Proceedings of CONF-MLA 2025 Symposium: Intelligent Systems and Automation: AI Models, IoT, and Robotic Algorithms
DOI: 10.54254/2755-2721/2025.LD29138

A Systematic Study of LLM-Based Code Translation from Multiple Perspectives

Jingxuan Yu

Smart City Industry Institute, Beijing Union University, Gu'an County, China
Yujingxuan20050511@126.com

Abstract. Large Language Models (LLMs) have demonstrated powerful capabilities in code analysis, exhibiting a deep understanding of code semantics and functionality. Programming code lies at the heart of software development, and the automation and intelligent generation of code can effectively shorten development cycles and reduce labor costs. Current research on code transformation using Large Language Models is gradually emerging. However, these works vary in research perspective, object, and goal, making it difficult to comprehensively evaluate the advantages and characteristics of Large Language Models in code translation tasks. Moreover, existing code translation primarily focuses on simple code translation for explicit tasks and remains incomplete for code translation of complex software systems.Therefore, this paper analyzes the inherent characteristics of Large Language Models in code translation based on their working mechanisms and characteristics. It comprehensively investigates research on code-intelligent generation using Large Language Models over the past two years, particularly examining effectiveness for complex generation tasks and relevant technologies from both task-oriented and technical perspectives. During this process, the impact of prompt engineering methods in code translation is specifically examined. Through systematic analysis and research, it has been found that Large Language Model systems like ChatGPT [1] are effective for code translation tasks with clear objectives. However, they still exhibit room for improvement in handling complex tasks, such as the inability to accurately translate code with tightly coupled contextual logic and the failure to generate code for complex software-system translation.

Keywords: GPT, code translation, LLM, software development

1. Introduction

Large Language Models (LLMs) have demonstrated powerful capabilities in the field of code analysis, showing a relatively deep understanding of code semantics. Programming code is the core of software development, and the automated generation of code can effectively reduce the development cycle and lower labor costs. From the modernization of legacy systems explored by [Krishna et al., 2021; Perez-Castillo et al., 2021; Kaleem et al., 2021] to the facilitation of seamless cloud migration discussed by [Author et al., 2022], the capability to automatically and precisely translate code remains critical [2].

© 2026 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/2025.LD29138

Research on code translation using Large Language Models (LLMs) has gradually emerged, yet these studies differ in research perspectives, objects, and objectives, making it difficult to uniformly evaluate the advantages and characteristics of LLMs in code translation tasks. Additionally, current code translation primarily focuses on simple code transformation for explicit tasks and remains insufficient for code translation in complex software systems. For instance, Marcos Macedo et al. (2025) presented CODETRANSENGINE—a plug-and-play backend solution designed to streamline [3] LLM-powered code translation—and Jialing Pan et al. (2024) introduced SteloCoder, a decoder-only StarCoder-based LLM designed specifically for multi-programming language-to-Python code translation [4].

These code translation tools and techniques, including those powered by large language models, have significantly streamlined the code transfer process by enhancing specific functionality. Some advanced systems are capable of achieving automated translation within large software frameworks or for particular programming languages without explicitly leveraging pre-trained models and domain-specific knowledge.

However, despite these advancements, current technologies still exhibit notable limitations, especially when handling complex tasks. In the context of multi-language code translation, challenges such as inconsistent semantic representation across languages, lack of support for less common language pairs, and difficulty in accurately translating idiomatic code constructs remain prevalent. For code with tightly coupled logical contexts, existing methods often struggle to capture the full range of dependencies and behavioral nuances, resulting in translation outputs that may contain syntactic errors, semantic discrepancies, or incomplete functionality. These limitations underscore the need for further research and development to improve the robustness and accuracy of code translation systems in complex scenarios.

Against the backdrop of LLMs gaining extensive adoption in code processing, this paper explores their application in code translation. Starting from the fundamental mechanisms and distinctive traits of LLMs, it dissects their inherent strengths and constraints in code translation tasks. Through a systematic review of research on code translation using LLMs over the past two years, this study analyzes their advantages and drawbacks from two key dimensions: task objectives and methodological approaches. Notably, it places particular emphasis on exploring how prompt engineering techniques influence the effectiveness of code translation.

