

Longitudinal Evidence of Architectural Sustainability in Adaptive Software Remodularization: A 280-Release Empirical Study

Randeep Singh¹, Ganesh Khekare²

¹ Department of Computer Science Engineering, Lincoln University College, Selangor Darul Ehsan, Petaling Jaya, Malaysia;

² School of Computer Science and Engineering, Vellore Institute of Technology, Vellore, India

Email ID pdf.Randeep@lincoln.edu.my, khekare.123@gmail.com

Abstract

Evaluating software remodularization at a single release snapshot is widely accepted as insufficient. However, still it remains the dominant practice in literature. This paper reports the longitudinal experimental results of AMORE, an Adaptive Multi-Objective REmodularization framework, assessed across 280 consecutive releases drawn from 15 mature open-source Java systems spanning 5 to 10 years of evolution. We measure four purpose-built longitudinal metrics namely Modular Stability Index (MSI), Change Localization Ratio (CLR), Predictive Maintainability Index (PMI) trend, and Architectural Drift Index (ADI). AMORE achieves an average MSI of 0.81 — 16–31% higher than competing baselines — while maintaining a CLR of 0.74, an ADI of 0.19 (up to 39% lower than structural-only methods), and a flat-to-improving PMI trajectory. All gains are statistically significant ($p < 0.05$) with medium-to-large Cliff's delta effect sizes. Taken together, these findings demonstrate that adaptive, multi-dimensional remodularization produces genuinely durable architectural improvements, not transient snapshot-level boosts.

Keywords— software remodularization; longitudinal evaluation; architectural stability; AMORE; NSGA-III; maintainability

1. Introduction

Software systems are considered as living artefacts as they constantly undergoes evolution overtime. They accumulate new features, absorb bug fixes, and adapt to shifting runtime environments. This constant evolution of software puts quiet pressure on the underlying modular structure. If that pressure isn't managed carefully, coupling creeps up, cohesion breaks down, and what once began as a clean architectural design eventually becomes a liability. Software remodularization is one of the strategies that can automate or semi-automate recovery of improved module decompositions. It is the software engineering field's primary answer to this architectural decay.

Although methodological advances have been made in literature that range from simple graph-clustering tools like Bunch [1] to many-objective evolutionary frameworks like NSGA-III [2]. However, in all these approaches one evaluation habit has stubbornly persisted. Most of the existing remodularization studies in literature assess their technique on a single release snapshot. The problem with this approach is obvious in retrospect. A decomposition that looks excellent at version t may dissolve quickly as the system evolves. It is also important to keep in mind that short-term improvements can be only temporary [3, 4].

In this paper, we aims to directly address this identified gap. We reported the full experimental results of AMORE (Adaptive Multi-Objective REmodularization [5]) in this paper. We evaluated AMORE longitudinally across 280 releases from 15 mature open-source systems. The evaluation is structured around four special newly proposed metrics that cannot be computed from a single snapshot. These

metrics includes Modular Stability Index (MSI), Change Localization Ratio (CLR), long-term Predictive Maintainability Index (PMI) trend, and Architectural Drift Index (ADI). The authors in this paper are of the opinion that these four metrics together assess whether architectural improvements endure, generalize across systems, and/or withstand real-world software evolution pressures.

2. Related Work

2.1 Architectural Erosion and its Consequences

During the maintenance of a system, software architecture degrades continuously. According to Mens et al. [6], architectural quality is a temporal property rather than a permanent attribute. This study clearly implies the importance of evolution-aware analysis. Garcia et al. [7] demonstrate that architectural bad smells accumulate systematically as systems evolve. This negatively affects maintainability and comprehension at the structural level. More recently, Arcelli Fontana et al. [8] conducted a large-scale multi-release analysis. They found that architectural smells tend to persist over time and they are closely associated with increased developer maintenance burden. These findings motivate remodularization necessary and as an intervention. Simultaneously these studies highlights the necessity of conducting longitudinal evaluations, as erosion is a phenomenon that unfolds over time.

