Proceedings of CONF-SEML 2026 Symposium: Importance of Machine Learning Methods and Analysis in Engineering
DOI: 10.54254/2755-2721/2026.AS32228

A Multi-Perspective Investigation into Code Migration for Large Language Models

Yuqi Wang

Sichuan University, Chengdu, China
1173681988@qq.com

Abstract. With the rise of large language models (LLMs), their powerful capabilities in code understanding and generation have paved a new technical path for automated code migration, making LLM-based code migration a growing research hotspot. Despite the surging number of relevant studies, there is still a lack of systematic collation, resulting in an unclear overall picture of its technical paths and task adaptability. To address this issue, this paper conducts a comprehensive survey of typical works related to LLM-based code migration in recent years and a systematic multi-perspective investigation of the important ones, focusing on two core dimensions: Methods and Tasks. Centering on different methodological paths such as LLM feedback and traditional analysis, as well as task scenarios including data resources, quality analysis, and tool development and improvement, the paper systematically summarizes their highlights and limitations. The study finds that methodologically, the generation and understanding capabilities of LLMs have improved the efficiency and accuracy of code migration; in terms of tasks, existing works have realized the leap from simple snippet translation to complex industrial-grade scenarios. However, current research still has obvious common limitations in handling non-functional attributes such as in-depth semantic understanding, cross-file global dependencies, and performance and security assurance. Based on the survey results, this paper points out that the lack of technical reliability and evaluation testing dimensions is the key bottleneck at present, and proposes future development directions including constructing a knowledge-enhanced migration framework, establishing a standardized evaluation system, and processing full project associations.

Keywords: Code Migration, Code Analysis, Software Evaluation, LLM

1. Introduction

In the life cycle of software systems, with the rapid evolution of technical ecosystems and continuous advancement of hardware environments, code migration has become a critical task to ensure system vitality and maintainability. As an important branch of the code analysis field, code migration aims to convert source code from one programming language or framework to another while maintaining the original semantic equivalence. Traditional code migration relies heavily on human intervention, which is not only inefficient but also prone to introducing imperceptible logical deviations. In recent years, large language models (LLMs) with the Transformer architecture as the

DOI: 10.54254/2755-2721/2026.AS32228

core have demonstrated code modeling capabilities that surpass traditional rule-based translators. From the early dedicated Neural Machine Translation (NMT) architectures to the current models with strong reasoning capabilities such as GPT-4 and DeepSeek, the introduction of LLMs has brought brand-new development opportunities for code migration.

Nevertheless, despite the abundant existing research results, researchers and engineers still face severe challenges. Current works exhibit fragmented characteristics, with significant differences in task definition, methodological paths and evaluation criteria among different studies, and a lack of systematic collation and integration. The unclear overall picture of technical paths and task adaptability makes it difficult for developers to select the most effective technical route from numerous solutions when facing large-scale, high-reliability industrial-grade migration tasks.

To address the above problems, this paper conducts a systematic multi-perspective survey of LLM-based code migration technologies. We comprehensively retrieved and analyzed typical works on intelligent code migration in the past five years, deeply dissected 20 distinctive and important works among them, and constructed a dual analytical framework including task orientation and methodology. Starting from these two core dimensions, this paper systematically summarizes the highlights and limitations of existing research. From the methodological perspective, this paper deeply analyzes the evolutionary logic and core ideas of three technical paths: LLM-based, NMT-based, and traditional non-learning-based analysis. From the task perspective, it covers research progress in five core dimensions including datasets and benchmark suites, evaluation dimensions and quality analysis, and model improvement and tool development. Through the comparative analysis of these cross dimensions, this paper conducts a comprehensive dissection of the research status, highlights and limitations in the field of code migration, and draws several important conclusions: current code migration methods have made substantial progress in improving functional correctness by introducing iterative feedback and hybrid verification mechanisms; meanwhile, the evaluation system is shifting from superficial syntactic similarity to in-depth testing of functional attributes and robustness. The survey results of this study reveal that despite significant technological progress, existing methods are generally limited in dealing with complex control flow, in-depth semantic understanding and cross-file system dependencies, and pay obvious insufficient attention to non-functional attributes such as performance and security. Based on this, this paper further proposes several future development directions worthy of attention, including developing verification methods independent of complete test suites, realizing globally aware migration of project-level dependencies, and incorporating non-functional attributes such as performance and security into migration constraints. It is hoped to provide a reference for the next generation of more intelligent and reliable automated code migration tools, and narrow the huge gap between current academic models and the actual industrial-grade code refactoring needs.