Through systematic analysis and research, LLM systems like ChatGPT have proven effective in code translation tasks with clear objectives [1]. However, for complex tasks such as translation of code with intricate contextual logic or multi-language hybrid code, issues such as incomplete functional generation or failure to generate still persist, highlighting substantial opportunities for enhancement.

2. Background

2.1. LLM and ChatGPT

Large Language Models (LLMs) stand at the forefront of AI, aiming to achieve high-precision human language understanding and generation. Trained on massive text corpora (encompassing books, websites, academic papers, etc.), they handle diverse natural language processing tasks with capabilities in context comprehension, coherent response generation, and complex language operations across domains.

OpenAI's ChatGPT, a highly influential and widely used LLM built on the GPT (Generative Pre-trained Transformer) architecture—a benchmark in natural language processing—boasts notable

DOI: 10.54254/2755-2721/2025.LD29138

effectiveness in language tasks [1]. It supports context-coherent conversations, addressing queries across fields from scientific research to daily life (e.g., offering in-depth academic analysis for students); excels in text summarization, distilling lengthy texts into precise core content; features cross-language translation for high-precision multilingual translation; aids creativity (e.g., assisting writers in plot development, marketers in copy generation); and provides code snippet suggestions in software development to boost programming efficiency.

Other representative LLMs also hold key industry positions. Google's Bard [5] integrates knowledge graph and search engine advantages with real-time information processing capabilities suitable for common-sense QA, content generation, and current affairs queries. Baidu's Wenxin Yiyan [6] focuses on the Chinese context, excelling in understanding subtle Chinese semantic differences and cultural connotations, making it ideal for Chinese literature processing, historical research, and domestic business scenarios.

LLM-based technical applications have permeated various industries. In education, they act as virtual tutors for homework assistance and concept discrimination. In business, they power intelligent customer service for 24/7 consultations and auto-generate marketing content (e.g., social media copy, product descriptions). In software development, beyond aiding code writing and debugging, they excel in code translation (e.g., Python to Java, adapting old C programs to modern languages), lowering cross-language development barriers. At the personal level, they support personalized needs such as travel planning (generating itineraries with attractions, catering, and transportation).

2.2. Code translation

Code translation is the art and science of transforming code from one programming language to another while preserving its original functionality, logic, and purpose. Picture it as a skilled linguist retelling a software story in a different dialect, allowing code to thrive across diverse platforms and tap into the strengths of various languages.

Historically, code translation was a painstaking, error-prone manual endeavor. Programmers had to rewrite every line—a time-consuming task that often led to mistakes. Rule-based tools emerged next, armed with predefined rules for language mapping. Yet, they struggled to capture complex semantics and context. Statistical methods then stepped in, analyzing heaps of parallel code to make better translation choices. The real game-changer arrived with LLMs. Trained on vast code repositories, LLMs understand code context and intent, delivering more accurate and efficient translations.

In software modernization, code translation breathes new life into legacy systems. Outdated languages are transformed into contemporary ones, enhancing performance, security, and maintainability. For cross-platform development, it is key to adapting code for different operating systems and devices. In open-source projects, it welcomes developers of all language backgrounds, ensuring their contributions integrate seamlessly. CodeTransOcean [7] addressed dataset limitations in language variety and scale for real-world needs. It built datasets, tested multilingual strategies with CodeT5+, and evaluated ChatGPT, effectively expanding resources for improved translations. SteloCoder aimed to translate five languages to Python without specifying the input and boost training efficiency [4]. Using techniques like LoRA [8], MoE [9], and curriculum learning, it optimized translation pathways, achieving significant results on the XLCoST dataset. TransAgent took a distinct approach, employing a multi-agent system to fix syntax and semantic errors in code translation [10]. The agents acted as vigilant editors, ensuring translated code quality.

