International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(10s) 2026 pp. 1099-1111
https://doi.org/10.70917/ijcisim-2026-3701
Article

Architecture Mismatch-based Refactoring Recommendations: A Literature Review

Sreeji K S[1], J Vijayarani[2]

[1]Hindustan Institute of Technology and Science, Padur Chennai-603103, Email: sreejiks71@gmail.com
[2] Hindustan Institute of Technology and Science, Padur, Chennai-603103, Email: vijayarj@hindustanuniv.ac.in

Abstract: Software Architecture (SA), which simulates the structure, scalability, and maintainability of applications, is a fundamental characteristic of software development. The characterization of architecturally considerable requirements' greatly impacts the rapid software development lifecycle. Technological developments have increased software complexity. The present research studies have focused on the system requirements mismatch by continuous integration practices. The predominance of refactoring strategies failed to deal with these systems. Consequently, such systems is a failure to performance, incurring high development and maintenance costs. But recommendation systems in refactoring are still in their infancy to detect intelligently, architecture-mismatch-based inconsistencies. The current research of mismatch-based refactoring largely uses static analysis and modularity violation principle- based reflection models which are computationally intensive and not explainable. The proliferated studies on architecture mismatch -based refactoring is focused on rule-based, context-aware based, data-driven and machine-learning based refactoring approaches. However, integrating deep learning strategies will improve timely and quality recommendations. Therefore, the practitioners in industry needs a rigorous literature review on mismatch-based refactoring integrated with deep learning strategies. In this study, we have included several architectural issues discussed in various literatures that affect system maintainability and long-term evolution. According to the rigorous study, the deep learning and quality -based method improved the overall maintainability and evolution of the system. The strategy decreases response time for refactoring recommendations and the overall complexity of the system.

Keywords: Software Architecture (SA), Refactoring Recommendations, Mismatch Detection, Context-Aware Refactoring, Microservices Migration, Quality-Driven Refactoring, and Artificial Intelligence (AI) - Based Refactoring.

1. Introduction

The architecture mismatch term was coined with a group of design choices that impact the composition, external behaviour, and quality of the software system. For further adoptions, reducing these discrepancies plays a prominent role in assessing the overall performance and maintainability of the system [1]. As the complexity of the system gradually exceeds, the architectural decisions, styles and intrinsic PATTERNS INFLUENCE the system. Therefore, the effect incurs a long-term maintenance COST. In SA, the vitality of the system performance is necessary to ensure that they function faultlessly [2]. Nevertheless, software with longer life cycles experiences the destruction of the applied architecture from the intended architecture, thereby affecting the performance of the software, routine maintenance, improvement activities, and SQ factors [3]. The occurrence of architectural mismatches' during system conformation from numerous independent software parts is a technical barrier in SA. architectural mismatches negatively impact SQ attributes. To alleviate the matters, software refactoring is essential for optimizing SQ and confirming the long-term sustainability and viability of complex software systems. It is an important Software Engineering (SE) method utilized for ensuring the maintainability and sustainability of complicated software systems [4]. The quality and sustainability of software have been improved with a recent focus on mismatch-based refactoring recommendations as a more intelligent way of identifying architectural problems and proposing appropriate refactoring actions. In this review, research on SA mismatches and mismatch-based refactoring recommendations within the SE domain is offered.

The key objective of this study is to reconnect the relationship between architecture mismatch-based refactoring and its prioritization to improve software quality and reduce the system complexity. Reconciling mismatches from

large software systems, especially cloud-based, microservice architectures is difficult. There is a lack of automated recommendation systems to

recommend first-quality suggestions in architecture refactoring. Existing studies are computationally intensive and not easily explainable.

Our review is organized as follows: Section 2 explores review of current literature Section 3 explains the summary of the review; Section 4 provides the conclusion with future research.

1.1. RQs AND ARTICLE SELECTION STRATEGY

RQs include:

1. What is the significance of SA Mismatches?

2. What are the prominent contributions in mismatch-based refactoring?

3. which approaches by refactoring are for architectural improvement?

4. What is the relevance of the Mismatch-Based Refactoring Recommendations?

To answer the above questions, a Systematic Literature Review (SLR) method was employed. We gathered the famous repositories such as IEEE Xplore, ACM Digital Library, SpringerLink, Science Direct, Scopus, and Google Scholar, research articles were gathered during the timeframe of 2016 to 2026. According to the inclusion criteria, the review involved only articles that included information on architecture mismatches and recommendation systems for refactoring. The studies that were less relevant to SA, published before 2016, or not written in English were excluded. Moreover, to provide a structured and systematic method of conducting the review, the articles were chosen as according to the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA); - Figure 1 exhibits the flowchart of the process.

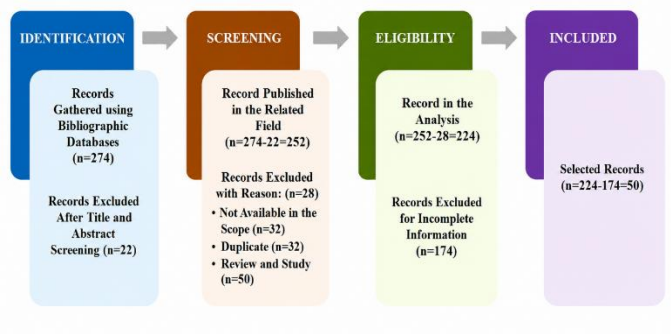


Figure 1: PRISMA framework

2. LITERATURE EVALUATION

2.1. Software Architecture Mismatches

When integrated components have inconsistent, usually implicit, assumptions about their environment, the data binding, or interaction between components, SA mismatches occur [5].

