COLOMBO SCIENTIFIC PUBLISHING

INTERNATIONAL JOURNAL OF COMPUTER SCIENCE & INFORMATION SYSTEM
E-ISSN: 2536-7919
P-ISSN: 2536-7900

DOI: - https://doi.org/10.55640/ijcsis/Volume11Issue05-02 **PAGE NO: 13-21**

Strategies for Migrating Monolithic Enterprise Applications to Modular Microservices Architecture

Abdukhalimov Abduaziz

Senior Full Stack Developer, Barso LLC, Remote Penslvania, USA

ARTICLE INFO

Article history:

Submission: February 18, 2026
Accepted: March 25, 2026
Published: May 07, 2026
VOLUME: Vol.11 Issue 05 2026

Keywords:
monolithic architecture, microservices migration, enterprise applications, ERP modernization, bounded contexts, database per service, observability, microservices testing, architectural anti-patterns, legacy modernization.

ABSTRACT

Enterprise ERP systems still rely on large monoliths that bind finance, HR, reporting, and operations to a single codebase and database. That structure slows releases, widens regression risk, and spreads failures across unrelated functions. This review synthesizes recent research on migrating to a modular microservices architecture, with attention to migration sequence, service decomposition, data ownership, and post-migration control. The source base comprises 10 peer-reviewed publications from 2021 to 2025, including systematic reviews, literature reviews, case studies, and a peer-reviewed conference paper. Comparative source analysis, conceptual synthesis, typologization, and analytical generalization were used. The reviewed literature converges on four conditions of stable migration: phased transition, business-aligned bounded contexts, explicit persistence boundaries, and runtime control through observability and testing. On that basis, the article formulates a decision model for enterprise teams that need to modernize large business systems while preserving operational continuity, data reliability, and manageable coordination costs.

Introduction

Large enterprise applications still concentrate financial, personnel, reporting, and operational logic within a single codebase and a single persistence layer. In ERP environments, such an architecture often persists for years because it supports core business processes and accumulated domain rules. The same structure creates delivery drag. A local code change can trigger broad regression testing, a data schema adjustment can affect several departments at once, and a failure in one module can surface far outside its functional area.

A typical ERP monolith illustrates this structure with particular clarity. Finance, HR, reporting, and operational routines often share one database and accumulate years of cross-module dependencies. Under such conditions, a local modification in payroll logic can affect reporting output, a schema change in finance can trigger failures in adjacent functions, and each release cycle expands because teams have to retest flows that are only indirectly related to the initial change. This enterprise scenario provides a practical framework for understanding why migration decisions in large business systems must address module coupling, persistence ownership, communication style, and release risk simultaneously.

Migration to microservices entered this space as a response to those pressures, yet practice has shown that service extraction alone does not secure a better architecture. Teams split code, but cross-service chatter remains high; they separate deployment units but still depend on a single shared database; they adopt

asynchronous flows, but discover that data ownership was never clarified in the first place. The migration problem lies within architectural boundaries, data flows, and operational controls simultaneously.

This study aims to develop an analytical framework for migrating monolithic enterprise applications to a modular microservices architecture. Three research objectives shape that framework. The first objective is to identify migration sequences that recur across recent studies and determine which sequence best fits enterprise systems with live business workloads. The second objective is to examine how decomposition logic and persistence design influence service autonomy. The third objective is to formulate a practical post-split control model focused on runtime visibility, testing, and structural drift.

The study contributes an integrated reading of four bodies of literature that often remain separated in publication practice: modernization process studies, service extraction research, data migration work, and operational governance studies on observability, testing, and anti-patterns. That integrated view fits ERP modernization, where business capability boundaries intersect with transaction discipline, reporting demands, and team coordination. The working hypothesis is that migration becomes sustainable when enterprise teams redesign service boundaries, persistence ownership, and control loops within a coordinated program.

MATERIALS AND METHODS

The source base comprises 10 peer-reviewed publications published between 2021 and 2025, selected for their direct relevance to monolith modernization, microservice reengineering, service boundary formation, database migration, observability, testing, and anti-pattern detection. The corpus combines systematic mapping studies, literature reviews, empirical migration papers, and one peer-reviewed conference publication on domain-driven migration with hybrid database design [1–10]. Screening followed three filters: publication recency, direct relevance to enterprise-scale migration, and analytical value for a review article without an experimental section. The resulting corpus covers four clusters of questions: migration sequencing and modernization workflow [2; 3; 7; 9], service identification and boundary logic [7; 8; 10], persistence redesign and database transition [6; 8; 9], and post-migration governance through anti-pattern control, observability, and testing [1; 4; 5].