DOI: 10.54254/2755-2721/2025.LD29138

These ongoing endeavors, inspired by projects like CodeTransOcean [7], SteloCoder [4], and TransAgent [10], push the boundaries of code translation capabilities, making software development more versatile and powerful than ever before.

3. Study

In this section, a systematic review of LLM-based code translation is conducted from the perspectives of task objectives and technical methods, respectively.

3.1. The perspective of task objectives

Research on LLM-based code translation has gradually gained momentum in recent years. In terms of task objectives, it can be broadly divided into two aspects: general code translation and specific code translation.

3.1.1. General code translation

Typical code translation tasks aim to leverage enhanced Large Language Models and code analysis techniques to optimize performance in code translation. For instance, XIN YIN et al. investigated root causes of errors (e.g., compilation, runtime, functionality, and infinite loops) occurring during LLM-based code translation (e.g., package import failures, loop boundary errors). They proposed a universal error correction model, Rectifier [11], which learns from erroneous LLM implementations to achieve cross-model repair. Experiments on C++, Java, and Python translation tasks demonstrated this model's effectiveness in correcting translation errors. Both K3Trans and Rectifier possess optimization and error-correction capabilities [12]. K3Trans builds a self-evolving knowledge base based on triple knowledge, generates enhanced prompts, and corrects errors through a self-debugging mechanism. It is suitable for warehouse-level multilingual translation without restriction to specific language pairs. UniTrans [13] improves translation quality through three stages: test case generation, translation enhancement, and translation repair, supporting multiple language pairs and scenarios. The difference between the two lies in K3Trans' reliance on a knowledge base and self-debugging mechanism, whereas UniTrans depends on test case generation and repair processes.

Yan et al. developed CodeTransOcean [7], a large-scale benchmark dataset supporting language translation, which includes three multilingual datasets: MultilingualTrans (translation between mainstream languages), NicheTrans (translation between niche and mainstream languages), and LLMTrans (evaluation of LLM-generated code executability), as well as the DLTrans dataset for cross-deep learning frameworks. PolyHumanEval [14] constructs a multilingual benchmark extended from HumanEval covering 14 programming languages, focusing on evaluating LLMs' cross-language translation capabilities. In model frameworks and learning strategies, UniTranslator [2] adopts a multi-agent collaborative framework where functional LLM agents are coordinated by a Director LLM to perform translation and error correction. CoTran [15], built on CodeT5, optimizes translation via reinforcement learning with compiler feedback (CF: fine-grained rewards from syntax error positions) and symbolic execution feedback (SF: using bidirectional LLMs and symbolic execution to generate test cases for functional equivalence verification). It supports Java-Python mutual translation and extends to other language pairs. Regarding application scenarios and advantages, TransGraph [16] excels at translating complex software projects by leveraging call graphs and bridged debuggers, effectively reducing compilation and runtime errors. Meanwhile, Codetransengine [3], as a universal backend framework, supports multiple LLM inference backends

DOI: 10.54254/2755-2721/2025.LD29138

and integrates evaluation datasets and algorithms, providing infrastructure support for multilingual translation.

3.1.2. Specific code translation

Some works utilize LLMs to perform code generation for specific functions to support subsequent translation, such as software code generation, code testing, and code similarity detection. Mu et al [6]. proposed the ClarifyGPT [17] framework for code generation and test case translation, which improves accuracy via "detecting ambiguity → generating clarification questions → optimizing requirements → regenerating code." Experiments show it increases GPT-4's pass rate on MBPP from 70.96% to 80.80% and boosts multiple models' average performance by 7.17 percentage points in simulation tests. SteloCoder [4], based on StarCoder, is a dedicated multilingual-to-Python model with five language experts, auto-identifying mixture-of-experts, and curriculum learning. It scores 73.76 CodeBLEU on XLCoST (≥3.5 above prior SOTA) and achieves >95% recognition for C++, PHP, and JavaScript. Fluorine [18] is a framework for C/Go-to-Rust translation, using JSON-serialization fuzzing to verify I/O equivalence (without target tests) and four repair strategies (simple restart performs best). GPT-4/Claude 2.1 achieve 47.3%/44.2% success; simple restart boosts success rates by 7–21%, with Rust outputs 1.98× more concise than C2Rust. Both are scenario-specific, high-performing code translation tools.