2. Background

2.1. LLM

As the core of the generative artificial intelligence field, Large Language Models (LLMs) exhibit excellent language understanding and content generation capabilities by virtue of ultra-large-scale parameter learning based on the Transformer architecture. Through pre-training on massive textual data, LLMs can not only handle complex natural language tasks such as logical reasoning, multi-turn dialogue and text creation, but also show extremely strong code modeling capabilities. In the field of software engineering, LLMs can accurately capture the syntactic structure, logical flow and semantic correlation of programs, thus realizing code completion, defect detection and automated

DOI: 10.54254/2755-2721/2026.AS32228

generation. At present, typical representatives of general-purpose LLMs include OpenAI's GPT series, Google's Gemini series and Anthropic's Claude series; meanwhile, the academic and industrial circles have also launched a series of domain-specific LLMs optimized for programming tasks, such as Hugging Face's StarCoder, Meta's CodeLlama, as well as domestic models including DeepSeek-Coder and Qwen-Coder. Through continuous evolution on a large number of code repositories, these models have greatly improved the automation software development and maintenance.

2.2. Code migration

Code migration refers to the process of converting source code from one programming language or framework to another and ensuring that it maintains the original semantic equivalence in the target environment, which is a key link in software evolution and system upgrading. Its typical application scenarios include the modernization of legacy systems (e.g., migrating obsolete COBOL code to Java), cross-platform porting, and language refactoring to improve operational efficiency (e.g., rewriting Python logic into higher-performance C++ or Rust). Despite the widespread demand, code migration tasks face severe challenges: there are often huge differences in syntactic structure between different programming languages, conflicts in underlying paradigms (such as the differences between procedural-oriented and object-oriented), and complex incompatibility issues of library function APIs. In addition, how to achieve high-precision automated conversion in the absence of large-scale parallel corpus (i.e., bilingual code pairs with the same function) and ensure that the migrated code is completely consistent with the original code in logic remains a core problem to be solved urgently in current academic research and engineering practice.

3. Multi-perspective study

3.1. Overview

This study conducts a comprehensive survey of LLM-based code migration technologies. We systematically retrieved nearly 100 papers from top conferences and journals related to LLM-based code analysis in the past five years, and finally identified 20 cutting-edge works directly related to code migration through rigorous screening in terms of the relevance and influence of research content, themes and methods. Among them, 13 are from top journals in the software engineering field and 7 from top journals in the AI field, with 85% being influential works in the past three years, covering diverse migration scenarios from code snippet translation to industrial-grade version iteration. This reflects that LLMs are in a stage of rapid evolution in the field of code migration and their importance is gradually increasing in current academic research and industrial practice. To fully demonstrate the characteristic attributes of relevant works, this paper will conduct a comprehensive analysis of these 20 papers from the perspectives of task orientation and methodology respectively.

Table 1. Overview of the studied works

NO.	Work Name	Task (Problem to solve)	Approach	Result	Limitation
1	Functional Property Testing [1]	Testing functional properties of translation models	Extended NOMOS framework; property-guided search	Detected thousands of property violations; search process significantly increased correct translations.	Coverage of automated property generation needs further improvement.