Claudia, *et al* [6] described OSS components, system requirements, and matching and mismatching resolution practices. Semi-structured comprehensive interviews were conducted with twenty-five respondents who had performed Requirements Engineering (RE) activities in software development projects. Their ideologies paved the way in increasing awareness of OSS components for software-intensive firms integrating OSS components. Mismatch resolutions observed might not be applicable directly to safety-critical domains. Grace *et al* [7] explained the Component Mismatches (CMs) as a prominent competor for fielding AI-enabled schemes in the public sector. Machine-readable ML-Enabled System Element Descriptors were utilized as a mechanism to facilitate the detection of mismatches. According to the analysis, identifying 2-3 mismatches in a project was the target, which was identified through automation. Different fields possess completely different descriptive languages and assumptions. Moreover, disjointed systems are often caused by incorporating AI components into broader systems. Shawn *et al* [8] defined hidden dependencies and component variants in an SBOM-centric software composition investigation. Two mismatch

patterns were hidden code-level dependencies and component variants (clones). According to the outcomes,those mismatches led to varying vulnerability reporting and management of VEX statements. The studies were restricted to Java projects. However, the evaluated features and tools are language-agnostic.

- Grace *et al* [9] developed a portrayal and recognition of mismatches in ML-Enabled Schemes. Interviews were conducted to collect and validate common types of mismatches. Seven classifications of ML mismatches, together with 34 subclasses, were identified. Representatives of all practitioners involved in the improvement and deployment of ML-enabled systems were not covered. Hence, the mismatches might not cover all types of of mismatches. Ming *et al* [10] established the behavioural impact of mismatch negativity neuro-feedback's on foreign language learning. To analyse the practical impacts of mismatch negativity, a feasibility study was conducted with a smaller participant group (NF and control groups; 6 participants each). The findings show that, NF led to longer-term learning persisting for a minimum of two months. Marc *et al* [11] quantified the variability mismatch between the issue and solution space. A concept for deriving a solution-space feature model that permitted the reusage of prevailing #SAT analyses for computing was presented. According to the outcomes, the mismatch was confirmed by the derivation scales for the real-time software product lines. Solution-space feature models were not used for direct sampling on the solution space.
- Cristina *et al* [12] typified interoperability mismatch contests in heterogeneous SOA-centric Systems. An in-depth analysis of diverse mismatch issues was conducted. The presented solution focused on the definition and employment of a machine-readable description of service contracts, Contract Description Language (CDL). Description of the service contract in its entirety was inadequate. Shaikh *et al* [13] analysed the behavioral mismatch backward incompatibilities of Java Software Libraries. A larger-scale cross-version regression testing was presented on sixty-eight consecutive version pairs from fifteen renowned Java software libraries. According to the outcomes, client developers fixed 67% of the client bugs produced by backward incompatibilities in software libraries. All test cases that passed in the original versions were examined. Identified BBIs were biased toward the intended behavioral changes.

2.2. Refactoring in Software Systems

Refactoring is an inevitable phase of the evolution and maintenance of software systems. In addition, it involves the internal structure, maintainability, and design of software without changing the external functionality or behavior of that software [14]. Poorly structured code can lead to increased development cost, decreased scalability, and the inability to add new functionality [15]. Mohammad *et al* [16] presented a model transformation methodology for performing refactoring on SA by utilizing refactoring patterns regarding the necessities of stakeholders. The primary step was to convert SA to a pivot system. Thereafter, the refactoring procedure was performed on the pivot system regarding the refactoring patterns. The findings show that, conducting refactoring on code increased the code's maintainability and understandability. However, the heterogeneity of the metrics was a limitation.

Davide *et al* [17] defined an evolutionary method for multi-objective SA Refactoring (EASIER). A benchmark of diverse configurations was applied to the custom NSGAII for the Emilia specification of a Fire Tracking System (FTS). The results demonstrated the efficiency and applicability of the approach through a case study. Uninterrupted resolution of non-functional models was required to assess SA models. Gurpreet *et al* [18] enhanced software quality through refactoring. An algorithm was applied to improve the refactoring process, leading to high maintainability. From the stated measurements, it was found that the normal chances were 4-9 times higher in the classes impacted by refactorings compared to non-refactored classes. However, the study was restricted to medium-scale projects.

Abdullah *et al* [19] described refactoring approaches to improves SQ from the viewpoints of practitioners. Most of the conventional works choose refactoring methods based on their common usage in academic research. The findings revealed highly utilized refactoring methods, programming languages, and techniques for applying the refactoring methods. However, internal quality attributes such as abstraction, coupling, cohesion, and complexity have not been explored. Chaima *et al* [20] analysed the impact of refactoring on security during quality improvement. To find a balance between quality attributes and security concerning correlation, a multi-objective refactoring recommendation method was applied. Based on the outcomes, developers should make trade-offs between security and other qualities while refactoring software systems because of the negative associations amongst them. The security metrics were restricted to eight measures. Abdullah *et al* [21] developed situations for utilizing refactoring methods for improving software system quality. To assess the impact of refactoring methods on quality attributes, the Quality Model for Object-Oriented Design (QMOOD) was utilized. According to the results, the QMOOD factor played an important role in the diverse effects of refactoring methodologies on quality attributes. Limitations were conflicting effects on quality attributes and insufficient empirical metrics.

Vahid *et al* [22] described a dynamic and interactive search-centric methodology for software refactoring recommendations. In the discussed scenario, NSGA-II was utilized to find a set of ideal refactoring solutions, that optimized the SQ while diminishing the deviation from the primary design. Statistical analysis showed that the incredible, dynamic interactive refactoring method considerably outperformed 4 conventional search-centric refactoring approaches and one fully automated refactoring tool, which was not centred on heuristic search. A highly motivated issue was the Pareto front exploration. Michael and Des [23] analyzed MultiRefactor as an automated refactoring to improve SQ. The individual metrics were examined across 5 code-bases to deduce the highly efficient metrics for general quality enhancement. The multi-objective methodology generated appropriate developments in quality in less time than the ideal solutions of each objective, permitting rapid maintenance cycles. A limited number of detection rules were utilized for isolating certain types of design smells because of the uncertainty involved. Moreover, Table 2 depicts the significance of refactoring in software systems-related research studies.