2.2 The Limits of Snapshot-Based Evaluation

Most of the existing remodularization studies assess modular quality at a single point in time using static cohesion and coupling metrics. Mumtaz et al. [9] surveyed the literature related to architectural smell detection. They concluded that detector validation is most commonly performed on isolated snapshots in literature. The literature significantly lacks empirical evidence that link smell removal to improved future maintainability. Sas et al. [10] analysed architectural smell evolution in an industrial system across releases and showed that short-term refactoring improvements are frequently transient. This highlights importance of longitudinal studies as they capture aspects which something a single-snapshot study would entirely miss. Similarly, Gnoyke et al. [11] characterized smell evolution over hundreds of releases for a large system. They identified patterns like long-lived smells and cyclical re-emergence, which single-release studies cannot detect properly. Candela et al. [12] further showed that structural metrics such as modularization quality (MQ) vary considerably across releases and development activities. This finding cautions against over-reliance on single-version assessments as evidence of robust architectural quality.

2.3 Remodularization Techniques

The field of software remodularization research has evolved from focusing solely on structural clustering to adopting multi-dimensional, evolution-aware methodologies. Early graph-clustering methods like Bunch [1] and ACDC treat remodularization purely as a cohesion & coupling optimization problem. Subsequent IR-based approaches incorporated lexical similarity in remodularization process which is obtained from identifiers and comments [13]. Evolution-aware methods added co-change and logical coupling information from version control histories [14]. They demonstrated that historical data improves maintenance task localization. Search-based approaches modelled remodularization as a multi-objective problem using NSGA-II and NSGA-III variants [2, 15]. Such approaches simultaneously optimizing cohesion, coupling, size balance, and semantic coherence aspects of a software system. AMORE [5] extends search-based remodularization by integrating structural, semantic, and runtime behavioral evidence. It applies an adaptively weighted similarity model along with using NSGA-III optimization with a similarity-aware warm-start strategy.

2.4 Longitudinal Studies in Software Engineering

Palomba et al. [16] conducted one of the first large-scale longitudinal investigations of design smells across releases. They showed that smells often persist for long periods and that snapshot correlations can

be misleading without temporal context. Carruthers et al. [3] empirically assessed the temporal validity of commonly used software quality indicators across 72 snapshots. They found that many metrics change over time and that their predictive validity is limited unless temporal stability is considered. Jolak et al. [17] investigated the impact of architectural smells on maintainability over time. Their study reinforced the need for evolution-aware evaluation strategies.

In this paper, we aim to close the gap by undertaking the first longitudinal evaluation of an adaptive multi-dimensional remodularization framework spanning 280 releases. We also aim to propose metrics specifically crafted to assess sustainability that cannot be measured by snapshot-based research.

3. Proposed Methodology

Figure 1 presents the six-phase longitudinal evaluation pipeline used in this study. Each phase is executed independently per release under a strict no-look-forward protocol, ensuring that remodularization and metric computation for release t rely only on information available at that point in time.

