

Compiler-Guided Legacy Code Translation for Maintainable Software Migration

Authors

Matthias Huber^{1*}, Aline Morel², and Giulia Bianchi³

Affiliations

School of Computer and Communication Sciences, EPFL – École polytechnique fédérale de Lausanne, Station 14, CH-1015 Lausanne, Switzerland

*Corresponding author: m.huber@epfl.ch

Abstract

Legacy software migration remains difficult because old systems often contain undocumented business rules, obsolete library calls, implicit data formats, and tightly coupled procedures. Large language models can translate code between programming languages, but direct translation may change program behavior, introduce hidden runtime errors, or produce code that is hard to maintain. This study investigates compiler-guided legacy code translation for software migration. We propose TransAgent, a multi-agent translation model with a program slicing agent, a semantic translation agent, a compiler feedback agent, a behavior equivalence agent, and a refactoring agent. The program slicing agent separates legacy modules into control-flow units, data-processing units, and external-interface units. The semantic translation agent converts source code into target-language implementations while preserving variable dependencies and exception logic. The compiler feedback agent identifies syntax errors, type mismatches, unreachable code, and missing libraries. The behavior equivalence agent compares outputs between legacy and translated versions using regression tests and generated boundary cases. The refactoring agent improves naming consistency, modular structure, exception handling, and dependency clarity. The empirical study focused on enterprise migration from legacy Java and procedural Python scripts to modern Java and TypeScript services. The dataset contained 186 modules from inventory, billing, customer service, and reporting systems, with 214,000 lines of source code and 3,420 archived change requests. Instead of only measuring pass rates, the study also measured migration effort, behavior preservation, and maintainability. TransAgent preserved reference outputs in 88.6% of migrated modules after two repair rounds, compared with 71.9% for direct LLM translation. The number of manual correction hours decreased from 2.8 hours to 1.4 hours per thousand lines of code. The maintainability index increased by 16.2 points on average, and circular dependencies were reduced by 31.5%. In modules involving financial calculation rules, numerical deviation was kept below 0.1% in 94.3% of test cases. The results indicate that compiler feedback and behavior-equivalence checking can make automated code translation more suitable for real software migration.

Keywords: Code translation; legacy software migration; compiler feedback; behavior equivalence; program repair; maintainability; multi-agent systems

1. Introduction

Legacy software remains essential in many enterprise systems, but maintaining and modernizing these systems is increasingly difficult because they often depend on outdated libraries, obsolete language features, undocumented data structures, and business rules that exist only in source code. Code migration therefore requires more than syntactic conversion between programming languages. A successful migration must preserve program behavior, data-processing logic, numerical results, external interfaces, and interactions with surrounding modules. Early neural code models demonstrated that programming-language translation could benefit from representations of code structure, identifiers, and contextual semantics [1,2]. Related research on robust representation learning has also shown that predictive models may rely on multiple shortcut features simultaneously and that identifying and removing such shortcuts can improve generalization [3]. This issue is relevant to code migration because translation models may reproduce lexical patterns or familiar API structures without preserving the underlying computational intent. Later translation methods incorporated unit tests, execution signals, and compiler representations to improve functional correctness [4,5]. These advances substantially improved translation at the function level, but enterprise migration introduces broader challenges that are not captured by isolated code snippets. Real-world legacy modules frequently span multiple files and depend on project-specific classes, external libraries, configuration files, database schemas, and historical interfaces [6,7]. A translated function may therefore compile correctly while still changing the behavior of the larger system. API substitutions can alter exception handling, numerical precision, memory behavior, or data representation, while seemingly minor modifications to loops, conditionals, and type conversions can propagate into substantial output differences. These problems become more severe when source and target languages follow different execution models or when legacy dependencies no longer have direct equivalents. As a result, syntactic similarity or token-level translation accuracy provides only limited evidence that the migrated program remains semantically equivalent to the original implementation.

