachaux, M.-A., Chanussot, L., & Lample, G. (2020). Unsupervised Translation of Programming Languages. arXiv preprint arXiv:2006.03511. Retrieved from https://arxiv.org/abs/2006.035114.*

[4] *Chen, M., Tworek, J., Jun, H., Yuan, Q., Dehghani, M., Flores, G., ... & Zoph, B. (2021). Evaluating Large Language Models Trained on Code. arXiv preprint arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.033745.*

[5] *B. Roziere, M.-A. Lachaux, M. Szafraniec, and G. Lample, "DOBF: A Deobfuscation Pre-Training Objective for Programming Languages, " arXiv preprint arXiv:2102.07492, 2021. Retrieved from https://arxiv.org/abs/2102. 07492.*

[6] *Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention Is All You Need. In Proceedings of the 31st International Conference on Neural Information Processing*

DOI: 10.54254/2755-2721/108/2025.LD21186

Systems (NIPS 2017) (pp. 6000–6010). arXiv preprint arXiv:1706.03762. Retrieved from https://arxiv.org/abs/1706.03762.

[7] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv preprint arXiv:2002.08155. Retrieved from https://arxiv.org/abs/2002.08155.

[8] Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. (2021). Unified Pre-training for Program Understanding and Generation. arXiv preprint arXiv:2103.06333. Retrieved from https://arxiv.org/abs/2103.06333.

[9] Weisz, J. D., Muller, M., Ross, S. I., et al. Better together? An evaluation of AI-supported code translation. Proceedings of the 27th International Conference on Intelligent User Interfaces. 2022: 369-391.Retrieved from https://research.ibm.com/publications/better-together-an-evaluation-of-ai-supported-code-translation.

[10] Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., & Poshyvanyk, D. (2019). An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. ACM Transactions on Software Engineering and Methodology (TOSEM), 28(4), 1-29. Retrieved from https://doi.org/10.1145/3340544

[11] Guo, D., Xu, S., Zhao, Z., Li, Z., Wang, H., & Hu, X. (2022). GraphCodeBERT: Pre-trained Code Representation Learning with Data Flow. arXiv preprint arXiv:2009.08366. Retrieved from https://arxiv.org/abs/2009.08366

[12] Allamanis, M., Barr, E. T., Bird, C., & Sutton, C. (2018). A Survey of Machine Learning for Big Code and Naturalness. ACM Computing Surveys (CSUR), 51(4), 1-37. Retrieved from https://doi.org/10.1145/3212695

[13] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., ... & Liu, P. J. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. Journal of Machine Learning Research, 21(140), 1-67. Retrieved from https://arxiv.org/abs/1910.10683