2.2.1 Refactoring Approaches for Enhancing Software Architecture

Improving SA strategies aim to improve the SA by improving the quality, scalability, maintainability, and performance of software applications while ensuring that the overall behavior of these applications remains unchanged from the client perspective [24]. Modularization, decomposition, code migration, design patterns implementation, service extraction, and rearrangements are some of the more common refactoring strategies [25]. Mohan [26] described AI-augmented SA with an autonomous design pattern and refactoring awareness. An innovative AI-augmented framework was applied for these autonomous refactoring of these systems. The P-Max model might attain a 45% reduction in cyclomatic complexity; however, it had a low explainability score (e.g., 0.4/1.0), with saliency maps that were difficult to interpret. The quality of the existing test suite limits the functional equivalence check. Yang *et al* [27] investigated refactoring-centric program repair applied to programming assignments. The dataset included around1,800 real-time, inaccurate Python program submissions from 361 students. According to the experimental findings, the proposed method is more efficient and effective than the applied feedback generation methods. The limitations were fault localization and the management of algorithmic complexity.

Jonas *et al* [28] presented an approach for quality-driven migration to microservices as a multi-model design science study. The evaluations among software professionals, comprising 9 architects, 4 researchers, (including a professor), and three students, exhibited an overall positive upshot concerning usefulness, efficiency, and usability. These promising results were confirmed by two industrial case studies conducted over 6 months. In the study, a limited sample size of only three participants was obtained. Thakshila *et al* [29] described the reengineering of software systems into microservices as the top-notch and future directions. A taxonomy development methodology was established to systemize knowledge in these areas. In this investigation, manifold reengineering approaches, including static, dynamic, hybrid, and artifact-driven approaches, were revealed. Coupling, cohesion, and modularity were the identified substantial assessment criteria. A limitation of this study is that advanced methods and assessment strategies for microservice discovery were not explored.

Gokul and Ravi [30] explained the automated refactoring of monolithic applications in to a cloud native-container. A fully automated AI-driven methodology was established using Generative AI (GenAI) approaches. According to the findings, the AI-driven system accelerated application modernization, reduced engineering efforts, and enhanced SQ. the limitations of the study are the LLM capabilities and security concerns. Kishore *et al* [31] demonstrated an AI-Augmented model for refactoring enterprise monolithic scenarios. An AI-augmented modular refactoring framework was introduced in the proposed approach. According to the evaluation results, the approach delivered measurable improvements across multiple dimensions of legacy modernization. The methodology was designed only to support decision-making (DM). Roberta and Fabio [32] examined a quality-driven refactoring method in BIM Italia. The applied method facilitated the satisfaction of the target performance quality restraints. According to empirical assessment, the refactoring method optimized the response time of the refactored microservices' up to 5 times concerning1st version of the microservices. The imitations were the automated detection constraints and localized microservice trade-offs. Djezar *et al* [33] represented the quality-driven transformation of legacy systems under uncertainty using the fuzzy logic-based concern-oriented modelling approach. In the applied methodology, the KDM-based model transformation was integrated with an aspect-oriented design. The results showed improved management of cross-cutting concerns and performance, reduced alert latency, and better scalability. Furthermore, Table 3 depicts, - the significance of the refactoring approaches for enhancing SA-related research articles.

2.3. Mismatch-Based Refactoring Recommendations

An intelligent approach for maintaining software by identifying the differences between a system's current architecture, design rules, code structure, quality attributes, and intended functionality is referred to as Mismatch-Based Refactoring Recommendations. Recommendations were created utilizing four methodologies: the rule-based method, machine-learning model, historical change analysis, and context-aware decision systems [34]. Ruru *et al* [35] explained automatic clone mismatch-based refactoring recommendations regarding past and present. By extracting features from the current status, CREC was developed to recommend clones. CREC considerably outperformed a top-notch similar approach on the dataset. Additionally, it achieved better F-scores. The high cost of manual reviews and the frequent divergence between syntactical similarities are the limitations. Alessandro *et al* [36] analysed an intelligent DM support system for mismatch-based refactoring recommendations in a cloud framework. For processing historical data, an ML procedure centred on neural networks for data regression was utilized. The study conclusions exhibited how the ML method could implement customer features, supplier features, and solution features to efficiently predict customer behavior. The limitations were information overload from massive service repositories and severe data sparsity for new resources.

Lei *et al* [37] presented an intelligent matching recommendation model for a manufacturing potential - sharing platform with fairness concerns. A two-dimensional crossover and an order-1st mutation were established and implemented with GAs, comprising the GA and NSGA-II. As per the findings indicate that the, algorithms concerning conventional GA and NSGA-II were efficient for diverse schemes. The limitations include multi-stakeholder fairness, dynamic data handling, and computational scalability. Diego *et al* [38] described the effect of refactoring on smells as a Longitudinal Study of twenty-three Software Projects. The research analysed how commonly utilized refactoring types affected the density of thirteen types of code smells along the version histories of twenty-three projects. The outcomes suggested that developers need more guidance for completely remove a code smell after restructuring a smelly element. Positive effects were observed in recurring refactoring-smell pairs. Linna *et al* [39] examined BRAFAR a bidirectional refactoring, fault localization, alignment, and repair for programming assignments and mismatch-based refactoring recommendations. The Brafar approach was applied to explore feedback generation for IPAs. Brafar performed considerably better than Refactory and Clara, thus producing accurate repairs for more inaccurate programs with small patch sizes. Its performance relies heavily on existing Automated Fault Localization (AFL) techniques [3] and is limited in generating complex fixes.