Compiler feedback and execution results provide important signals for correcting these errors. Test-guided code generation has shown that models can use pass/fail information to revise candidate programs, while self-debugging approaches allow models to inspect failed outputs and generate subsequent repairs [8,9]. Multi-agent frameworks have extended this idea by assigning code generation, testing, diagnosis, and repair to separate components [10,11]. Such methods are useful because compilation errors, type mismatches, missing imports, and failed tests often provide direct evidence about translation defects. However, compiler success alone cannot verify that the migrated code preserves the intended behavior. A program may compile and execute

while producing incorrect boundary conditions, inaccurate numerical results, changed state transitions, or incompatible output formats. Effective migration therefore requires feedback mechanisms that combine syntactic validation with semantic and behavioral checks. Recent studies have further demonstrated that library usage, algorithmic logic, and differences between language or library versions remain major sources of translation failure [12,13]. Benchmarks at the class, repository, and execution-environment levels have consistently shown lower model performance when dependencies, build systems, and realistic runtime settings are incorporated into the task [14,15]. These findings indicate that translation quality decreases as the evaluation moves from independent functions toward realistic software systems. Repository-level migration also creates an error-propagation problem: an incorrect type definition, interface mapping, or dependency replacement in one component can cause failures across many downstream modules. Models therefore need mechanisms for identifying which portions of a legacy program are relevant to a specific translation target and for preserving relationships between callers, callees, shared variables, and external resources. Industrial studies suggest that language models can reduce developer effort when automated translation is combined with systematic review and validation [16-19]. Static analysis, compiler-based repair, and call-graph information can further improve translation by exposing structural dependencies that are not obvious from local code context [20-24]. Nevertheless, legacy systems often provide only incomplete unit tests, sparse documentation, and limited examples of expected output [20,21]. In these settings, direct end-to-end translation is especially risky because the model has few external signals for distinguishing a plausible-looking implementation from a behaviorally correct one. Fine-grained execution alignment has recently improved the localization of translation errors by comparing intermediate program states [25-27], but important practical questions remain. Existing studies often emphasize compilation rate, test success, or translation accuracy, whereas migration projects also depend on developer correction effort, code maintainability, numerical consistency, and the ability to validate behavior when test coverage is incomplete. Another limitation is that most existing approaches treat translation, error repair, and code cleanup as largely separate stages. In enterprise migration, these processes are closely connected [28,29]. Translating irrelevant portions of a large module can introduce unnecessary dependencies, while aggressive refactoring during initial translation may obscure whether an error originates from semantic conversion or code restructuring. Program slicing can reduce this problem by identifying statements, variables, and dependencies that directly influence a migration target. Semantic translation can then focus on preserving program intent within the relevant slice[30-33], while compiler feedback and behavior checks provide complementary evidence about correctness. Refactoring can be applied after behavioral consistency has been established, allowing modernization to improve readability and maintainability without unnecessarily changing validated program logic. Integrating these stages into a coordinated workflow may therefore provide a more reliable strategy than treating code translation as a single generation step[34-36].

This study develops TransAgent, a compiler-guided framework for legacy code migration that integrates program slicing, semantic translation, compiler-based diagnosis, behavior-equivalence checking, and post-translation refactoring. The framework is designed to address both local translation errors and system-level inconsistencies that arise from dependencies, outdated APIs, and undocumented implementation rules. Program slicing is used to isolate behaviorally relevant code and reduce unnecessary translation context. Semantic translation then converts the selected program logic while retaining data dependencies and control relationships. Compiler feedback is used iteratively to identify syntax, type, interface, and dependency errors, while behavior-equivalence checks compare the source and migrated implementations using available inputs, outputs, and execution results. Refactoring is subsequently applied to improve readability and maintainability after functional consistency has been established. TransAgent is evaluated on 186 enterprise modules using behavior preservation, migration effort, numerical accuracy, and maintainability as the principal evaluation dimensions. By combining structural analysis, iterative repair, and explicit behavior validation, this study aims to move automated code migration beyond syntactically correct translation toward reliable preservation of software functionality. The broader significance of the work is to reduce the manual effort required for legacy-system modernization while improving confidence that migrated software retains the operational rules and computational behavior on which enterprise applications depend.

2. Materials and Methods

2.1. Sample and Study Setting

The dataset included 186 enterprise software modules from inventory, billing, customer-service, and reporting systems. The source code contained 214,000 lines of legacy Java and procedural Python, with 3,420 archived change requests. Java modules were migrated to current Java services. Procedural Python scripts were migrated to TypeScript services. A module was included when its source files, build settings, dependency records, and business purpose were available. It also needed regression tests, archived input-output records, or both. Modules with missing files, unavailable external interfaces, or incomplete execution records were not used. The final set included data-processing rules, financial calculations, file handling, exception logic, database access, and external service calls.