DOI: 10.54254/2755-2721/2026.AS32228

Table 1. (continued)

2	Code Smells & Correctness [2]	Evolution of Code Smells and correctness	Empirical study of 9 LLMs; iterative repair and static analysis	Revealed that LLMs replicate typical smells of target languages; iterative repair may harm functional correctness.	Lack of deep metrics for long-term code maintainability.
3	CoST dataset & MuST-PT [3]	Parallel data scarcity in program translation	CoST dataset; MuST-PT model	Supported 42 language pairs; significant performance boost for low-resource languages.	Restricted to snippet-level alignment; lacks project-level context.
4	LLVM IR-based Translation [4]	Capturing cross-language semantic differences	LLVM IR enhancement; IR-aware unsupervised objectives	Java→Rust accuracy improved by 79%; significantly reduced type-related errors.	High dependency on compiler frontends; limited generalizability across non-compiled languages.
5	PyCraft: TBE Automation [5]	Automating code changes via examples	PyCraft tool (LLM + static/dynamic analysis)	Expanded example sets by 58x; 83% acceptance rate by developers in real-world projects.	Performance bottlenecks when handling extremely complex control-flow variants.
6	Mutation-Based Evaluation [6]	Evaluating trustworthiness and robustness	MBTA (Mutation-Based Translation Analysis); MTS metric	Revealed "rote learning" (memorization) and overfitting to original code despite high BLEU scores.	High computational overhead for large-scale mutation analysis.
7	UniTrans: Iterative Repair [7]	Unleashing LLM power in translation	UniTrans framework (Execution-based feedback & iterative repair)	LLaMA-7B accuracy increased by 28%; significantly reduced logic and I/O errors.	Still struggles with deep logic contradictions and complex I/O types.
8	Python Library Migrations [8]	Characterizing Python library migrations	PyMigBench-2.0 dataset and PyMigTax taxonomy	40% of migrations involve non-function mappings; challenged the "function-to-function" assumption.	Restricted to the Python ecosystem; lacks cross-language universality.
9	G-TransEval Taxonomy [9]	Correcting bias in trivial benchmarks	G-TransEval (Complexity-based 4-level taxonomy)	Revealed that models struggle with library-level and algorithm-level (Type 3/4) translations.	Manual data collection size limits the coverage of long-tail scenarios.
10	Code Clone Detection Review [10]	Impact of code representation on clone detection	Systematic review of text, lexical, syntactic, and semantic layers	Semantic representations detect complex clones but sacrifice scalability and language independence.	Lack of targeted research on clones generated by large language models.
11	LLM4SE: Systematic Review [11]	SLR of LLMs for Software Engineering	Systematic Literature Review (SLR); Quasi-Gold Standard	Identified the structural and semantic capture advantages of encoder-decoder architectures.	SLR is time-sensitive; limited coverage of the latest Agent-based workflows (post-2024).
12	LLM4SE: Trends & Challenges [12]	Trends and open problems in LLM4SE	Qualitative analysis; model categorization	Emphasized that encoder-decoder models (T5/BART) are most suitable for translation.	Functional correctness and hallucination problems remain core unresolved challenges.
13	CodeXGLUE Benchmark [13]	Benchmark for code understanding and generation	CodeXGLUE benchmark	Established a standard comparison platform; identified large room for improvement in capturing semantics.	Evaluation metrics still over-rely on syntactic indicators (BLEU).
14	TransCoder: Unsupervised NMT [14]	Unsupervised source-to-source translation	TransCoder (Unsupervised NMT: XLM + DAE + Back-translation)	Achieved high accuracy for C++/Java/Python without any parallel training data.	Fails on deep refactoring involving complex 3rd-party library calls.
15	CodeTransOcean Benchmark [15]	Multilingual and niche language evaluation	CodeTransOcean benchmark; DSR@K metric	Proved that multilingual modeling significantly improves translation for both high and low-resource languages.	Automated debugging success in real-world projects remains a major challenge.
16	LLM Translation Bug Taxonomy [16]	Taxonomy of bugs introduced by LLMs	Large-scale empirical study (7 LLMs, 1700 samples)	GPT-4 performed best, but project-level success was extremely low (0-8.1%); categorized bug distributions.	Lack of effective automated bug prevention and filtering mechanisms.