Danilo *et al* [40] defined RefDiff by detecting refactorings in Version Histories for mismatch-based refactoring recommendations. Regarding static analysis and code similarity, RefDiff implemented an amalgamation of heuristics for detecting thirteen renowned refactoring types. According to the evaluation, RefDiff has better precision and recall than conventional top-notch techniques. A limitation was over-reliance on AST token matching, which caused false negatives.

2.4 Recent advancements in architecture recovery, mismatch- based refactoring

Liu *et al* [41] addressed the architectural inconsistency problem between expert-developed intended architecture and implemented architecture through reflection models and, static analysis and attempted to minimize inconsistencies at the module and entity levels. The approach is not explainable, although it is effective, and there is no prioritization of architectural inconsistencies. Ding *et.al* [42] conducted an empirical study by evaluating the trends on Stack Overflow and analyzing the posts from practitioners and conduct discussion threads. The study collected the potential risks faced by practitioners and identified 12 types of architecture refactoring upholding the demand for further research on conducting architecture refactoring effectively. Pan *et al* [43] addressed the challenges of recovering accurate architecture from large legacy software code repositories. Classic static analysis techniques, and explicit human intervention are used to generate reference architectures but graph code reasoning and LLM-powered semantic synthesis lack accurate and explainable recovery architectures. Nursapa *et al* [44] fine-tuned CODE-BERT and Code T5 strengths using transformer-based classification. The refactoring problem was formulated as a multi-class classification problem. Frames the refactoring tasks maps to three classification refactoring labels. More code fragments are mapped to the buggy classes. A systematic detection and prioritization architectural mismatches or inconsistencies is not addressed.

Ahadeaf *et.al* [45] merged method and class-level code smells into a binary relevant random forest and then performed a tree based multi-label classification which limits scalability. The transfer learning approach is limited to the java language and data sets. Alharbi *et.al* [46] addressed the lack of a comprehensive and benchmark study of automated refactoring tool selection. The study identified search-based, rule-based, clustering-based and text-based refactoring tools. The study demands merging the advancements of AI with semantic-based machine learning and AI driven refactoring recommendations. W Zhao *et.al* [47] addressed software architecture recovery, which is a crucial

step in identifying architectural mismatches between the implemented and intended architecture. The approach used large language models to capture semantic information at both the implementation and architectural level fused with improved clustering algorithm. This approach has good recovery measures. Zhang *et.al* [48] proposed a recommendation method for move method refactoring by fusing semantic, structural and code metric features and then feeding them into a BiLSTM branch. The approach was narrowed to address only a single code smell. However, they conclude that that deep learning approaches successfully recommend refactoring operations.

Ahmed *et.al* [49] stated that architecture degradation affects system quality, maintainability and adaptability, however, the study disconnects between detection and remediation. Proactive approaches are rare in the literature. Future research should focus on building a framework for systematically analysing and remediating architectural degradations. Rasshid *et.al* [50] integrated CODEBERT embeddings with BiLSTM for code smell prioritization and the explainability of various refactorings using LIME. However, approached a lower level of code discrepancies and failed to address architectural inconsistencies.

3. REVIEW SUMMARY

Answer to our research investigations is exemplified in the following tables. Table 1 shows the significance of SA mismatch-related research studies. Every category concentrated on diverse kinds of mismatches (e.g., CM, behavioral mismatch, variability mismatch, and interoperability mismatch). Each of these mismatches utilized mathematical algorithms for the assessment of the mismatch recognition and resolution process, comprising cluster analysis, machine learning/AI, and slicing algorithms. According to study findings, both CM and behavioral mismatch were the two mismatches that had the most research done, while the other types of mismatches did not have nearly as much research. Evidence for several limitations in the research, including restricted project scopes, absence of machine-readable architectures, and delays in communications, was also offered by the studies.

Table 1: Significance of the software architecture mismatch-related research studies

Study type	Tools used	Algorithm	Time	Patterns	Refactors	Response rate	Cost	Challenges	Ref
Research study	Model transformation approach	Decision-making algorithm	Nil	100%	Nil	100%	Nil	The presented refactoring framework could only use quantitative metrics.	[16]
Experimental evaluation	EASIER	Evolutionary algorithm	27.53 s	Nil	One refactoring action in the sequence: 92%	69%	Nil	Only additive actions were considered in the Æmilia Refactoring Actions Library.	[17]
Experimental evaluation	Refactoring techniques	Refraction-based algorithm	Extended execution time	0.1	553	Nil	High maintenance cost	Limited to medium-scale approaches.	[18]
Observational evaluation	Refactoring techniques	Substitute algorithm	60%	Nil	68	50%	Reduced maintenance cost	Attained a low agreement rate.	[19]
Observational evaluation	QMOOD model	Multi-objective search algorithm	30 times	Nil	14	Nil	70% of lifetime budgets	The transformations might increase the security metrics.	[20]

Observational evaluation	QMOOD model	Multi-objective search algorithm	657 times	Nil	68	Nil	80% total software cost	Presence of limited scalability.	[21]
Experimental evaluation	Dynamic Search-Based Approach	NSGA-II algorithm	Time reduced by 51%	Nil	87% of the refactoring operations were considered as feasible	93%	Nil	Refactoring tools did not update their recommended refactoring solutions.	[22]
Experimental evaluation	Multirefactor	Search-based optimization algorithm	3 hours 46 minutes 17 seconds	23	26	Nil	Nil	Limited selection of refactorings.	[23]

As shown in Table 2, the EASIER tool, which used an evolutionary algorithm, summarized its execution in 27.53s with 92% refactoring efficiency and 69% response rate. The long execution time and high maintenance cost stuck the 553 refactors from refraction-based algorithms. The substitute algorithms provided 60%-time efficiency, with 68 refactors produced, and a 50% response rate. QMOOD model studies, performed using multi-objective search algorithms, required 30 and 657 executions, while consuming 70% and 80% of the software cost budget, respectively. While the Multirefactor Algorithm produced 71.3% savings in execution time with 23 patterns and 26 refactorings, the NSGA-II algorithm produced a 51% reduction in execution time and a 93% response rate.