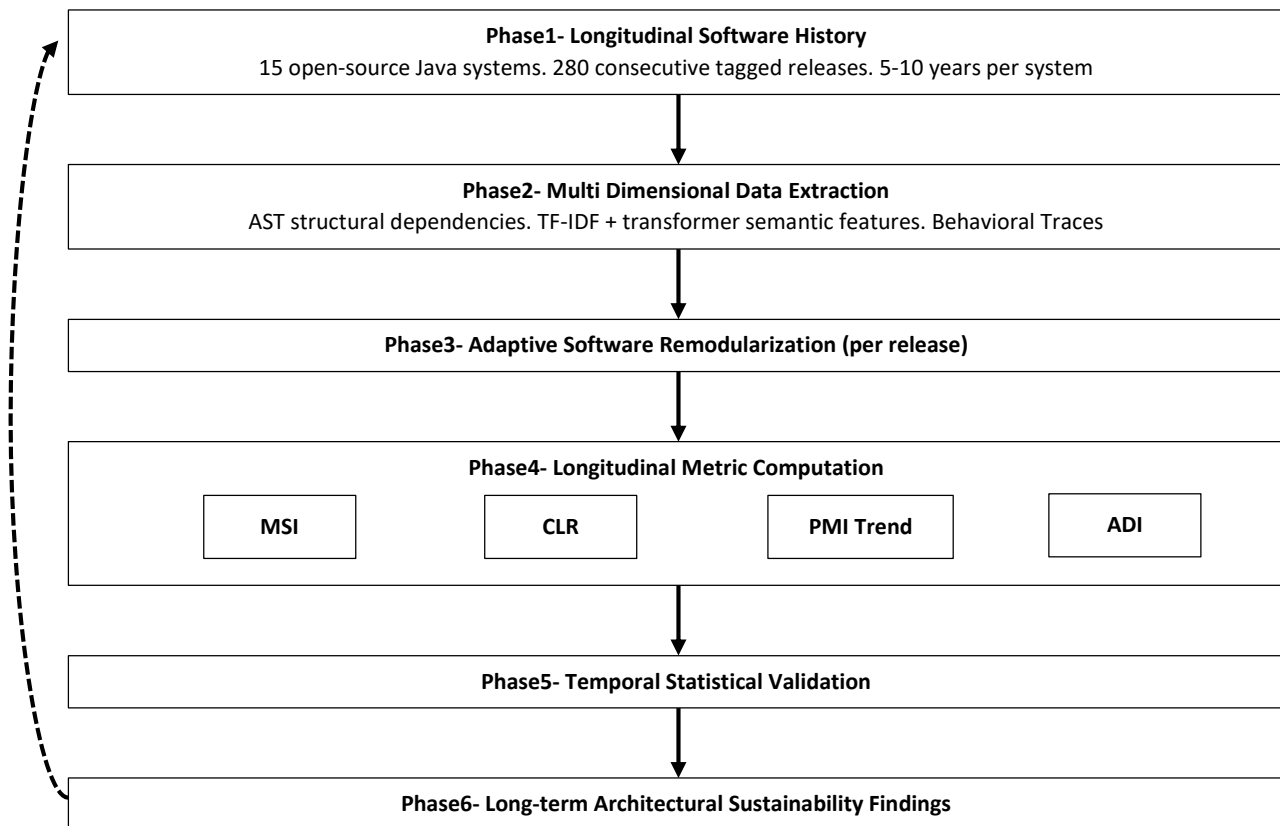


Figure-1: Proposed Longitudinal Evaluation Pipeline.

3.1 Subject Systems and Baselines

The study covers 15 mature open-source Java systems from diverse domains (build tooling, messaging, server runtimes, utility libraries, databases). The size of these systems ranges from 520 to 1,950 classes. Further, for each of the considered systems, we analyzed between 10 and 30 consecutive tagged releases. This results in yielding 280 versions which spans 5 to 10 years of maintainability per project. Five baseline techniques are evaluated alongside AMORE: Bunch (B1, structural-only), Co-Change clustering (B2,

evolution-aware), Structural–Semantic Hybrid (B3), Graph-Based Intelligent Clustering (B4), and NSGA-III (B5). All of these considered baselines use their published recommended configurations. Further, each of these baselines is applied independently per release with no look-ahead strategy. It means baselines considers only one chosen release during implementation and their outcome in no means depends on previous or future releases.

3.2 Longitudinal Study Design

Three properties distinguish this study from snapshot work commonly considered in literature. Firstly, identical metrics are computed for every consecutive release of every system. This produces time-series data rather than isolated measurements. Secondly, a strict no-look-forward protocol prevents hindsight from inflating early results. It means each release is processed strictly in chronological order. Thirdly, within-system comparisons are used throughout. It ensures that AMORE is always benchmarked against baselines applied to the same system history. Since, repeated measurements within a system violate independence assumptions. Therefore, in this paper system-level summary statistics are derived before hypothesis testing. This generates one observation per system per technique. Non-parametric Wilcoxon signed-rank and Friedman tests are also applied in this paper and Cliff's delta (δ) parameter is reported as the effect size measure.

4. Longitudinal Evaluation Metrics

Let $M^t = \{M_1^t, \dots, M_k^t\}$ denote the modularization at release t . Also, let C be the set of classes common to two consecutive releases.