The study uses comparative analysis of recent publications, source analysis, conceptual synthesis, typologization of migration paths, and analytical generalization. Those methods connect the selected sources with the three research objectives and support a review-based implementation model for enterprise modernization.

RESULTS

Recent studies describe migration as a staged architectural program. Systematic mappings and reviews show that migration decisions accumulate across several layers: decomposition, communication style, persistence redesign, operational tooling, and release governance [2; 7; 9]. One mapping of legacy-system modernization organizes transition work into macro-activities that start with analysis and proceed through decomposition, development, execution, and operation [2]. A broader review of reengineering approaches groups the field into static, dynamic, hybrid, and artifact-driven approaches and notes that iterative transition dominates current practice [7]. A migration-focused mapping study reaches a similar conclusion, reporting that the field still focuses on decomposition while giving less attention to tools, communication design, and database transition [9]. A case-based study refines that picture by showing that a modular monolith can serve as a useful intermediate state because boundary flaws appear early, before network distribution multiplies them [3]. Across these studies, the same pattern appears. Teams reduce migration risk when they sequence the work, expose boundary problems early, and postpone large-scale cutover until service seams survive real change.

Among the migration routes discussed in recent literature, the Strangler Fig pattern offers a particularly convincing fit for enterprise systems, particularly through progressive replacement. Teams extract selected business capabilities, gradually route traffic, and keep the remaining monolith alive while new service boundaries are tested under production-like conditions. That sequence reduces exposure to broad regression and gives architects time to verify whether extracted functions are autonomous or still depend

on hidden shared models. In ERP settings, this route is relevant because finance, HR, and reporting do not usually separate cleanly in a single move. A phased replacement path creates room for architectural correction before distribution turns local design flaws into system-wide operational problems.

Figure 1 condenses that review pattern into a migration sequence suited to enterprise monoliths.

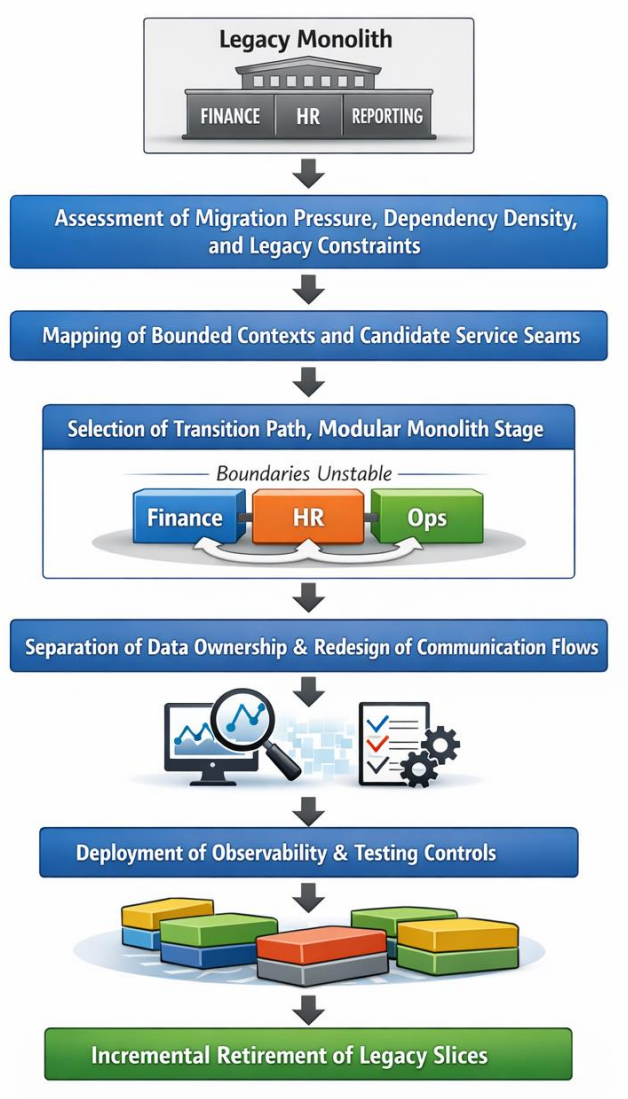

Figure 1. Review-based migration sequence for enterprise monolith modernization, adapted from [2; 7], and [9].