DOI: 10.54254/2755-2721/2026.AS32228

Table 1. (continued)

17	HPGN: Hierarchical Attention [17]	Capturing inter-sentence dependencies	HPGN (Hierarchical Pointer Generator Network)	Overall score improved by 20.4; significantly smaller parameter footprint	Restricted generalization ability toward unseen translation patterns.
18	Human-AI Partnership Study [18]	Human-AI partnership in code migration	Qualitative study with 11 Java developers	Defined AI as a "teammate"; emphasized that robust test suites are crucial for building trust.	Small sample size; conclusions may be subjective or context-specific.
19	Google: Internal Case Study [19]	Large-scale industrial code migration	Google case study (LLM + AST hybrid workflow)	Achieved 50% time savings; revived long-stagnant internal migration projects.	Highly dependent on Google's internal infrastructure; high external adoption cost.
20	CodeMEnv: Cross-Env Benchmark [20]	Cross-environment library migration	CodeMEnv benchmark	Revealed that models are familiar with new functions but struggle with "New2Old" back-porting.	Extremely poor performance on adapting new code to older library environments.

3.2. Task-based analysis

From the perspective of the core problems to be solved and the attributes of the tasks performed, research in the field of code migration has evolved from a single syntactic conversion to a multi-dimensional task system, aiming to comprehensively improve the reliability, scientificity and efficiency of LLMs in actual migration scenarios.

3.2.1. Study and survey

This type of task aims to sort out the development trajectory of the code migration field, define the core challenges at the current stage and point out future directions. Reference [10] systematically summarizes the research progress of code clone detection and discusses the evolution from textual to semantic representation. Through a Systematic Literature Review (SLR) of more than 400 papers, Reference [11] identifies the inherent advantages of LLMs in processing structured code in code translation tasks. Reference [12] further analyzes the rising trend of LLMs in software engineering applications (including code migration), and clearly points out that hallucination and correctness verification are still open problems that have not been completely overcome.

3.2.2. Datasets and benchmark suites

The core of such tasks is to provide researchers with comparable experimental materials and execution environments. Among the above papers, aiming at the problem of data scarcity and low-resource migration scenarios, Reference [3] proposes the CoST dataset containing 42 language pairs. As a multi-dimensional comprehensive data benchmark covering 10 tasks, CodeXGLUE released in Reference [13] has established a standardized comparison platform for works in the field. Reference [15] focuses on multilingual and niche programming languages, filling the relevant gaps through CodeTransOcean. In addition, CodeMEnv developed in Reference [20] has expanded the benchmark testing environment to more challenging cross-library version and dynamic adaptation scenarios.

3.2.3. Evaluation dimensions and quality analysis

With the continuous improvement of model performance in recent years, on the basis of having standardized benchmarks, how to scientifically measure the quality of migrated code has immediately become a research focus. Many new evaluation indicators and methods are proposed in the above papers, for example, Reference [1] introduces property-guided search to test functional correctness. To solve the bias of evaluation indicators, Reference [6] proposes to use mutation

DOI: 10.54254/2755-2721/2026.AS32228

analysis to detect the robustness of translators, while Reference [9] establishes a complexity-based four-level taxonomy to achieve fine-grained model evaluation. At the same time, for the empirical evaluation of migration effects, Reference [2] focuses on the analysis of the spread and evolution of code smells, and Reference [16] classifies the types and distributions of bugs introduced by LLMs through large-scale empirical research.

3.2.4. Model improvement and tool development

This task is committed to improving the success rate of code migration through technical frameworks. In terms of model performance improvement, Reference [4] and [17] have improved the semantic modeling capabilities of models by introducing Compiler Intermediate Representation (IR) and hierarchical attention mechanism respectively. TransCoder proposed in Reference [14] has realized high-precision source code translation under unsupervised conditions; at the tool level, the PyCraft framework proposed in Reference [5] and the UniTrans framework constructed in Reference [7] use iterative feedback mechanisms to handle complex syntactic variants and repair migration defects. In addition, Reference [19] shares how Google internally applies the LLM + AST hybrid workflow to large-scale industrial-grade code migration, proving the practical production value of automated tools.

