DOI: https://doi.org/10.15276/hait.09.2026.23
UDC 004.415:004.4'24

A method for structuring functional and technical domains of an information system based on a project change history

Vitalii S. Halamai[1)]
ORCID: https://orcid.org/0009-0001-9960-4263; vitalii.s.halamai@lpnu.ua
Viktor A. Melnyk[2)]
ORCID: https://orcid.org/0000-0002-5046-8002; viktor.melnyk@kul.pl. Scopus Author ID: 57200786767
[1)] Lviv Polytechnic National University, 12, Stepan Bandera Str. Lviv, 79013, Ukraine
[2)] The John Paul II Catholic University of Lublin, 14, Al. Racławickie, Lublin, 20-950, Poland

ABSTRACT

Relevance: Information systems continuously evolve through commits and iterative modifications, which gradually reduce architectural transparency and blur responsibility boundaries between components. In long-lived systems, static architectural documentation often becomes outdated, while repository change history preserves objective evidence about how artifacts evolve together and how evolutionary dependencies emerge over time. **Aim of the article**: The paper aims to develop a method for the automatic structuring of the functional and technical domains of an information system, based on a project change history and evolutionary coupling signals extracted from repository data. **Tasks**. The research tasks include formalizing the change history model, defining evolutionary cohesion indicators, constructing an integrated evolutionary cohesion graph, developing a graph-based domain grouping procedure, differentiating recovered domains into functional and technical responsibility areas, and defining the output representation and validation strategy. **Methods**: The proposed research combines repository mining, evolutionary coupling analysis, graph-based clustering, semantic similarity analysis, temporal cohesion analysis, and dependency-aware modeling. The approach integrates co-change frequency, temporal proximity, semantic similarity, and structural dependency signals into a unified evolutionary cohesion measure represented as a weighted graph. **Scientific novelty**: The scientific novelty of the research reflected in the paper lies in recovering domain structures not from static architecture or documentation, but from observable evolutionary behavior recorded in repository history. The proposed method introduces an integrated evolutionary cohesion measure and an explainable differentiation mechanism that separate recovered domains into functional and technical responsibility areas based on ownership dispersion, reuse index, and cross-domain connectivity indicators. **Practical significance**: The proposed method improves architectural transparency and supports maintenance planning, refactoring, change impact analysis, technical debt management, and responsibility reasoning in evolving information systems. The resulting domain map and interaction graph can assist developers in identifying highly coupled areas and understanding evolutionary dependencies between subsystems. **Results**. The paper presents a complete pipeline for transforming repository evolution data into an interpretable domain structure model. The method constructs an evolutionary cohesion graph, identifies candidate domains through graph-based grouping, and classifies them into functional and technical areas using explainable indicators. An illustrative example demonstrates how stable co-evolution patterns form interpretable domain areas even without explicit architectural documentation. **Conclusions**. The proposed method enables recovering interpretable functional and technical domain structures from a project change history by combining co-change, semantic, dependency-aware, and temporal signals. The resulting domain decomposition supports architectural understanding and maintenance-oriented reasoning while reducing dependence on incomplete or outdated architectural documentation. Full empirical validation and further enhancement of classification and weighting strategies remain directions for future research.

Keywords: Evolutionary coupling; community detection; repository mining; cohesion graph; domain decomposition; functional domains; technical domains

***For citation*: Halamai V. S., Melnyk V. A. "A method for structuring functional and technical domains of an information system based on a project change history". *Herald of Advanced Information Technology*. 2026. Vol. 9. No. 3: 351–365. DOI: https://doi.org/10.15276/hait.09.2026.23**

INTRODUCTION

Information systems during the development cycle are continuously modified through commits. As a side effect, this may gradually reduce architectural transparency and blur component responsibility boundaries. Version control and change history, therefore, provide an empirical trace of this process, indicating which artifacts are modified together and when evolutionary dependencies emerge.

Recovering domain structure from change history is practically useful for maintenance, change planning, refactoring, technical debt management, and risk control. In this work, domains are treated as groups of artifacts with a shared role and strong internal evolutionary cohesion, identifying an evolution-aware decomposition rather than a purely static modular view.

Functional and technical decomposition often remains undocumented or becomes outdated as systems evolve through continuous commits.

Along with the progress, as architectural boundaries erode, responsibility separation becomes blurred, making it harder to assess change impact, establish ownership, and control cross-cutting technical dependencies.