2.2. Experimental Design and Control Conditions

The experimental system was TransAgent. It used five agents for program slicing, semantic translation, compiler feedback, behavior checking, and refactoring. The control system was a direct LLM translator[37-39]. It received the same source code and migration request but translated the full module in one step. Both systems used the same base model, context limit, source files, target-language rules, and execution environment. The original target code was not included in either system input. TransAgent was allowed two repair rounds after compiler or

behavior checks. The direct translator was evaluated from its first complete output. Each translated module was compiled and tested in an isolated target-language environment. This comparison measured whether slicing, feedback, behavior checks, and refactoring improved migration quality[40-43].

2.3. Measurement Methods and Quality Control

Regression tests and generated boundary cases were used to check behavior preservation. The source and translated programs were compared by output value, returned data structure, exception type, and error message. Financial modules were tested with normal, boundary, negative, and high-value inputs. Java and TypeScript compilers checked syntax errors, type mismatches, missing imports, unreachable code, and unsupported library calls. Manual correction time was recorded from the first generated version until the code passed the required checks. Maintainability was measured by code size, cyclomatic complexity, comment coverage, naming consistency, and module coupling. Circular dependencies were identified from the target dependency graph. Compiler versions, library versions, test data, and runtime settings were kept the same for all modules. Two software engineers independently reviewed a random sample of translated modules. Differences were discussed and resolved together.

2.4. Data Processing and Model Formulation

The source files were parsed into syntax trees and dependency graphs before translation. The slicing agent divided each module into control-flow units, data-processing units, and external-interface units. File paths, private identifiers, customer information, and service endpoints were replaced with neutral labels before model input. Candidate translations were scored by behavior preservation, compilation status, maintainability, and expected manual editing effort:

$$\operatorname{argmax}_{c \in C} [\alpha B(c) + \beta K(c) + \gamma M(c) - \delta E(c)]$$

The weights α, β, γ , and δ represented behavior preservation, compilation success, code maintainability, and editing effort. For numerical outputs, behavior preservation rate was calculated as:

$$BPR = \frac{1}{N} \sum_{i=1}^N I(|\hat{y}_i - y_i^*| \leq \tau_i)$$

In this equation, $\hat{y}_i$ was the translated-program output, y_i^* was the source-program output, and τ_i was the allowed difference. Text outputs, exception types, and Boolean results were checked by exact matching.

2.5. Evaluation Metrics and Statistical Analysis

The main measures were behavior preservation rate, compilation success rate, manual correction hours per thousand lines of code, maintainability-index change, circular-dependency reduction, and numerical deviation in financial modules. A translated module was counted as

behavior-preserving only when it passed all required regression tests and boundary cases. TransAgent and the direct translator were tested on the same module set. Results were reported as mean values or percentages for all modules. Confidence intervals were calculated by bootstrap resampling at the module level. Paired compilation and behavior results were compared with McNemar’s test. Manual correction time, maintainability change, and numerical deviation were compared with the Wilcoxon signed-rank test. A two-sided significance level of 0.05 was used.

3. Results and Discussion

3.1. Behaviour Preservation after Translation

After two repair rounds, TransAgent preserved the reference outputs in 88.6% of the migrated modules. The direct LLM translator reached 71.9%. The improvement was 16.7 percentage points. This shows that compiler feedback and behaviour checks improved translation quality. Direct translation often produced readable code, but some modules changed exception handling, data conversion, or control-flow conditions [25,26]. As shown in Fig. 1, the proposed process checked code fragments before the full module was completed. Earlier project-level studies also used code partitioning and output checks. However, their language pairs, datasets, and evaluation units were different.

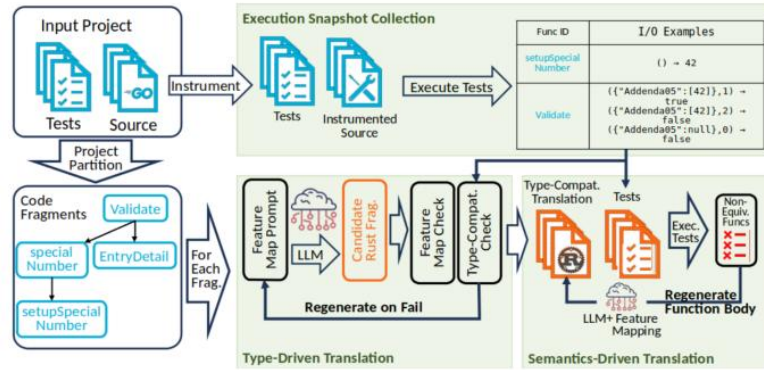


Fig. 1. Steps for code translation, compiler checks, output tests, and code repair.