3.2.5. Empirical research and human-AI collaboration

Studying the essential characteristics of migration tasks and the interaction mode between developers and AI can guide better technological implementation. For example, Reference [8] conducts an in-depth empirical analysis of the characteristics of Python library migration, challenging the traditional "function-to-function" assumption. Reference [18] explores the trust-building process when developers collaborate with AI through qualitative research, and emphasizes the key role of automated verification methods in supporting human decision-making.

To sum up, the research task focus of current works has shifted from the early "syntactic mapping" to a comprehensive exploration covering data provision, model improvement, quality evaluation and engineering empirical research. All tasks jointly promote the development of the code migration field towards an application dimension with greater breadth and depth, laying a solid technical foundation for addressing increasingly complex challenges in the future.

3.3. Approach-oriented analysis

From the perspective of core technical paths, code migration research has formed three main methodological paths: LLM-based, Neural Machine Translation (NMT)-based, and traditional non-learning-based analysis.

3.3.1. LLM-based approaches

LLM-based approaches have become the mainstream paradigm of current research. Such approaches utilize the contextual understanding and generation capabilities of large-scale pre-trained models, and assist with various guidance strategies and enhancement techniques to implement code migration tasks. In terms of pure LLM application and prompt engineering, Reference [2], [16] and [18] deeply discuss how to use prompt engineering to stimulate the potential of general-purpose models in handling code smells and migration tasks. To further improve the correctness of generated code, LLM enhancement and feedback iterative repair technologies have emerged as the times

DOI: 10.54254/2755-2721/2026.AS32228

require. For example, the PyCraft tool proposed in Reference [5] processes complex syntactic variants through expansion, while the UniTrans framework in Reference [7] introduces a real-time feedback repair mechanism based on execution results. In addition, the hybrid application of LLMs with traditional technologies such as Abstract Syntax Tree (AST) or symbolic verification has also shown great value. The internal practice of Google reported in Reference [19] proves that this hybrid generation and verification framework can effectively meet the rigorous requirements in large-scale industrial-grade code migration.

3.3.2. NMT-based approaches

This type of approach is mainly based on Neural Machine Translation (NMT) technology, that is, using specially designed neural networks to realize code sequence translation, and usually focuses on capturing code semantics through architectural innovation or the introduction of program representation.

In terms of Intermediate Representation (IR) enhancement, Reference [4] significantly enhances the model's ability to capture underlying logic by introducing language-agnostic LLVM IR, reducing translation errors in cross-language type mapping. Targeting the hierarchical characteristics of code structure, progress has also been made in specific architectural design tasks, such as the Hierarchical Pointer Generator Network (HPGN) designed in Reference [17], which can better model the dependencies between sentence tokens. Under the condition of limited corpus resources, unsupervised learning and pre-training methods play a key role. TransCoder proposed in Reference [14] has laid the technical foundation for unsupervised source code translation, while the MuST-PT model proposed in Reference [3] improves the generalization performance of models for low-resource programming languages through snippet-level alignment pre-training.

3.3.3. Traditional non-learning-based analysis approaches

Such approaches do not rely or do not fully rely on the generation capabilities of models, but provide quality assurance and theoretical support for code migration through program analysis technologies or empirical research. Mutation analysis and testing are the key tasks among them. Reference [1] extends the NOMOS framework for functional property testing, while Reference [6] introduces mutation analysis as the core means to measure the robustness of translators, effectively identifying the overfitting risk of models. In the dimension of static analysis and clone detection, Reference [10] lays a theoretical foundation for understanding code duplicate logic and representation methods through a systematic review of traditional technologies. In addition, manual verification and taxonomic research provide clear practical guidance for automated tools. For example, the empirical classification of the characteristics of Python library migration in Reference [8] reveals the complexity of asymmetric API mappings in actual development, thus providing important empirical support for the design of subsequent automated migration solutions.