4.1 Modular Stability Index (MSI)

The proposed MSI metric measures the proportion of classes that retains their module assignment between consecutive releases. High MSI indicates resilient modular boundaries. In contrast, a declining trend signals progressive architectural instability. Mathematically, this metric is expressed as follows:

$$MSI_{t, t+1} = \frac{1}{|C|} \cdot \sum_{c \in C} \eta[module(c, t) = module(c, t + 1)]$$

Where $\eta[\cdot] = 1$ iff class c belongs to the same module in both successive releases. The term $module(c, x)$ denotes the module to which class c belongs in release x . Also, $|C|$ represents number of classes present in both consecutive releases.

4.2 Change Localization Ratio (CLR)

CLR quantifies what fraction of inter-release changes remain inside module boundaries. A higher CLR indicates effective change containment and reduced ripple effects:

$$CLR_{t, t+1} = \frac{\sum_i \Delta intra(M_i^t, t \rightarrow t + 1)}{\sum_i [\Delta intra(M_i^t, t \rightarrow t + 1) + \Delta inter(M_i^t, t \rightarrow t + 1)]}$$

Where, the term $\Delta intra(M_i^t, t \rightarrow t + 1)$ denotes the number of dependency changes between release t and $t+1$ that are confined within module M_i . The term $\Delta inter(M_i^t, t \rightarrow t + 1)$ represents the number of dependency changes that crosses module boundaries.

4.3 Predictive Maintainability Index (PMI) Trend

PMI is a composite maintainability score computed per release $\{PMI(1), PMI(2), \dots, PMI(T)\}$. It produces a time series $PMI(t)$. Further, a linear regression model $PMI(t) = \alpha \cdot t + \beta$ is fitted across this time series. Here, the slope α is the key indicator of long-term maintainability direction. A flat or negative slope means maintainability is stable or improving. A rising slope reflects accumulation of architectural technical debt.

4.4 Architectural Drift Index (ADI)

ADI measures the loss of architectural information between releases using a normalized mutual-information formulation over module entropy. ADI approaching 0 means strong alignment across releases; ADI approaching 1 indicates near-total drift. Mathematically, it is given by following formula:

$$ADI_{t, t+1} = \frac{H(M^t) + H(M^{t+1}) - 2I(M^t; M^{t+1})}{H(M^t) + H(M^{t+1})}$$

Where, the term $H(M^t)$ represents the modular entropy at release t . It measures the uncertainty (disorder) in the module size distribution. Further, the term $I(M^t; M^{t+1})$ represents the mutual information between two modularizations and it quantifies how much knowing a class's module at release t tells you about its module at release $t+1$.

5. Experimental Results

5.1 Modular Stability across Releases

Table 1 presents average MSI values per system across all consecutive release pairs. AMORE achieves an overall mean MSI of **0.81** (range 0.78–0.84), compared with 0.74 for NSGA-III — the strongest baseline — 0.70 for the hybrid approach, and 0.62 for structural-only Bunch. This translates to 16–31% more classes retaining their module membership per release transition when AMORE is used.

Table 1: Average Modular Stability Index (MSI) across consecutive releases (higher is better)

System	B1 (Struct.)	B2 (Evo.)	B3 (Hybrid)	B4 (Graph)	B5 (NSGA-III)	AMORE
S1 – Commons Lang	0.61	0.66	0.69	0.71	0.73	0.79
S2 – Commons Coll.	0.62	0.67	0.70	0.72	0.74	0.80
S3 – Apache Tomcat	0.64	0.69	0.71	0.73	0.75	0.83
S4 – Spring Core	0.63	0.68	0.70	0.72	0.74	0.81
S5 – Hibernate ORM	0.64	0.69	0.71	0.73	0.75	0.82
S6 – JUnit	0.60	0.65	0.68	0.70	0.72	0.78
S7 – Elasticsearch	0.63	0.68	0.70	0.73	0.76	0.84
S8 – Apache Maven	0.62	0.67	0.70	0.72	0.74	0.80
S9 – Apache Ant	0.61	0.66	0.69	0.71	0.73	0.79
S10 – Kafka	0.62	0.67	0.69	0.72	0.75	0.82
S11 – Log4j	0.60	0.65	0.68	0.70	0.72	0.78
S12 – Apache POI	0.63	0.68	0.70	0.72	0.74	0.81
S13 – Jetty	0.64	0.69	0.71	0.73	0.75	0.82
S14 – Guava	0.62	0.67	0.70	0.72	0.74	0.80
S15 – Neo4j	0.64	0.70	0.72	0.74	0.76	0.83
Average	0.62	0.68	0.70	0.72	0.74	0.81