The second cluster of findings concerns service decomposition. The reviewed literature does not support a split driven only by technical layers, package structure, or deployment convenience. Service boundaries hold better when teams align them with business capabilities and bounded contexts [7; 8]. The domain-driven migration paper by Ng et al. links bounded contexts to a hybrid database design and uses this combination to preserve continuity during transition [8]. Automated extraction research points in the same direction from another angle. Sellami et al. formulate microservice identification as a multi-objective optimization problem and evaluate candidate cuts through cohesion, coupling, granularity, and semantic dependence [10]. Their work improves boundary discovery, yet it does not remove the need for expert judgment. Enterprise ERP systems explain why. Finance, HR, and reporting often share audit rules, reference entities, and cross-domain reads that a static structure alone cannot capture. A decomposition that looks clean in code can still cut through one business responsibility and relocate coordination problems into runtime traffic. Recent studies support a combined approach. Structural techniques and search-based methods can generate candidate boundaries, while domain reasoning decides which candidates deserve organizational ownership and transactional autonomy [7; 8; 10].

Three decomposition logics recur in practice, and each leads to a different migration outcome. Domain-

driven decomposition groups services around business capabilities and bounded contexts, which makes ownership, change responsibility, and transaction boundaries easier to define. Functional decomposition starts from major business functions and can work well at an early planning stage. However, it often remains too coarse for long-term service design if shared subdomains stay unresolved. Technical decomposition follows layers such as user interface, business logic, or data access. That route often looks convenient during the first split because it mirrors the internal structure of the monolith, yet it tends to preserve the old dependency pattern in a distributed form. For ERP applications, the literature gives the strongest support to domain-driven decomposition, while functional decomposition can serve as an intermediate analytical lens and technical decomposition usually reproduces coupling in a more expensive runtime environment.

Persistence design introduces another line of convergence across the reviewed corpus. Migration from one monolithic database to service-level persistence changes the engineering problem from local ACID control to cross-service coordination, compensation, and read-model design [6; 8; 9]. The migration mapping by Martínez Saucedo et al. lists database migration among the most recurrent pain points in the field [9]. Kazanavičius et al. focus on that problem directly and propose a migration route from monolithic persistence to multi-model polyglot storage aligned with microservice requirements [6]. Ng et al. extend the same line of thought through hybrid database design and eventual consistency as transitional mechanisms in a domain-driven migration program [8]. These studies place database-per-service at the center of service autonomy. Once one team owns one persistence boundary, hidden coupling becomes harder to maintain. The literature suggests that systems with dense data interdependence often need a transitional period in which ownership rules, message flow, and consistency windows are formalized before full persistence isolation is feasible [6; 8; 9].

The literature on data transition also points to a recurring architectural mistake: the persistence layer is nominally split while services continue to depend on shared tables, shared schemas, or direct cross-service data access. This shared database pattern keeps old coupling alive and weakens service ownership even after teams introduce separate deployment units. A more stable transition path relies on database-per-service together with coordination mechanisms that make distributed consistency explicit. Saga patterns address multi-step business processes that span several services and replace one large transaction with a chain of local transactions and compensating actions. The Outbox pattern reduces the risk of lost or duplicated domain events by aligning state changes and message publication within a single service boundary. Selective event sourcing can add value in domains where auditability and state reconstruction justify the extra complexity. At the same time, eventual consistency remains acceptable only when teams define reconciliation windows, failure handling, and read-model refresh rules in advance.

A further finding concerns false modularity. The anti-pattern literature shows that poor microservice design has already produced a wide vocabulary of recurring failures, many of them tied to migration and team organization [1]. Cerny et al. catalog 58 anti-patterns derived from 203 originally identified cases and classify them across several categories, including system migration and monitoring practices [1]. The case-based migration study by Faustino et al. shows that performance penalties and data consistency side effects can emerge during the modular monolith stage itself [3]. The migration mapping by Martínez Saucedo et al. reports similar pressure points at the field level, around service communication and database transition [9]. Read together, those studies clarify why enterprise teams so often end up with a distributed monolith. The architecture appears decomposed from the outside, yet services still make excessive synchronous calls, depend on legacy shared models, or require cross-domain coordination for a routine business change. In ERP systems, that pattern tends to surface first around reporting chains, cross-service reads, and transaction-heavy domains that were split before their consistency model was redesigned.