Table 2: Importance of refactoring in software systems-related research studies

Study type	**Tools used**	**Algorithm**	**Time**	**Patterns**	**Refactors**	**Response rate**	**Cost**	**Challenges**	**Ref**
Research study	Model transformation approach	Decision-making algorithm	Nil	100%	Nil	100%	Nil	The presented refactoring framework could only use quantitative metrics.	[16]
Experimental evaluation	EASIER	Evolutionary algorithm	27.53 s	Nil	One refactoring action in the sequence: 92%	69%	Nil	Only additive actions were considered in the Æmilia Refactoring Actions Library.	[17]
Experimental evaluation	Refactoring techniques	Refraction-based algorithm	Extended execution time	0.1	553	Nil	High maintenance cost	Limited to medium-scale approaches.	[18]
Observational evaluation	Refactoring techniques	Substitute algorithm	60%	Nil	68	50%	Reduced maintenance cost	Attained a low agreement rate.	[19]
Observational evaluation	QMOOD model	Multi-objective search algorithm	30 times	Nil	14	Nil	70% of lifetime budgets	The transformations might increase the security metrics.	[20]

Observational evaluation	QMOOD model	Multi-objective search algorithm	657 times	Nil	68	Nil	80% total software cost	Presence of limited scalability.	[21]
Experimental evaluation	Dynamic Search-Based Approach	NSGA-II algorithm	Time reduced by 51%	Nil	87% of the refactoring operations were considered as feasible	93%	Nil	Refactoring tools did not update their recommended refactoring solutions.	[22]
Experimental evaluation	Multirefactor	Search-based optimization algorithm	3 hours 46 minutes 17 seconds	23	26	Nil	Nil	Limited selection of refactorings.	[23]

Table 3: Significance of the refactoring approaches for enhancing software architecture-related research articles

Refactoring approaches	**Data**	**Complexity reduction**	**Time**	**Size**	**Accuracy**	**Rate**	**Challenges**	**Ref**
Pattern-based refactoring	20 simulated clients	25%	Nil	Small patch size	Higher accuracy	99.8% test pass	The refactoring task was complex.	[26]
Pattern-based refactoring	1,800 records	Nil	5.5 sec	40%	40%	67.08% sampling rate	Limited feedback generation techniques.	[27]
Quality-based refactoring	110 scientific records	Nil	Nil	Nil	Improved accuracy	Usability rate: 80%	Presence of resource limitations.	[28]
Microservices-based quality	117 primary records	Nil	65% low response time	Nil	Nil	Response rate 44%	Lack of standardized mechanisms.	[29]
Quality-based refactoring	Not reported	70%	70%	Nil	97%	Potentiality: 70%	Presence of scalability issues.	[30]
Component-based refactoring	Not reported	38%	Less runtime	420	Nil	Effort reduction: 28.3%	The approach relied primarily on static analysis and historical structural signals.	[31]
Pattern-based refactoring	10,000 requests	43%	177 seconds	Nil	Higher accuracy	Response time	Limited time allocation for	[32]

						improvement: 47%	refactorization.	
Quality-based refactoring	5178 records	40%	18 ms	Low cohesion size	75%	Performance rate: 68%	Limited database overhead.	[33]

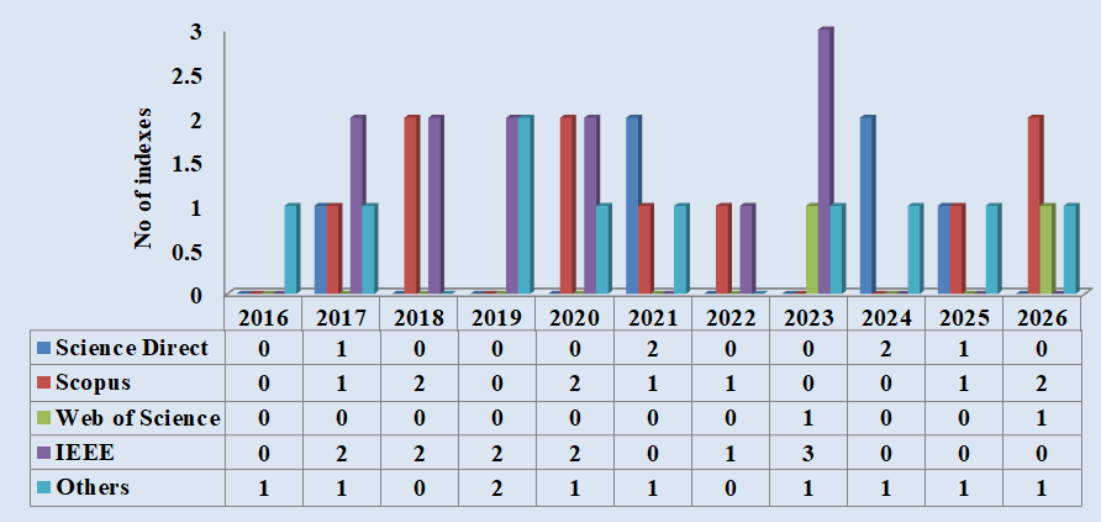
Table 4: Significance of the mismatch-based refactoring recommendations related to research articles