The longitudinal trajectory supports this observation. All of the considered five baselines show a monotonically declined MSI metric value across different releases. This results clearly indicates of progressive architectural instability. In case of AMORE, we observed that decay curve is notably flatter in nature especially for fast-evolving systems such as Apache Tomcat, Elasticsearch, and Neo4j. This clearly indicates that MSI actually stabilizes in later releases of these systems. The Wilcoxon test confirms statistical significance ($p < 0.05$, Cliff's $\delta = 0.47$ – 0.61 , large effect).

5.2 Change Localization and Architectural Drift

Table 2 shows average CLR and ADI for all techniques. AMORE achieves a mean CLR of **0.74**. It is roughly three-quarters of all dependency changes remain within module boundaries as compared with 0.69 for NSGA-III. Further, it is only 0.58 for structural-only clustering. On the ADI side, AMORE achieves a mean of **0.19** versus 0.25 for NSGA-III and 0.39 for Bunch, representing a 23–39% reduction in architectural drift. All baseline techniques show a monotonically increasing ADI trend across releases; AMORE's ADI trajectory remains flat even in large rapidly-changing systems.

Table 2: Average CLR and ADI per technique- CLR: higher better; ADI: lower better

Technique	Avg. CLR	Avg. ADI
B1 – Structural-only	0.58	0.39
B2 – Evolution-aware	0.63	0.34
B3 – Hybrid	0.65	0.31
B4 – Graph-based	0.67	0.28
B5 – NSGA-III	0.69	0.25
AMORE	0.74	0.19

Notably, AMORE's CLR improves over successive releases in several fast-moving systems like Elasticsearch and Apache Kafka — something no baseline achieves. This suggests that adaptive semantic and behavioral weighting causes module boundaries to progressively align with the evolving functional structure of the system, rather than simply lagging behind it.

5.3 Long-Term Maintainability Trends

Table 3 tracks average PMI values at sampled releases from R1 to R12. The structural-only baseline climbs from PMI 0.28 at R1 to 0.39 by R12. This clearly indicates a substantial accumulation of maintainability debt. NSGA-III performs best among the considered baselines in this paper. Its values shift from 0.25 to 0.29. AMORE begins at **0.24** and remains stable until R7. From R7 onwards, it actually *improves* to 0.23 and these improvements again stabilizes upto R12. In our longitudinal investigation, this is the only technique that deliver a negative PMI slope across the entire observation window. Further, the divergence with baselines widens as releases accumulate. This confirms that AMORE's advantage compounds over time rather than narrowing.

Table 3: Predictive Maintainability Index (PMI) across releases- lower is better

Release	Structural	Evo-Aware	Hybrid	Graph	NSGA-III	AMORE
R1	0.28	0.27	0.26	0.25	0.25	0.24
R3	0.30	0.28	0.27	0.26	0.25	0.24
R6	0.33	0.29	0.28	0.27	0.26	0.24

R9	0.36	0.31	0.30	0.29	0.28	0.23
R12	0.39	0.32	0.31	0.30	0.29	0.23

5.4 Statistical Validation

Table 4 summarizes the statistical testing results across all four metrics. Every comparison between AMORE and each of the five baselines reaches significance at $p < 0.05$. Effect sizes range from medium (PMI slope, CLR against weaker baselines) to large (MSI and ADI across the board). Cross-metric correlation analysis further shows that systems with higher MSI also tend to show higher CLR and lower ADI — boundary stability, change containment, and architectural resilience mutually reinforce each other. AMORE improves all four simultaneously.