3.2. The perspective of technical methods

Typical LLM-based code translation technical methods include prompt engineering, code analysis, fine-tuning, and RAG, which are introduced as follows.

3.2.1. Prompt engineering

Yan et al. improved general code generation via prompt engineering optimization [7]: complex prompt designs impact output—complex prompts introduce noise, whereas simple ones perform best. Ambiguity impairs quality. One-shot learning boosted DSR@0 by 1.72%, but Chain-of-Thought (CoT) reduced performance. LLMs in PolyHumanEval [14] (e.g., CodeLlama) also used prompt engineering; providing language function signatures increased cross-language translation correctness accuracy (CA) by 3.47–3.70 percentage points. With two-shot examples, CodeLlama-34B achieved a 43.5% CA improvement in Python-to-other-language translation versus zero-shot.

In prompt engineering, UniTrans [13] integrates generated test cases into prompts to enhance LLMs' understanding of logic and I/O. Combined with iterative prompting based on error contexts, it increases the success rate by 5.5% (with GPT-4 reaching 12%) and reduces compilation errors.

3.2.2. Code analysis

Building on CodeT5+'s multilingual modeling approach in CodeTransOcean [7], four strategies optimize translation by analyzing grammar and other elements. Executability is verified through LLMTrans and AutoTransExecuter. ChatGPT is evaluated using DSR@K, where "debugging success rate@K" aids error identification.

TRANSAGENT [10] employs code analysis through multi-agent collaboration: Code Aligner achieves block alignment via Control-Flow Analysis (39.6% higher than TransMap); Semantic Error Fixer locates errors dynamically; Syntax Error Fixer parses compiler errors (32.9% CA improvement), boosting Python-to-Java CA to 89.5%. TransGraph [16] uses code analysis in Call

DOI: 10.54254/2755-2721/2025.LD29138

Graph Construction (LSP) and Bridged Debuggers (DAP), outperforming baselines by 10.2% (71.4% vs. GPT-4's 61.4%), with components contributing 8–15% to accuracy.

Similarly, K3Trans [12] embodies code analysis. It extracts target-language code samples (e.g., Rust open-source projects), analyzes their grammatical styles and idiomatic features for translation reference, mines dependency usage examples from the repository, analyzes project-specific dependencies to ensure contextual adaptability, and examines cross-language grammatical translation patterns through historically successful translated function pairs to guide structural alignment. CoTran [15] integrates code analysis via three components: CF, SF, and kw-Tok, enhancing compilation with error analysis, semantic checks, and structural precision. Code analysis supports translation error classification and defect analysis. Rangeet Pan et al. manually analyzed 1,748 GPT-4 failed translations, categorizing errors into five types and fifteen subcategories (syntactic/semantic issues, etc.) via compilation/runtime analysis. They compared LLMs and transpilers (e.g., C2Rust), highlighting transpilers' structural strengths and LLMs' natural language advantages to inform hybrid solutions.

3.2.3. Fine-tuning and RAG (Retrieval-Augmented Generation)

3.2.3.1. Models using fine-tuning technology

To optimize multi-language-to-Python translation and improve specific language pair accuracy, SteloCoder [4] uses the 15.5B-parameter StarCoder backbone. It fine-tunes five specialized language experts (for C++, C#, Java, JavaScript, PHP) via Low-Rank Adaptation (LoRA). Each expert contains only 9M parameters (0.06% of StarCoder's total), enabling parameter-efficient training. In CodeTransOcean [7], CodeT5+ [19] explores strategies like One-to-One and One-to-Many through fine-tuning to optimize translation quality for both high-resource and low-resource language pairs. It enhances generalization by leveraging shared source-language features and improves performance for low-resource languages (e.g., Visual Basic, Go). Based on the lightweight CodeT5+ 220M model, Rectifier [11] learns error-repair patterns of existing LLMs (e.g., ChatGPT, StarCoder) via fine-tuning, enabling cross-model universal error correction without adjustments for specific LLMs. FedLLM [20] uses LoRA [21] in federated learning, fine-tuning local models then aggregating parameters to enhance translation (e.g., C#→Java) with multi-client data while preserving privacy. CodeLlama [14] employs self-training on validated parallel code, boosting cross-language transfer (e.g., Python-to-Go accuracy +14.57%).