Type	**Approach**	**Dataset**	**Samples**	**Time**	**RMSE**	**Rate**	**Challenges**	**Ref**
AI-Based	Learning-based approach	Nil	34 features	50 times	Nil	Likelihood rate: 50%	Every clone couldn't be refactored.	[35]
AI-Based	Cloud framework	Neural network-based datasets	60 customers	29.49 h	0.4	SUR: 77.45%	Limitations in the pure collaborative filtering approach.	[36]
Context-based	NSGA-II	Mendely dataset	15 sellers and 20 buyers	30 times	Nil	Nil	Research did not take characteristics of manufacturing capacity sharing into consideration.	[37]
Prioritization	Code smells for refactoring	Private dataset	23 software projects	9.7% improvement	Nil	Refracting rate: 33.3%	Analysis of all possible values was not viable.	[38]
Selection-based	Refactory	BRAFAR	1783 submissions	3.0 s	0.49	Repair rate: 93.9%	The model was limited to fixing multi-location errors.	[39]
Rule-based	ReDiff	Public dataset	2441 refactorings	894 ms	Nil	Precision: 85.7%	The evaluation oracle resulted in human errors.	[40]

Table 3 shows that pattern-based refactoring techniques reduce response time, complexity, but improved testing accuracy. The quality attributes such as usability, scalability and accuracy of the systems were improved. The response time improvements were from the microservice- based architectures but there are barriers due to the lack of resources and databases. However, many of these approaches are purely based on static analysis and historical data which limits the adoption capability in dynamic environments.

AI-based, context-based, prioritization-based, selection-based, and rule-based recommendations were various types of recommendations and refactoring approaches in the field of SE. The AI-based recommendation methods achieved a moderate degree of success, but had issues with refactoring any/all code clones. Context-based recommendations made use of the NSGA-II optimization method, thereby enhancing the DM process. Nevertheless, the context-based recommendation method was less effective in identifying manufacturing capacity (sharing) characteristics. A better success rate (repair) was achieved using the selection-based method (refactoring). To identify third-party changes related to refactoring, rule-based recommendations, such as RefDiff, achieved high accuracy, with an overall accuracy of 77% in the human error factor for the evaluation process. The significance of the mismatch-based refactoring recommendations related to research articles is explored in Table 4.

The following Figure 2 summarizes the research indexes in each of the research repositories.



	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026
Science Direct	0	1	0	0	0	2	0	0	2	1	0
Scopus	0	1	2	0	2	1	1	0	0	1	2
Web of Science	0	0	0	0	0	0	0	1	0	0	1
IEEE	0	2	2	2	2	0	1	3	0	0	0
Others	1	1	0	2	1	1	0	1	1	1	1

Figure 2: Research indexes VS Research Repositories

The selected 50 studies were rigorously examined to assess and finalize the necessity of mismatch-based refactoring recommendations intelligently and explainable. Most of the relevant literatures focused on a minor code clone detection, code-smell prioritization and architecture smell - based buggy or non - buggy classification - based refactorings. Studies have mainly concentrated on either improve the quality or reduce the inherent complexity of software systems. However, there were challenges with the scalability of recommendation systems, developer context issues and real-time monitoring and evaluation issues. Thus, an automated framework for the automated recovery of the reference architecture, summarizing the mismatches between the implemented and reference architecture intelligently and explainable with deep learning strategies are vital to the current research.

4. CONCLUSION

This study focused on the relevant research on architecture-mismatch-based refactoring recommendations between 2016 and 2026. The adopted refactoring approaches are rule-based, pattern-based, component-driven, quality oriented, machine-learning based and context-aware. According to our structured study, the refactoring literature has proliferated since the last decade, and studies have shown an important fact that AI-assisted refactoring is more promising for achieving better results. However few researches are available in the automated recovery of the reference architecture, prioritizing mismatches intelligently and producing explainable, context-aware, reliable and interpretable refactoring recommendations. The study demands the development of a hybrid AI framework for explainable and real-time adaptable refactoring recommendations with validation in industry settings.

Declaration of competing interestS

No conflict of interest either in the case of financial or other sources.

Acknowledgment

We thank God almighty to support and complete this review process.

Data availability

From IEEE, SCOPUS, science direct and other well-known research repositories and cited properly in references.

AI USE and Declaration of generative AI use

Some tools like paperpal and chatgpt is used in the review process and some internet sources.