To sum up, the methodological evolution of code migration presents an obvious trend of transformation from "domain-specific architectures" to "large-scale model adaptation". Although early methods based on deep neural networks have provided valuable experience in structure capture, the current research focus has tilted towards LLMs. More importantly, single generative models have also gradually withdrawn from the application stage, and the current dominant approach is an integrated method combining generation and verification, which deeply integrates the generation capabilities of LLMs with traditional program analysis technologies such as mutation testing and static analysis. This not only improves the performance of code migration in functional

DOI: 10.54254/2755-2721/2026.AS32228

correctness, but also provides a more solid technical guarantee for addressing more complex and large-scale software engineering evolution tasks in the future.

4. Highlights vs. limitations

This paper conducts an in-depth review of the 20 cutting-edge works surveyed. On the whole, current research has made leapfrog progress in the automation level of migration and the ability of semantic capture, but still exposes obvious limitations in terms of robustness and systematic support when facing complex industrial scenarios.

From the perspective of the highlights of typical works, the introduction of the feedback closed-loop mechanism is one of the most important breakthroughs in recent years. For example, Reference [7] (UniTrans) and Reference [5] (PyCraft) have changed the previous "one-time generation" mode. By introducing dynamic execution feedback and multi-step iterative repair, the model can autonomously correct deviations according to the error information of compilers or test cases. This method has significantly improved the functional correctness of code, upgrading LLMs from simple translators to intelligent tools with preliminary repair capabilities.

Another major highlight is the continuous deepening of evaluation dimensions. For example, Reference [1] and Reference [6] no longer only rely on traditional text similarity indicators (such as BLEU), but deconstruct the underlying logic of models through functional property testing and mutation analysis. This kind of evaluation reveals the vulnerability of models when facing minor code perturbations, providing scientific measurement criteria for building more reliable migration tools. In addition, the large-scale migration practice of Google shown in Reference [19] proves that combining the flexibility of LLMs with deterministic AST verification is an effective path to solve the migration of industrial legacy systems, providing a successful example for the transformation of academic research to practical application. However, while existing works show great potential, they also have unavoidable limitations.

First, the verification of code migration results is over-reliant on the testing environment. The core challenge of code migration lies not only in syntactic conversion, but also in how to ensure that the migrated code can maintain logical equivalence with the original code in the target language. At present, most of the excellent migration solutions (e.g., Reference [7]) verify the correctness of migration results through executing unit tests. However, in the migration of real legacy systems, the original projects often lack the comprehensive test suites mentioned in papers, which means that many automated migration tools are difficult to perform verification without ready-made test cases. Therefore, when facing migration tasks with high reliability requirements, security still faces huge challenges.

Second, the research granularity is mostly limited to the function level. Although Reference [14] and Reference [17] can handle independent code logic well, they lack an in-depth understanding of project-level global architectural dependencies (such as inter-file invocation relationships and global state management). In addition, the lack of non-functional attributes is another blind spot. Current research focuses almost entirely on "functional correctness", while ignoring the operational performance, memory overhead and security of the migrated code. For example, Reference [16] finds that models may introduce security vulnerabilities while ensuring functionality, which is unacceptable in actual production environments. Finally, as revealed in Reference [20], models are dependent on training data. When facing rapidly iterated APIs or private code ecosystems, existing offline models often show obvious lag, and it is difficult to handle the latest version adaptation problems due to the lack of relevant training data.

DOI: 10.54254/2755-2721/2026.AS32228

While examining the limitations of the above academic research, it is necessary to make a horizontal comparison with the current mainstream commercial large models. Globally, OpenAI's GPT-4o and Anthropic's Claude 3.5 Sonnet have performed prominently in code migration tasks with their strong reasoning capabilities. At the same time, domestic models such as DeepSeek-Coder-v2 and Qwen2.5-Coder have also shown extremely strong competitiveness, reaching the world-leading level in indicators such as instruction following and multilingual translation. Through comparison, it can be found that commercial large models and the cutting-edge academic frameworks surveyed in this paper present a complementary relationship. The core advantage of commercial large models lies in their powerful zero-shot generation capability; although they can write syntactically correct code, they are still prone to logical hallucinations when handling extremely complex migration tasks. Cutting-edge academic research, on the other hand, focuses more on building reliability frameworks including compilation checks, symbolic verification or AST constraints on the basis of the underlying capabilities of these commercial large models. In other words, commercial models can provide efficient generative power, while academic research is committed to developing supporting quality inspection and navigation tools to ensure that the generated code truly meets the industrial-grade logical equivalence requirements, each with its own advantages and limitations.