3.2.3.2. Models utilizing RAG technology

K3Trans [12] constructs a self-evolving knowledge base via triple knowledge enhancement. It retrieves target-language code samples (e.g., Rust projects), dependency usage examples (function/variable calls), and historically successful function pairs to enhance context awareness and guide syntax translation. CodeTransEngine [3] integrates algorithms like N-shot translation and Tree of Code Translation (ToCT), optimizing translation paths by retrieving intermediate language examples and improving accuracy using historical translation patterns.

UniTranslator [2], aiming to enhance semantic accuracy for low-resource languages and cross-paradigm translation (e.g., Solidity to Move), implements concept-driven translation via a Concept Agent that retrieves core concepts from textbooks/docs, combined with Specialized Code Agents' language expertise.

DOI: 10.54254/2755-2721/2025.LD29138

3.2.3.3. Summary

Fine-tuning optimizes models for specific pairs/tasks via parameter-efficient methods (e.g., LoRA [21]), as seen in SteloCoder and FedLLM, balancing performance and resources. RAG enhances contextual adaptability by retrieving external knowledge (code samples, etc.), represented by K3Trans and UniTranslator. Their combination effectively boosts code translation accuracy, domain adaptability, and generalization.

4. Discussion

4.1. Advantages

Generally, current code translation works focus on optimizing functions like accuracy and executability, improving results through innovative techniques. Based on LLMs, they enhance translation capabilities by combining analysis, test cases, and feedback mechanisms. They cover multiple programming languages and scenarios, breaking traditional tools' language limitations. Automated processes reduce manual intervention and improve efficiency. Compared to traditional approaches, these works exhibit excellent overall performance, with optimization in translation accuracy and executability. Robustness is significantly improved; multi-strategy optimization alleviates issues like low-resource languages and complex dependencies. For example, K3Trans achieves a Pass@1 of 67.7% in repository-level migration via triple knowledge augmentation. These works advance theoretical development and provide efficient solutions for industrial needs (e.g., legacy system upgrades, cross-platform migration), accelerating software modernization.

4.2. Disadvantages

Despite technological breakthroughs, these models face limitations in adaptability and scalability (e.g., SteloCoder requires retraining for new languages; LLMs show low success rates in real projects). They also encounter environmental constraints like dependence on high-quality knowledge or standardized structures, alongside shortcomings in error repair and generalization. Feedback mechanisms lack standardization, exhibiting insufficient accuracy and coverage. Addressing these requires standardizing feedback, enhancing complex scenario handling, and reducing environmental dependencies.

4.3. Future directions

Future research should focus on four aspects:

- Improving adaptability/scalability: Develop general architectures to reduce retraining costs for new languages/scenarios. Combine technologies like meta-learning to overcome bottlenecks in low-resource languages, complex grammars, and large codebases.
- Addressing dependency/environmental constraints: Build flexible knowledge integration mechanisms to handle source-language dependencies. Explore lightweight multi-agent frameworks to optimize system complexity/overhead.
- Enhancing error repair/generalization: Design robust semantic modules to locate/fix complex errors. Dynamically tune models via diverse scenario training and reinforcement learning.
- Optimizing feedback mechanisms: Establish unified standards to clarify error definitions/priorities. Improve accuracy through precise analysis and expand coverage for cross-file/dynamic runtime errors.