3.2. Effect of Compiler Feedback on Migration Effort

Manual correction time decreased from 2.8 to 1.4 h per thousand lines of code. Compiler feedback removed many basic errors before manual review. These errors included missing imports, invalid type conversions, unsupported library calls, and unreachable statements. Compilation success alone did not confirm correct program behaviour. Some modules compiled but returned incorrect results for boundary inputs. The behaviour-equivalence agent addressed this problem by comparing outputs from the source and translated programs after each repair round. Earlier compiler-guided methods mainly focused on syntax and type errors. The present results show that compiler feedback is more useful when it is combined with execution tests [27,28].

3.3. Maintainability and Dependency Improvement

The maintainability index increased by 16.2 points, while circular dependencies decreased by 31.5%. The translated code had clearer names, smaller modules, explicit exception handling, and simpler dependency paths. Direct translation often retained the tightly connected structure of the legacy code. The refactoring agent separated data-processing logic from external-interface code and removed repeated dependencies [29,30]. Fig. 2 shows that repository-level migration must consider dependencies and test files, not only individual functions. A local translation may appear correct, while imports, class links, or project settings remain inconsistent. Therefore, maintainability should be assessed together with functional correctness.

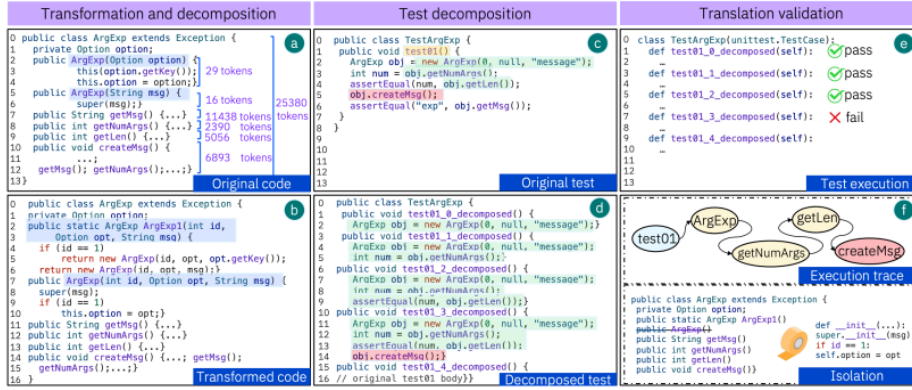


Fig. 2. Main issues in repository migration and the checks used to address them.

3.4. Financial Calculations, Limitations, and Practical Use

For financial calculation modules, 94.3% of test cases had numerical deviations below 0.1%. This is important because small calculation errors may affect billing, inventory values, or customer balances. The remaining 5.7% of cases still required manual review. These cases mainly involved implicit rounding rules and old data formats. The evaluation included 186 enterprise modules and two migration routes: legacy Java to modern Java services and procedural Python to TypeScript services. The findings may not apply to COBOL, C++, distributed systems, or modules that depend on unavailable external services. Future work should use larger industrial datasets and include tests for concurrent programs and database operations.

4. Conclusion

Compiler-guided translation improved the migration of legacy software in the present experiments. TransAgent split each module into smaller code units, translated them, used compiler messages to correct errors, compared source and target outputs, and then cleaned the final code. It retained the expected outputs for 88.6% of modules, compared with 71.9% for direct LLM translation. Manual correction time fell from 2.8 to 1.4 h per thousand lines of code. The maintainability index increased by 16.2 points, and circular dependencies decreased by 31.5%. In financial modules, 94.3% of test cases had numerical differences below 0.1%. The results show

that compiler checking alone is not enough; output testing is also needed to keep the original business rules. The method can support the update of old Java and Python systems. The study was limited to 186 modules and two migration paths. Later work should test larger projects, more languages, concurrent code, databases, and external services.