Communication design determines whether the new architecture gains autonomy or only relocates monolithic coupling into network traffic. A hybrid model often offers the most balanced route for enterprise migration. Synchronous REST calls fit request-response interactions that require immediate confirmation, controlled latency, and clear interface contracts. Asynchronous event-driven exchange through platforms such as Apache Kafka is well-suited for domain events, state propagation, integration with downstream consumers, and workflows, where temporal decoupling reduces coordination pressure. Problems begin when teams overuse synchronous internal calls and create chatty services, because a single business operation then becomes a long chain of fragile network interactions. The same risk appears when

asynchronous communication is introduced without event ownership rules, idempotency discipline, and error recovery logic. Recent research and enterprise practice both indicate that communication style should follow domain semantics, consistency requirements, and failure tolerance.

Observability and testing close the gap between design intent and runtime behavior. The observability mapping study by Gomes et al. shows that the field has moved beyond basic monitoring and now treats observability through taxonomies of tools, datasets, classifications, and unresolved research problems [4]. The testing mapping by Hui et al. identifies nine categories of microservice testing methods and highlights unresolved gaps in communication-heavy scenarios [5]. Teams need evidence about request paths, trace completeness, interface stability, reconciliation lag, and deployment side effects. Eventual consistency, distributed transactions, and service-level persistence remain fragile until teams can observe their behavior across real workflows [4; 5; 8]. Testing has the same function. Contract testing, integration testing, and communication-aware testing expose boundary errors that decomposition logic alone cannot eliminate [5].

The operational layer adds another design burden once services move from conceptual boundaries to live deployment. Containers and orchestration platforms such as Docker and Kubernetes support repeatable packaging, rollout control, scaling, and consistent environments. Yet, they also raise the cost of runtime coordination if service boundaries remain weak. API gateways and service discovery mechanisms help manage entry points, routing, and inter-service lookup, though they do not correct flawed decomposition on their own. Observability has to work across three linked planes: metrics, logs, and traces. Teams often combine Prometheus and Grafana for metric collection and dashboarding, ELK-based pipelines for centralized log analysis, and Jaeger for distributed tracing across service calls. This stack gives engineers the diagnostic visibility needed to detect latency growth, cascading failures, broken contracts, and hidden coupling after the split.

Comparison across four strands of literature sharpens the answer to the third research objective. Modernization studies explain where migration work begins [2]. Reengineering reviews classify the methods teams use to split and reshape legacy systems [7]. Service extraction research supplies formal criteria for candidate boundaries [10]. Migration mappings identify the points at which practice breaks down, in communication and persistence [9]. Operational studies on anti-patterns, observability, and testing explain how teams prevent the target architecture from slipping back into hidden dependencies after the split [1; 4; 5]. Enterprise migration stabilizes when service boundaries, data ownership, and runtime controls mature together. Isolating one of these lines while postponing the others only shifts the problem from code to operations.

DISCUSSION

Enterprise migration needs a threshold test before any service extraction starts. Distribution pays off only after the expected reduction in coordination inside the monolith exceeds the extra burden of network calls, service ownership, deployment pipelines, and runtime diagnosis. In ERP environments, that threshold rarely appears across all modules simultaneously. Read-heavy capabilities, external integration points, and domains with clearer ownership lines usually form better early candidates than transaction-heavy cores that still rely on shared invariants and dense reporting dependencies.

Table 1 compares four migration routes that often appear in enterprise planning. The comparison focuses on fit, operational exposure, and the control burden each route imposes during transition.