© Halamai V., Melnyk V., 2026
This is an open access article under the CC BY license (https://creativecommons.org/licenses/by/4.0/deed.uk)

In practice, this increases maintenance effort, raises the risk of unintended side effects, and accelerates technical debt accumulation. Typical approaches that rely only on static structure or only on co-change signals often produce clusters that are difficult to interpret and do not clearly separate functional responsibilities from technical infrastructure. The key research gap is therefore a method that derives domains from evolutionary signals and explains the resulting domains as functional or technical areas. Unlike many existing approaches that focus primarily on structural decomposition or isolated co-change relations, the proposed method integrates heterogeneous evolutionary signals and provides explainable, responsibility-oriented domain interpretation.

A domain [1] refers to a group of code artifacts with a shared role, exhibiting stronger internal evolutionary cohesion compared to its external relationships. Two domain classes are considered: functional domains (business logic and application-level functionality) and technical domains (infrastructure, shared services, and supporting modules). This binary separation reflects a widely adopted architectural distinction between components that implement business capabilities and those that provide cross-cutting infrastructural support. It represents a minimal yet actionable classification that improves interpretability for maintenance and change impact reasoning.

The aim of the research work presented in this paper is to design a method that extracts evolutionary cohesion features from change history (co-change patterns, frequencies, semantics, dependencies, and temporal signals), after which system components should be grouped into domains. The relationships between components should be represented as an evolutionary cohesion graph, where dense subgraphs correspond to candidate domains. Unlike simple clustering, the grouping relies on an integrated evolutionary connectivity measure rather than a single signal, reflecting the shared evolution of components. After domain structuring, the resulting clusters should be interpreted as functional or technical areas using criteria such as ownership, shared infrastructure roles, and connectivity structure, supported by explainable features. This domain structure supports system evolution and architectural maintenance by clarifying component responsibilities, improving change impact analysis, and facilitating coordinated development.

RELATED WORKS

Architecture recovery is strongly motivated by the long-term phenomenon of architectural degradation [2], in which rapid evolution and frequent modifications gradually erode the intended decomposition and render structural reasoning unreliable. Classical recovery work frames the problem as clustering a system-wide dependency graph and asks what relations are sufficient to obtain a stable view, highlighting that a small set of relations [3] can sometimes approximate a more complete dependency portfolio. At the clustering level, empirical evaluation of modularity-based community detection [4] shows that greedy multi-level modularity clustering can achieve competitive accuracy while scaling to large systems. More recent recovery methods expand the evidence base: ensemble approaches combine structural, semantic, and directory dependencies [5] to reduce information loss that occurs when heterogeneous signals are merged prematurely. Multi-model recovery further demonstrates the benefit of explicitly integrating multiple dependency models and aggregating clustering results to improve MoJoFM similarity [6] and expose architectural anomalies. In parallel, modularization studies that combine clustering with lexical representations evaluate alternative pipelines through cohesion and coupling metrics [6], while specialized clustering strategies aim to improve modularization quality (e.g., weighted modularity quality [7]) and address practical issues such as overly large clusters.

A complementary line of work focuses on evolution and maintenance, where change impact analysis [8] is treated as a central maintenance activity that reduces cost and risk by estimating how a modification propagates across artifacts. Within this area, evolutionary coupling research shows that co-changes are not uniform events: fine-grained analysis identifies distinct co-change relationship types [9] that can reflect structural coupling, semantic coupling, or implicit dependencies, and these relations can shift across phases of system evolution. Several methods, therefore, aim to strengthen co-change signals beyond raw frequency. One direction integrates evolutionary coupling with additional change relationships [10] (e.g., structural and clone relations) to improve ranking-based impact prediction when historical co-changes are sparse. Another direction uses deep representations, showing that semantic code embeddings such as GraphCodeBERT [11] help capture non-obvious dependencies and maintain performance even at larger recommendation windows. Evolutionary

coupling is applied beyond change recommendation as well: co-evolution between tests and code can approximate relevance links for debugging when dynamic data is unavailable, using test–code evolutionary coupling [12] as a static signal for fault localization. Together, these works motivate using change history not only for prediction, but for constructing more interpretable structures that support maintenance reasoning.

Graph-based grouping methods connect these strands by treating recovery as community detection over dependency graphs and exploring which dependency configurations yield interpretable clusters. Recent evidence suggests that code co-changes can be treated as logical dependencies [13] that complement structural relations, but that raw co-change edges require filtering to reduce noise from unrelated commits; experiments show that combining structural and logical dependencies improves clustering outcomes under MQ and MoJoFM, while logical-only graphs can achieve strong modularity yet provide incomplete coverage [13]. Network-based approaches similarly model monolithic systems as dependency graphs and apply community detection (e.g., Louvain [14]) as decision support, emphasizing that multiple clustering perspectives still require expert validation. Because clustering quality depends on project characteristics and architectural issues, large-scale studies highlight the importance of selecting an appropriate technique and dependency type; CLUE addresses this by using ML-based customization and linking outcomes to architectural smells [15]. Graph clustering is further used as an enabling step for modernization: automated extraction of microservices combines community detection with optimization to balance cohesion and coupling [1] and to align service granularity with domain entities. Across these works, the recurring gap is that clustering outputs are often treated as structural decompositions, while the semantic meaning of recovered clusters remains underspecified.