In summary, current code migration research is in a critical stage of transition from syntactic mapping to semantic equivalence. Although feedback mechanisms and hybrid verification frameworks have greatly enhanced the practicality of tools, there is still a lack of comprehensive solutions for large-scale, cross-file system-level code refactoring without test support. In view of the above limitations, future research can synergistically use the efficient generation of commercial models and academic frameworks, focusing on specific directions such as real-time synchronization of the latest APIs, balanced consideration of code performance and security, and processing of full project associated dependencies. Through these follow-up works, it is expected to further narrow the gap between academic models and real industrial production needs.

5. Future directions

Based on the aforementioned various analyses of the literature collection, this paper extracts several research directions worthy of attention in the field of code migration in the future.

First, in view of the over-reliance of code migration verification on existing test cases, future research should explore verification methods in the absence of testing environments. Since many legacy systems do not have ready-made test suites, researchers can try to develop automated differential testing tools, which generate a large number of random inputs automatically and run them on both the old and new code at the same time, comparing their outputs to find logical inconsistencies.

Second, the granularity of code migration needs to leap from the function level to the full project level. Future tools should not only focus on how to write a certain code module, but also have a global perspective, which can automatically identify and process cross-file function calls, the transmission of global variables and the overall architectural design of the project. This indicates that the research direction will shift to how to enable models to understand complex engineering dependencies, ensuring that all associated parts of the entire project can be updated synchronously when a certain part is modified.

At the same time, future migration tasks must take into account the non-functional quality of code, rather than just making the code run successfully. Researchers need to take the operational efficiency, memory usage and security of code as important constraints in migration. This means

DOI: 10.54254/2755-2721/2026.AS32228

that the code generated by LLMs must not only be logically correct, but also conform to the best practices of the target language and have security. For example, automatically identifying and repairing potential security vulnerabilities during migration, or automatically optimizing algorithm performance according to the characteristics of the target language, which may be used as an important indicator to measure the maturity of migration tools.

Finally, to solve the problems of pre-trained knowledge lag and "translation hallucination" caused by the rapid update of modern software libraries, future research should focus on developing knowledge-enhanced migration architectures, enabling models to accurately capture the behavioral differences between source interfaces and the latest APIs of target languages when performing tasks. This can ensure that migration tools can still write accurate code that meets current standards when facing rapidly iterated technical frameworks or specific tasks, thus significantly improving the reliability of automated migration tools in real industrial scenarios.

In summary, the focus of future research can be on how to make migration tools more independent, more holistic, higher-quality and more timely. Through in-depth exploration in the directions of logical verification, project-level association processing, performance and security optimization, and real-time knowledge synchronization, code migration technology will continue to progress, providing stronger technical support for the relevant software industry.

6. Conclusion

This paper conducts a systematic review of the field of LLM-based code migration. Through an in-depth analysis of 20 cutting-edge literatures from the two dimensions of task orientation and methodology, we sort out the evolutionary path of this field from early syntactic mapping to the current pursuit of semantic equivalence. The analysis shows that the introduction of iterative feedback mechanisms and hybrid verification frameworks in recent years has significantly improved the functional correctness of code migration, and the evaluation system has deepened from simple text similarity indicators to more convincing quality measurements covering functional property testing, mutation analysis and so on. These progresses have jointly promoted the advancement of code migration from academic research to real industrial scenario applications. However, the research also reveals the common limitations of current code migration technologies, mainly including over-reliance on existing test suites, research granularity mostly limited to the function level with a lack of project-level architectural awareness, and general neglect of non-functional attributes such as code performance and security. These bottlenecks hinder the further widespread application of automated migration tools in large-scale and complex real code migration tasks.