Table 1. Migration path selection logic for enterprise applications

Criterion	Big Bang rewrite	Strangler-style phased extraction	Branch by Abstraction	Modular monolith first
Best fit	Small systems with limited coupling and low continuity risk	Large live systems that must keep serving users during transition	Tightly intertwined code that needs internal substitution behind stable interfaces	Large monoliths with unclear future service boundaries
Main advantage	Clean target design on paper	Progressive rollout with localized rollback options	Continued feature delivery during internal replacement	Boundary testing before network distribution
Main exposure	Concentrated delivery risk and broad regression surface	Long coexistence period with governance fatigue	Temporary abstraction debt and duplicated logic	Performance penalties can appear before service extraction
Data transition profile	Full redesign in one program	Ownership transfer by slice	Shared persistence often survives longer	Better preparation for later persistence isolation
Coordination burden	Highest at cutover	Sustained across the migration window	High inside the codebase	High during refactoring, lower after stable seams emerge
Recommended use	Narrow-scope replacement under strong control conditions	Default route for business-critical enterprise monoliths	Suitable for modules that cannot pause product change	Suitable when teams still need to prove domain seams

This comparison points to one practical rule. Boundary confidence matters more than architectural preference. Where ownership lines are already clear, phased extraction gives the best balance between continuity and structural progress. Where service seams remain speculative, the modular monolith approach buys learning time and reduces the risk of poor boundary decisions. Branch by Abstraction remains useful within codebases that must change during migration. Big Bang replacement belongs to rare cases with a narrow scope, low continuity pressure, and unusually strong delivery control.

This planning logic becomes more concrete when common migration failures are mapped to corrective mechanisms. Residual coupling after the first split usually indicates that teams extracted services before clarifying bounded contexts and data ownership. Distributed inconsistency tends to surface when eventual consistency is adopted without compensation logic, reconciliation rules, and message publication discipline. Excessive internal API traffic often points to technical decomposition or over-fragmented service cuts. Rising operational overhead usually appears when the number of services grows faster than the team's

ability to manage deployment, tracing, contracts, and failure recovery. A phased migration path addresses these pressures only when each architectural decision is paired with a control mechanism. Bounded contexts reduce residual coupling. Database-per-service, Saga, and Outbox structures provide distributed consistency, while hybrid communication prevents unnecessary synchronous chains. Observability and contract testing keep service growth from turning into opaque operational drag.

A second planning issue concerns measurement. Enterprise teams often describe migration success through generic promises about scalability, resilience, or agility. Teams need signals that expose whether boundaries hold under normal delivery pressure. Table 2 translates that need into a monitoring matrix for post-split control.

Table 2. Monitoring matrix for post-migration architectural control

Control dimension	What to measure	Early warning signal	Likely architectural interpretation	Immediate corrective move
Boundary health	Cross-service synchronous hop count per business transaction	Rising call depth in stable workflows	Services were cut too finely or along technical layers	Revisit service seams and merge over-fragmented flows
Data integrity	Reconciliation lag and unresolved compensation events	Persistent drift between source and downstream views	Consistency windows remain under-governed	Tighten ownership rules, retry policy, and compensation logic
Release flow	Lead time, failed deployments, and rollback frequency	Release cadence slows after additional services are introduced	Operational overhead grows faster than autonomy	Simplify pipelines and reduce cross-service release coupling
Runtime diagnosis	Trace completeness, log correlation, mean time to localize failure	Failures require manual reconstruction across services	Telemetry is too weak for distributed behavior	Enforce trace propagation and unified observability standards
Contract stability	Consumer-driven contract failures and interface break rate	Frequent downstream breakage after small changes	Public interfaces remain unstable, or ownership is diffuse	Freeze contracts and tighten version discipline
Team alignment	Number of cross-team changes needed for one business feature	One feature repeatedly spans several service owners	Architectural boundaries do not match work ownership	Re-map ownership or merge services that change together

The matrix clarifies a recurrent problem in enterprise programs. Many migration failures appear to be

infrastructure issues during incident response, even though their root causes lie in incorrect service cuts, weak ownership rules, or incomplete transition governance. A growing number of internal API hops usually indicates over-decomposition. Reconciliation lag points to poor consistency design. Slow-release flow signals that service count has risen faster than true autonomy. Runtime evidence needs to feed architecture decisions throughout the program.

Operational complexity deserves separate attention because service growth changes the engineering team's work model. Release orchestration, incident diagnosis, environment drift, schema evolution, contract control, and access management all become more demanding as services multiply. For that reason, migration should be evaluated based on whether the new architecture reduces coordination during business change while maintaining operational control within the team's capacity to run it.