References

1. Wu, C., & Chen, H. (2025). Research on system service convergence architecture for AR/VR system.
2. Mini Han Wang , " AI-Powered Innovations in Ophthalmic Diagnosis and Treatment ", Bentham Science Publishers (2025). <https://doi.org/10.2174/97988988122871250101>.
3. Zheng, J., & Makar, M. (2022). Causally motivated multi-shortcut identification and removal. *Advances in Neural Information Processing Systems*, 35, 12800-12812.
4. Xu, D., Gui, H., & Chen, H. (2026, March). Research on Layered Control and Fault Recovery Mechanisms for Fast Charging Safety Diagnosis of High-Voltage Battery Systems Under Charging Network Interoperability Conditions. In *2026 3rd International Conference on Electrical Technology and Automation Engineering (ETAE)* (pp. 669-672). IEEE.
5. Quan, Q., Gao, Y., & Wang, Q. (2025). The impact of teacher emotional support on students' engagement in AI-mediated English learning environments: The mediating role of resilience and self-efficacy. *Acta Psychologica*, 260, 105766.
6. Zhang, Z. (2026). A Study on the Impact of Cost Presentation on Decision-making Bias and Cancellation Behavior in Online Subscriptions. Available at SSRN 7349839.
7. Qi, C., & Qiao, X. (2026). Building and Operating a Large Scale Multi-Agent System: A Case Study from Industry. Available at SSRN 6795198.
8. Du, Y., Liu, Q., Dong, Y., & Chen, Z. (2026). Cross-Domain Transfer and Few-Shot Recognition of Defect Images in Complex Industrial Settings.
9. Wang, Z., Yu, J., Liu, H., Zheng, Z., Jin, Y., Li, S., ... & Zhang, K. (2025, July). Generative Music Models' Alignment with Professional and Amateur Users' Expectations. In *Findings of the Association for Computational Linguistics: ACL 2025* (pp. 6909-6920).
10. Xiong, W., Guo, Y., & Zeng, Y. (2026). A Study on the Energy Efficiency of MEP Systems and Coordinated Dispatch Mechanisms for Multi-energy Systems in High-density Urban Buildings. Available at SSRN 7120518.
11. Zhao, J., Fan, J., & Li, L. (2026). A Study on an Explainable Causal-Enhanced LLM Agent for Predicting the Forming Quality of Automotive Component Materials.
12. Jiao, Y., Shi, T., Zhao, B., & Wang, A. (2026). Retrieval-Guided Structured Reasoning and Interpretable Representation Learning for Large-Scale Video–Language Models.
13. Xu, Y., Cui, J., Zhu, Z., Shang, H., Luan, S., Xu, M., ... & Zhu, S. (2026). Wam-flow: Parallel coarse-to-fine motion planning via discrete flow matching for autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 24918-24928).

14. Liang, S., Du, Y., Chen, W., & Liu, Z. (2026). Order Allocation and Emergency Transportation Decisions Amid Multi-Source Procurement Disruptions.
15. Hong, Z., Liang, S., Xu, T., & Chen, H. (2026). A Study on Failover Verification and Recovery Objective Prediction for Cross-Region Cloud Services.
16. Huang, J., Yin, J., Xu, T., & Yang, J. (2026). Quality Assessment and User Adoption Prediction for Enterprise-Level Financial Reporting Data Products. Available at SSRN 7179118.
17. Zheng, X., Chen, X., Gong, S., Griffin, X., & Slabaugh, G. (2025, September). Xfmamba: Cross-fusion mamba for multi-view medical image classification. In *International Conference on Medical Image Computing and Computer-Assisted Intervention* (pp. 672-682). Cham: Springer Nature Switzerland.
18. Zhang, Z., Wang, J., Li, Z., Wang, Y., & Zheng, J. (2025). Annocoder: A mti-agent-based code generation and optimization model. *Symmetry*, 17(7), 1087.
19. Liang, S., Hsu, K. H., & Liu, Z. (2026). Rolling Scheduling and Capacity Balancing for Server Assembly Under Engineering Change Disturbances.
20. Bao, Y., & Wang, H. (2026). Large-scale Metrics Migration: A Systematic Approach to Rebuilding Production Measurement Infrastructure. Available at SSRN 7185660.
21. Su, J., He, C., Zhu, F., Xu, X., Guan, D., & Si, C. (2024, September). Hpff: Hierarchical locally supervised learning with patch feature fusion. In *European Conference on Computer Vision* (pp. 293-309). Cham: Springer Nature Switzerland.
22. Zhao, Z., & Welsch, R. E. (2026). Point-in-Time Financial RAG with Frozen LLMs and Market-Feedback Adaptive Retrieval. *arXiv preprint arXiv:2605.31201*.
23. Yuan, Y., Huang, J., Yin, J., & Xu, T. (2026). Confidence Calibration and Semantic Error Analysis for Ambiguous Coreference Resolution in User-generated Social Media Text. Available at SSRN 7393238.
24. Lyu, Y. (2025). Enhancing Approximate Conformance Checking Accuracy with Hierarchical Clustering Model Behaviour Sampling. *International Journal of Advanced Computer Science & Applications*, 16(8).
25. Bao, Y., & Wang, H. (2026). Orion: A High-Throughput, Fault-Tolerant Event Routing Architecture for Hyperscale Data Streams. *Fault-Tolerant Event Routing Architecture for Hyperscale Data Streams* (January 01, 2026).
26. Chaoyong Ma, Nan Si, Xuewei Song, Chen Liang and Haoming Luo, A Bearing Fault Diagnosis Method Based on Concise Empirical Wavelet Transform, *International Journal of Comprehensive Engineering*, 2023, 12(1), 1-6
27. Wang, S., Feng, Y., & Fang, X. (2026, May). A Large Language Model-Enabled Multi-Agent Collaboration Method for Complex Task Solving. In *2026 6th International Symposium on Computer Technology and Information Science (ISCTIS)* (pp. 253-256). IEEE.