In the future work, code migration research needs to develop towards a more autonomous, more global, more comprehensive and more timely direction. Specifically, future work should focus on developing verification methods independent of complete test suites, constructing architecture-aware models that can understand project-level dependencies, and incorporating non-functional goals such as efficiency and security into migration constraints. Through continuous exploration in these directions, it is expected to further bridge the gap between academic research and industrial practice, and ultimately provide solid and reliable automated support for the smooth evolution and modernization of software systems.

References

[1] Eniser H F, Wüstholz V, Christakis M. Automatically testing functional properties of code translation models [C]//Proceedings of the AAAI Conference on Artificial Intelligence. 2024, 38(19): 21055-21062.

DOI: 10.54254/2755-2721/2026.AS32228

[2] Feischl C, Kern R. Large Language Models for Code Translation: An In-Depth Analysis of Code Smells and Functional Correctness [J]. ACM Transactions on Software Engineering and Methodology, 2025.
[3] Zhu M, Suresh K, Reddy C K. Multilingual code snippets training for program translation [C]//Proceedings of the AAAI conference on artificial intelligence. 2022, 36(10): 11783-11790.
[4] Szafraniec M, Roziere B, Leather H, et al. Code translation with compiler representations [J]. arXiv preprint arXiv: 2207.03578, 2022.
[5] Dilhara M, Bellur A, Bryksin T, et al. Unprecedented code change automation: The fusion of llms and transformation by example [J]. Proceedings of the ACM on Software Engineering, 2024, 1(FSE): 631-653.
[6] Guizzo G, Zhang J M, Sarro F, et al. Mutation analysis for evaluating code translation [J]. Empirical Software Engineering, 2024, 29(1): 19.
[7] 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.
[8] Islam M, Jha A K, Akhmetov I, et al. Characterizing Python Library Migrations [J]. Proceedings of the ACM on Software Engineering, 2024, 1(FSE): 92-114.
[9] Jiao M, Yu T, Li X, et al. On the evaluation of neural code translation: Taxonomy and benchmark [C]//2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2023: 1529-1541.
[10] Chen QY, Li SP, Yan M, et al. Code clone detection: A literature review [J]. Ruan Jian Xue Bao/Journal of Software, 2019, 30(4): 962980 (in Chinese). http: //www.jos.org.cn/1000-9825/5711.htm
[11] Hou X, Zhao Y, Liu Y, et al. Large language models for software engineering: A systematic literature review [J]. ACM Transactions on Software Engineering and Methodology, 2024, 33(8): 1-79.
[12] Fan A, Gokkaya B, Harman M, et al. Large language models for software engineering: Survey and open problems [C]//2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE). IEEE, 2023: 31-53.
[13] Lu S, Guo D, Ren S, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation [J]. arXiv preprint arXiv: 2102.04664, 2021.
[14] Roziere B, Lachaux M A, Chanussot L, et al. Unsupervised translation of programming languages [J]. Advances in neural information processing systems, 2020, 33: 20601-20611.
[15] Yan W, Tian Y, Li Y, et al. Codetransocean: A comprehensive multilingual benchmark for code translation [J]. arXiv preprint arXiv: 2310.04951, 2023.
[16] Pan R, Ibrahimzada A R, Krishna R, et al. Lost in translation: A study of bugs introduced by large language models while translating code [C]//Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 2024: 1-13.
[17] Li Z, Xu MR, Wu YH, et al. Source code migration model based on hierarchical attention mechanism [J]. Application Research of Computers, 2023, 40(10). (in Chinese).
[18] Omidvar Tehrani B, M I, Anubhai A. Evaluating human-ai partnership for llm-based code migration [C]//Extended abstracts of the CHI conference on human factors in computing systems. 2024: 1-8.
[19] Nikolov S, Codecasa D, Sjövall A, et al. How is google using ai for internal code migrations? [C]//2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2025: 481-492.
[20] Cheng K, Shen X, Yang Y, et al. Codemenv: Benchmarking large language models on code migration [C]//Findings of the Association for Computational Linguistics: ACL 2025. 2025: 2719-2744.