Table 4: Statistical significance summary (AMORE vs. all baselines)

Metric	Test	p-value	Cliff's δ	Effect size
MSI	Wilcoxon	< 0.05	0.47–0.61	Large
CLR	Wilcoxon	< 0.05	0.38–0.55	Medium–Large
PMI Slope	Wilcoxon	< 0.05	0.41–0.52	Medium
ADI	Wilcoxon	< 0.05	up to 0.60	Large

6. Conclusion

This paper presented the first large-scale longitudinal experimental evaluation of an adaptive multi-dimensional software remodularization framework. Across 280 releases from 15 open-source systems, AMORE consistently and significantly outperforms five state-of-the-art baseline techniques on every longitudinal dimension measured — boundary stability, change localization, maintainability trajectory, and architectural drift.

The central message is plain: snapshot-based evaluation is not enough. Improvements visible at a single release can mask deterioration over time. AMORE's results show the opposite pattern — advantages that compound as evolution continues, and modular boundaries that stabilize rather than erode. Future work will extend evaluation to polyglot systems, investigate developer-in-the-loop refinement, and explore lightweight deployment variants for continuous integration environments.

REFERENCES

- [1] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch tool," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 193–208, 2006.
- [2] W. Mkaouer et al., "Many-objective software remodularization using NSGA-III," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, pp. 1–45, 2015.
- [3] J. A. Carruthers, J. A. Diaz-Pace, and E. Irrazábal, "A longitudinal study on the temporal validity of software samples," *Inf. Softw. Technol.*, vol. 168, pp. 107404, 2024.
- [4] D. Sas, P. Avgeriou, and U. Uyumaz, "On the evolution and impact of architectural smells — an industrial case study," *Empir. Softw. Eng.*, vol. 27, no. 4, pp. 86, 2022.
- [5] R. Singh, G. Khekare, S. K. Gupta, D. S. Abdelminaam, D. M. Alsekait, and A. Nabil, "AMORE: Adaptive Multi-Dimensional Similarity and Clustering Framework for Remodularization," *Inf. Softw. Technol.*, vol. XXX, no. XX, 2026.
- [6] T. Mens, A. Serebrenik, and A. Cleve, *Evolving Software Systems*. Springer, 2014.
- [7] J. Garcia et al., "Identifying architectural bad smells," in *Proc. CSMR*, 2009, pp. 255–258.

- [8] F. A. Fontana et al., "On evaluating the impact of the refactoring of architectural problems on software quality," in Proc. SANER, 2016, pp. 1–8.
- [9] H. Mumtaz, P. Singh, and K. Blincoe, "A systematic mapping study on architectural smells detection," J. Syst. Softw., vol. 173, pp. 110885, 2021.
- [10] D. Sas, P. Avgeriou, and U. Uyumaz, "On the evolution and impact of architectural smells — an industrial case study," Empir. Softw. Eng., vol. 27, no. 4, pp. 86, 2022.
- [11] P. Gnoyke, S. Schulze, and J. Krüger, "Evolution patterns of software-architecture smells," J. Syst. Softw., vol. 217, pp. 112170, 2024.
- [12] I. Candela et al., "Using cohesion and coupling for software remodularization: Is it enough?" ACM Trans. Softw. Eng. Methodol., vol. 25, no. 3, pp. 1–28, 2016.
- [13] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in Proc. ICSE, 2003, pp. 125–135.
- [14] T. Zimmermann et al., "Mining version histories to guide software changes," IEEE Trans. Softw. Eng., vol. 31, no. 6, pp. 429–445, 2005.
- [15] A. Prajapati, A. Parashar, and A. Rathee, "Multi-dimensional information-driven many-objective software remodularization approach," Front. Comput. Sci., vol. 17, no. 3, pp. 173209, 2023.
- [16] F. Palomba et al., "A large-scale empirical study on the lifecycle of code smell co-occurrences," Inf. Softw. Technol., vol. 99, pp. 1–10, 2018.
- [17] R. Jolak et al., "An empirical investigation of the impact of architectural smells on software maintainability," J. Syst. Softw., vol. 225, pp. 112382, 2025.