References

1. Sardar Mudassar Ali Khan, "Exploring the Impact of Software Architecture on System Performance and Maintainability: A Comprehensive Study", Research Gate, pp. 1-2, 2023. https://www.researchgate.net/publication/372231890_Exploring_the_Impact_of_Software_Architecture_on_System_Performance_and_Maintainability_A_Comprehensive_Study
2. Federico Ciccozzi, Juraj Feljan, Jan Carlson and Ivica Crnkovic, "Architecture optimization: speed or accuracy? Both," Software Qual Journal, vol. 26, pp. 1-24, 2018.
3. Sharon Andrews and Mark Sheppard, "Software Architecture Erosion: Impacts, Causes, and Management", International Journal of Computer Science and Security, vol. 14, no. 2, pp. 1-12, 2020.
4. Abdullah Almogahed, Hairulnizam Mahdin, Said Badreddine, Mazni Omar, Abdulwadood Alawadhi, Samera Obaid Barraood, Manal Othman, Shazlyn Milleana Shaharudin and SitiSalwani Yaacob, "Towards an effective refactoring taxonomy for sustainable software systems", Plos One, vol. 21, pp. 1-29, 2026.
5. Damian A. Tamburri and Rick Kazman, "General Methods for Software Architecture Recovery: A Potential Approach and its Evaluation", Empirical Software Engineering, vol. 23, no. 3, pp. 1-37, 2016.
6. Claudia Ayala, Anh Nguyen-Duc, Xavier Franch, Martin Höst, Reidar Conradi, Daniela Cruzes and Muhammad Ali Babar, "System requirements-OSS components: matching and mismatch resolution practices – an empirical study", Empirical Software Eng, pp. 1-56, 2018. https://doi.org/10.1007/s10664-017-9594-1
7. Grace A. Lewis, Stephany Bellomo and April Galyardt, "Component Mismatches Are a Critical Bottleneck to Fielding AI-Enabled Systems in the Public Sector", Arxiv, pp. 1-4, 2019. https://arxiv.org/pdf/1910.06136
8. Shawn Rasheed, Max McPhee, Lisa Patterson, Stephen MacDonell, and Jens Dietrich, "Hidden Dependencies and Component Variants in SBOM-Based Software Composition Analysis", Arxiv, pp. 1-20, 2026. https://arxiv.org/pdf/2604.21278
9. Grace A. Lewis, Stephany Bellomo and Ipek Ozkaya, "Characterizing and Detecting Mismatch in Machine-Learning-Enabled Systems", 201 IEEE/ACM Ist workshop on AI Engineering – Software Engineering for AI(WAIN) 30-31 May, 2021, pp: 133-140 https://doi.org/10.1109/WAIN52551.2021.0002.
10. Ming Chang, Hideyuki Ando, Taro Maeda and Yasushi Naruse, "Behavioral effect of mismatch negativity neurofeedback on foreign language learning", Plos One, vol. 16, no. 7, pp. 1-18, 2021.
11. Marc Hentze, Chico Sundermann, Thomas Thüm and Ina Schaefer, "Quantifying the Variability Mismatch Between Problem and Solution Space", In ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022. https://doi.org/10.1145/3550355.3552411
12. Cristina Paniagua, Jens Eliasson and Jerker Delsing, "Interoperability Mismatch Challenges in Heterogeneous SOA-based Systems", In 2019 IEEE International Conference on Industrial Technology (ICIT), pp. 788-793, 2019.
13. Shaikh Mostafa, Rodney Rodriguez and Xiaoyin Wang, "Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries", In Proceedings of 26th International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 2017. https://doi.org/10.1145/3092703.3092721
14. Abdullah Almogahed, Hairulnizam Mahdin, Mazni Omar, NurHaryani Zakaria, Salama A. Mostafa, Salman A. Alqahtani, Pranavkumar Pathak, Shazlyn Milleana Shaharudin and Rahmat Hidaya, "A Refactoring Classification Framework for Efficient Software Maintenance", IEEE Access, vol. 11, pp. 1-14, 2023.
15. Srinikhita Kothapalli, Aditya Manikyala, HariPriya Kommineni, et al., "Code Refactoring Strategies for DevOps: Improving Software Maintainability and Scalability", ABC Research Alert, vol. 7, no. 3, pp. 1-12, 2019.
16. Mohammad Tanhaei, "A model transformation approach to perform refactoring on software architecture using refactoring patterns based on stakeholder requirements", AUT Journal of Mathematics and Computing, vol. 1, no. 2, pp. 1-38, 2020.
17. DavideArcelli, Vittorio Cortellessa, MattiaD'Emidio and Daniele Di Pompeo, "EASIER: an Evolutionary Approach for multi-objective Software architecture Refactoring", In 2018 IEEE International Conference on Software Architecture (ICSA), pp. 105-10509, 2018. https://doi.org/10.1109/ICSA.2018.00020
18. Gurpreet kaur and Balraj Singh, "Improving the Quality of Software by Refactoring", International Conference on Intelligent Computing and Control Systems, pp. 1-7, 2017. https://ieeexplore.ieee.org/abstract/document/8250707/