28. Liang, J., Xiao, Y., Yang, H., Yu, Z., Liu, J. N., & Yang, K. (2026, April). C-HyPOD: Causal Hyperbolic Representation Learning with Prototype Orthogonal Disentanglement for Graph Out-of-Distribution Recommendation. In *Proceedings of the ACM Web Conference 2026* (pp. 6541-6550).
29. Xi, X., Chen, P., Qiu, Y., Luo, R., Tong, P., & Liang, J. Video-BCI: Bayesian Cognitive Integration of Self-Prior Hypotheses for Video Understanding. In *Forty-third International Conference on Machine Learning*.
30. Yang, Q., & Du, Y. (2026). Predicting Rollback Risks and Controlling Gradual Rollout Traffic for Online Ranking Feature Deployments.
31. Xu, Z., Zhang, X., Li, R., Tang, Z., Huang, Q., & Zhang, J. (2024). Fakeshield: Explainable image forgery detection and localization via multi-modal large language models. *arXiv preprint arXiv:2410.02761*.
32. Huang, Q., Xu, Z., Zhang, X., & Zhang, J. (2025). UniShield: An Adaptive Multi-Agent Framework for Unified Forgery Image Detection and Localization. *arXiv preprint arXiv:2510.03161*.
33. Jiang, C. (2026). Information doesn't work? Testing experiential drivers of VR application in heritage tourism by affective-cognitive pathways. *Journal of Hospitality and Tourism Technology*, 1-28.
34. Yuan, Y., Yin, J., Huang, J., & Xu, T. (2026). Weakly Supervised Anomaly Detection and Privacy Risk Scoring in Community Message Threads.
35. Xi Qiao, Di Zhao, Nan Si, Jifeng Sui and Yonggang Xu, Weighting Spectrum Editing Based on Amplitude Mapping and Its Application in Composite Fault Diagnosis of Rolling Bearings, *International Journal of Comprehensive Engineering*, 2024, 13(1), 13-21
36. Xu, Z., Zhang, X., Zhou, X., & Zhang, J. (2025). AvatarShield: Visual Reinforcement Learning for Human-Centric Video Forgery Detection. *arXiv preprint arXiv:2505.15173*.
37. Wang, T., & Xia, Z. (2025). Stability of In-Context Learning: A Spectral Coverage Perspective. *arXiv preprint arXiv:2509.20677*.
38. Liu, T., He, J., Zhou, R., Song, J., Liu, H., & Liang, Y. (2025). Revolutionizing clinical decision making through deep learning and topic modeling for pathway optimization. *Scientific Reports*, 15(1), Article 28787. <https://doi.org/10.1038/s41598-025-12679-z>
39. Wu, X., Xing, X., Yao, J., Qian, Q., & Song, J. (2025). Scnet: spectral convolutional networks for multivariate time series classification. *Applied Intelligence*, 55(6), Article 456. <https://doi.org/10.1007/s10489-025-06352-1>
40. Xixin Chen, Ming Tu, Lei Su and Ke Li, Rolling Bearing Fault Diagnosis Based on Digital Twin Method, *International Journal of Comprehensive Engineering*, 2024, 13(1), 22-39
41. Ma, C., Song, J., Xu, Y., Fan, H., Wu, X., & Sun, T. (2024). Vehicle-Based Machine Vision Approaches in Intelligent Connected System. *IEEE Transactions on Intelligent Transportation Systems*, 25(3), 2827-2836. <https://doi.org/10.1109/TITS.2023.3276325>

42. Bing Wang, Haihong Tang, Xiaojia Zu and Wuwei Feng, Application of Parameter Adaptive VMD in Bearing Fault Diagnosis, International Journal of Comprehensive Engineering, 2023, 12(1), 34-42