DOI: 10.54254/2755-2721/2025.LD29138

5. Conclusion

This work systematically studies LLM-based code translation, noting that LLMs exhibit strong capabilities in code analysis and generation. However, current research remains inadequate for translating complex systems. This paper reviews relevant work from task objectives (general/specific translation) and technical methods (prompt engineering, code analysis, fine-tuning, RAG). It analyzes these models' advantages in functional optimization and multi-language coverage, alongside disadvantages like limited adaptability and environmental dependence. Furthermore, it proposes future research directions across four dimensions to improve model adaptability, scalability, and performance robustness.

References

[1] ChatGPT [OL], https: //chat.openai.com/chat
[2] Karanjai R, Blackshear S, Xu L, et al. Collaboration is all you need: LLM Assisted Safe Code Translation [C]//Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering. 2025: 671-675.
[3] Macedo M, Tian Y, Adams B. CodeTransEngine: Ready-to-use Backend for LLM-based Code Translation [C]//ICLR 2025 Third Workshop on Deep Learning for Code.
[4] Pan J, Sadé A, Kim J, et al. SteloCoder: a decoder-only LLM for multi-language to Python code translation [J]. arxiv preprint arxiv: 2310.15539, 2023.
[5] Google. Response on the concept of artificial intelligence [Online]. https: //bard.google.com/, 29 Aug. 2025.
[6] Baidu. Response on the concept of blockchain [Online]. https: //yiyan.baidu.com/
[7] Yan W, Tian Y, Li Y, et al. Codetransocean: A comprehensive multilingual benchmark for code translation [J]. arxiv preprint arxiv: 2310.04951, 2023
[8] E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, and J. Ma, "LORA: LOW-RANK ADAPTION OF LARGE LANGUAGE MODELS, " arXiv preprint arXiv: 2106.09685, 2021
[9] M. I. Jordan and G. E. Hinton, "Adaptive mixtures of local experts, " Neural Computation, vol. 3, no. 1, pp. 79-87, 1991.
[10] Yuan Z, Chen W, Wang H, et al. Transagent: An llm-based multi-agent system for code translation [J]. arxiv preprint arxiv: 2409.19894, 2024.
[11] Code translation with corrector via llms [J]. arxiv preprint arxiv: 2407.07472, 2024.
[12] Ou G, Liu M, Chen Y, et al. Enhancing llm-based code translation in repository context via triple knowledge-augmented [J]. arxiv preprint arxiv: 2503.18305, 2025.
[13] Yang Z, Liu F, Yu Z, et al. Exploring and unleashing the power of large language models in automated code translation [J]. Proceedings of the ACM on Software Engineering, 2024, 1(FSE): 1585-1608.
[14] Tao Q, Yu T, Gu X, et al. Unraveling the Potential of Large Language Models in Code Translation: How Far are We? [C]//2024 31st Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2024: 353-362.
[15] Jana P, Jha P, Ju H, et al. Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution [J]. arxiv preprint arxiv: 2306.06755, 2023.
[16] Luo Y, Yu R, Zhang F, et al. Bridging gaps in llm code translation: Reducing errors with call graphs and bridged debuggers [C]//Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. 2024: 2448-2449.
[17] Mu F, Shi L, Wang S, et al. Clarifygpt: Empowering llm-based code generation with intention clarification [J]. arxiv preprint arxiv: 2310.10996, 2023.
[18] Eniser H F, Zhang H, David C, et al. Towards translating real-world code with llms: A study of translating to rust [J]. arxiv preprint arxiv: 2405.11514, 2024
[19] Wang Y, Le H, Gotmare A D, et al. Codet5+: Open code large language models for code understanding and generation [J]. arxiv preprint arxiv: 2305.07922, 2023.
[20] Kumar J, Janapati V L S, Tanguturi M R, et al. I Can't Share Code, but I need Translation--An Empirical Study on Code Translation through Federated LLM [J]. arxiv preprint arxiv: 2501.05724, 2025.
[21] Hu E J, Shen Y, Wallis P, et al. Lora: Low-rank adaptation of large language models [J]. ICLR, 2022, 1(2): 3.