19. Abdullah Almogahed and Mazni Omar, “Refactoring Techniques for Improving Software Quality: Practitioners’ Perspectives”, Journal of Information and Communication Technology, vol. 20, no. 4, pp. 1-29, 2021.
20. Chaima Abid, Marouane Kessentini, Vahid Alizadeh, Mouna Dhaouadi and Rick Kazman, “How Does Refactoring Impact Security When Improving Quality? A Security Aware Refactoring”, IEEE Transactions on Software Engineering, vol. 48, no. 3, pp. 1-16, 2020.
21. Abdullah Almogahed, Mazni Omar, NurHaryani Zakaria, Ghulam Muhammad, And Salman A. Alqahtan, “Revisiting Scenarios of Using Refactoring Techniques to Improve Software Systems Quality”, IEEE Access, vol. 11, pp. 1-20, 2023.
22. Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ocinneide and Ali Ouni, “An Interactive and Dynamic Search-Based Approach to Software Refactoring Recommendations”, IEEE Transactions on Software Engineering, vol. 46, no. 9, pp. 1-32, 2020.
23. Michael Mohan and Des Greer, “MultiRefactor: Automated Refactoring to Improve Software Quality”, In International Conference on Product-Focused Software Process Improvement, pp. 556-572, 2017.
24. Vittorio Cortellessa, Romina Eramo and Michele Tucci, “From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement”, Information and Software Technology, vol. 127, pp. 1-28, 2020.
25. Nivedhaa N, “Software Architecture Evolution: Patterns, Trends, And Best Practices”, International Journal of Computer Sciences and Engineering (IJCSE), vol. 1, no. 2, pp. 1-14, 2024.
26. Mohan Siva Krishna Konakanchi, “AI-Augmented Software Architecture: Autonomous Refactoring with Design Pattern Awareness”, International Journal of Emerging Trends in Computer Science and Information Technology, vol. 6, no. 3, pp. 1-7, 2025.
27. Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong and Abhik Roy choudhury, “Re-factoring based Program Repair applied to Programming Assignments”, In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 388-398, 2019. https://discovery.ucl.ac.uk/id/eprint/10091878/1/ase19.pdf
28. Jonas Fritzsch, Justus Bogner, Tobias Haller, Daniel Koch, Alfred Zimmermann and Stefan Wagner, “Developing a Framework for the Quality-Driven Migration to Microservices: A Multi-Method Design Science Study”, Software: practice and experience, pp. 1-36, 2026. https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.70062
29. ThakshilaImiya Mohottige, Artem Polyvyanyy, Colin Fidge, Rajkumar Buyya and Alistair Barros, “Reengineering software systems into microservices: State-of-the-art and future directions”, Information and Software Technology, vol. 183, pp. 1-27, 2025.
30. Gokul Chandra Purnachandra Reddy and Ravi Sastry Kadali, “Automated Refactoring of Monolithic Applications to CloudNative Containers: Application Modernization using GenAI and Agentic Frameworks”, International Journal of Intelligent Systems and Applications in Engineering, vol. 12, pp. 1-14, 2024.
31. Kishore Subramanya Hebbar, “An AI-Augmented Framework for Refactoring Enterprise Monolithic Systems”, International Journal of Intelligent Systems and Applications in Engineering, vol. 11, pp. 1-12, 2023.
32. Roberta Capuano and Fabio Vaccaro, “The Quality-Driven Refactoring Approach in BIM Italia”, In 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C), pp. 22-31, 2023. https://ieeexplore.ieee.org/abstract/document/10092590/
33. DjezarSarra, Aouag Sofiane and Kadri Salim, “Towards a Quality-Driven Transformation of Legacy Systems under Uncertainty: A Fuzzy Logic-Based Concern-Oriented Modeling Approach”, SSRN, pp. 1-35, 2026. https://dx.doi.org/10.2139/ssrn.6515366
34. Mykola A. Hodovy chenko and Dmytro D. Kurinko, “Analysis of existing approaches to automated refactoring of object-oriented software systems”, Herald of Advanced Information Technology, vol. 8, no. 2, pp. 1-18, 2025. https://doi.org/10.15276/hait.08.2025.11
35. RuruYue, ZheGao, Na Meng, Yingfei Xiong, Xiaoyin Wang and J. David Morgenthaler, “Automatic Clone Recommendation for Refactoring Based on the Present and the Past”, In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 115-126, 2018. https://doi.org/10.1109/ICSME.2018.00021
36. Alessandro Simeone, Yunfeng Zeng and Alessandra Caggiano, “Intelligent decision-making support system for manufacturing solution recommendation in a cloud framework”, The International Journal of Advanced Manufacturing Technology, vol. 112, pp. 1-16, 2021. https://doi.org/10.1007/s00170-020-06389-1
37. Lei Xie, Jianghua Zhang, Qingchun Meng, Yan Jin and Weibo Liu, “An Intelligent Matching Recommendation Algorithm for A Manufacturing Capacity Sharing Platform with Fairness Concerns”, International Journal of Production Research, pp. 1-35, 2022. https://doi.org/10.1080/00207543.2022.2155999
38. Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro and Alexander Chávez, “Understanding the Impact of Refactoring on Smells:A Longitudinal Study of 23 Software Projects”, Proceedings of the 2017 11th Joint Meeting on foundations of Software Engineering, pp. 465-475, 2017. https://dl.acm.org/doi/pdf/10.1145/3106237.3106259
39. Linna Xie, Chongmin Li, Yu Pei, Tian Zhang and Minxue Pan, “BRAFAR: Bidirectional Refactoring, Alignment, Fault Localization, and Repair for Programming Assignments”, Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 856-868, 2024. https://dl.acm.org/doi/pdf/10.1145/3650212.3680326.
40. Danilo Silva and Marco Tulio Valente, “RefDiff: Detecting Refactorings in Version Histories”, In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 269-279, 2017. https://doi.org/10.1109/MSR.2017.14.

41. J Liu, W Jin, J Zhou, O Feng, Ming Fan, H Wang, “3Erefactor: Effective, Efficent and Executable Refactoring Recommendation for Software Architectural Consistency”, IEEE Transactions on Software Engineering, Vol. 50, Issue 10, October 2024.
42. W Ding, Ran Mo, Chaochao Wu, Haeopeng Song, Hang Fu, Xinya Mu, “Exploring and Analyzing Architecture Refactoring in Practice”, IEEE Transactions on Software Engineering”, Jan 2026, pp. 286-303, vol. 52.
43. Rusheng Pan, Bingcheng Mao, Tianyi Ma, Zhenhua Ling, “Archagent:Scalable Legacy Software Architecture Recovery with LLMs”, https://doi.org/10.48550/arXiv:2601.13007.
44. Samal Nusarpa, Anastassiya Samuilova, Alessio Bucaioni, Phuong T. Nguyen, “ROSE: Transformer - based Refactoring Recommendation for Architectural Smells”, 2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2-3 Oct 2025.
45. Haneen M Ahadeaf, Mubarak Alrashoud, “A Unified Multi-Label Code Smell Dataset for Code Smell detection at different Granularities”, IEEE Access Vol. 14, pp. 714-737, Dec. 2025.
46. Maha Alharbi, Mohammed Alshayeb, “A Comparative study of Automated Refactoring Tools”, IEEE Access, Vol. 12, pp. 18764-18781, Feb 2024.
47. Wenting Zhao, Wuxia Jin, Yiran Zhang, Ming Fan, Haijun Wang, Li Li, Yang Liu, Ting Liu, “Software Architecture Recovery Augmented with Semantics”, IEEE Transactions on Software Engineering, Vol. 52, pp: 338-359, Issue 1 Jan 2026.
48. Y Zhang, Z Gu, N Zhang, K Zheng, “Recommending move method Refactoring Opportunities based on Feature Fusion and Deep learning”, IEEE Latin American Transactions, pp. 135-143, Vol. 24, Issue 2, Feb 2026.
49. N Ahmed, R Su, M Esposito, Andrea Janes, V Lenarduzzi, D Taibi, “Architectural Degradation: Definition, Motivations, Measurement and Remediation Approaches, arXiv: 2507.14547, July 2025.
50. M M Rashid, M H Osman, K Y Sharifand H ZulZalil, “Interpretable deep learning for efficient Code Smell Prioritization in Software Systems”, IEEE Access, March 2025.