A workable implementation model follows a sequence of decisions. Teams start by mapping bounded contexts and rating them by transactional intensity, reporting centrality, regulatory exposure, and dependency density. They then restructure the monolith internally until the codebase reflects those domain seams with enough stability to support ownership decisions. Service extraction begins with domains that have fewer transactional entanglements or stronger external integration pressure. Persistent ownership moves in parallel with service extraction, because shared data access keeps old coupling alive even after deployment units multiply. Teams shift transaction-heavy domains later, after observability, contract testing, and reconciliation control are already routine. The service count itself needs governance. A smaller set of well-bounded services creates less operational drag than a visually granular landscape that still changes in clusters.

CONCLUSION

Recent research treats enterprise migration as a sequenced modernization program built around analysis, boundary formation, and staged cutover. That reading supports phased transition paths for most live enterprise monoliths and reserves one-step replacement for narrow, tightly controlled cases.

Service autonomy depends on the way teams draw boundaries and redesign persistence. The reviewed corpus supports bounded contexts, expert-guided decomposition, and explicit data ownership as the strongest basis for modular migration. Search-based extraction and formal decomposition criteria help teams discover candidate seams, but domain judgment determines which seams can withstand organizational and transactional pressure.

Post-split stability depends on runtime control. Observability, communication-aware testing, and anti-pattern detection prevent migration from slipping into hidden dependencies after the initial extraction. The working hypothesis is confirmed. Sustainable migration emerges when teams reshape service boundaries, maintain ownership, and align operational controls within a single coordinated program.

REFERENCES

1. Cerny, T., Abdelfattah, A. S., Al Maruf, A., Janes, A., & Taibi, D. (2023). Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study. Journal of Systems and Software, 206, 111829. https://doi.org/10.1016/j.jss.2023.111829

2. Fávero, L. F., Almeida, N. R. de, & Affonso, F. J. (2025). A systematic mapping study on the modernization of legacy systems to microservice architecture. Applied System Innovation, 8(4), 86. https://doi.org/10.3390/asi8040086

3. Faustino, D., Gonçalves, N. R., Portela, M., & Rito Silva, A. (2024). Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation. Performance Evaluation, 164, 102411. https://doi.org/10.1016/j.peva.2024.102411

4. Gomes, F. A. A., Rego, P. A. L., & Trinta, F. A. M. (2025). A systematic mapping study on observability of microservices-based applications: fundamentals, classifications, and challenges. Computing, 107(9), 183. https://doi.org/10.1007/s00607-025-01540-w

5. Hui, M., Wang, L., Li, H., Yang, R., Song, Y., Zhuang, H., Cui, D., & Li, Q. (2025). Unveiling the microservices testing methods, challenges, solutions, and gaps: A systematic mapping study. Journal of Systems and Software, 220, 112232. https://doi.org/10.1016/j.jss.2024.112232

6. Kazanavičius, J., Mažeika, D., & Kalibatienė, D. (2022). An approach to migrate a monolith database into multi-model polyglot persistence based on microservice architecture: A case study for mainframe database. Applied Sciences, 12(12), 6189. https://doi.org/10.3390/app12126189

7. Martínez Saucedo, A., Rodríguez, G., Rocha, F. G., & dos Santos, R. P. (2025). Migration of monolithic systems to microservices: A systematic mapping study. Information and Software Technology, 177, 107590. https://doi.org/10.1016/j.infsof.2024.107590

8. Mohottige, T. I., Polyvyanyy, A., Fidge, C., Buyya, R., & Barros, A. (2025). Reengineering software systems into microservices: State-of-the-art and future directions. Information and Software Technology, 183, 107732. https://doi.org/10.1016/j.infsof.2025.107732

9. Ng, T., Bin Rawi, A. A., Sum, C. S., Tso, E., Yau, P. C. Y., & Wong, D. (2024). Migrating from monolithic to microservices with a hybrid database design architecture. In Proceedings of the 2024 9th International Conference on Intelligent Information Technology (pp. 536–541). ACM. https://doi.org/10.1145/3654522.3654602

10. Sellami, K., Ouni, A., Saied, M. A., Bouktif, S., & Mkaouer, M. W. (2022). Improving microservices extraction using evolutionary search. Information and Software Technology, 151, 106996. https://doi.org/10.1016/j.infsof.2022.106996