Finally, prior research provides conceptual grounding for distinguishing functional and technical structure through pattern- and concern-oriented views. Layered architecture reconstruction studies show that generic criteria such as cohesion and coupling can lack precision for identifying architectural layers, and therefore, inventory layering principles [16] and explicit rules (e.g., abstraction and responsibility) to guide reconstruction and technical debt detection. Cross-cutting concern analysis complements this view by showing that secondary concerns, such as logging or database access, become scattered and tangled [17], contributing to modularity loss; metadata-based techniques track such concerns through third-party components and quantify dedication using Dedication to Concern (DtC) [17]. Related clustering-based layer detection work proposes unsupervised recovery of MVC layers using hybrid structural and lexical features [18], reporting that results depend on naming consistency and still require manual interpretation. Collectively, these studies support the need for recovery methods that move beyond packaging-level clusters toward responsibility-oriented interpretations, where technical, cross-cutting support structures can be separated from business-facing functional areas in an explainable way.

PROBLEM STATEMENT

Recent advances in architecture recovery demonstrate strong progress through clustering-based modularization, multi-dependency integration, and evolutionary coupling analysis. However, important gaps remain when the objective is to recover interpretable domain structures that distinguish functional responsibility areas from technical support concerns.

Current modularization approaches often focus on producing cohesive clusters, but they do not explicitly provide an interpretation of the resulting groups as functional or technical domains. Co-change relations provide useful evolutionary signals, yet they remain noisy and require filtering and integration with additional evidence to reduce coincidental associations. Moreover, recovery outcomes strongly depend on the selected relation sets and clustering strategies, which limit their applicability across different projects.

The separation of functional and technical concerns remains particularly under-explored. Generic cohesion or coupling criteria often fail to reflect engineering intent, especially in layered systems and in the presence of cross-cutting infrastructural mechanisms. As a result, recovered clusters frequently require substantial expert interpretation rather than providing explainable responsibility boundaries.

This paper aims to develop a method for automatically reconstructing the domain structure of an information system from change history and differentiating the identified domains into functional and technical responsibility areas. To achieve this aim, the paper defines the input model of change history, formulates evolutionary cohesion indicators, constructs an integrated cohesion graph, specifies a graph-based domain grouping procedure, introduces

an explainable functional-technical classification mechanism, and outlines the validation strategy. The result is presented as an interpretable domain map and an interaction graph that support architectural understanding and maintenance-oriented reasoning.

The specific research tasks required to achieve this aim are formalized in the following subsection.

PROPOSED METHOD FOR STRUCTURING FUNCTIONAL AND TECHNICAL DOMAINS OF AN INFORMATION SYSTEM

This section describes the complete pipeline of the proposed method, from extracting evolutionary signals in change history to constructing an interpretable domain structure model.

TASK FORMULATION AND METHOD OVERVIEW

The main task is to automate the structuring of the functional and technical domains of an information system based on its change history during the development cycle. The novelty of the approach lies in deriving domains not from static architecture or documentation, but from evolutionary change patterns observed in the repository.

The method takes repository evolution data (commits, modified files, metadata) as input and produces a domain structure model that maps components to functional and technical domains.

To achieve this aim, the following research tasks are formulated:

1) to formalize the input data model of change history, including commits, artifacts, authors, timestamps, and commit messages, and to define the granularity of analyzed system artifacts;

2) to justify the set of evolutionary cohesion indicators used for domain recovery, including co-change frequency, temporal proximity, semantic similarity, and dependency-aware signals;

3) to develop an integrated evolutionary cohesion measure that combines heterogeneous evidence sources and represents relationships between system artifacts as a weighted evolutionary cohesion graph;

4) to define a graph-based domain grouping procedure that partitions system artifacts into candidate responsibility domains based on stronger internal evolutionary cohesion and weaker external connectivity;

5) to define an explainable procedure for differentiating recovered domains into functional and technical responsibility areas using domain-level indicators such as ownership dispersion, reuse index, and cross-domain connectivity;

6) to specify the output representation of the method as a domain map and interaction graph that can support architectural understanding, change impact reasoning, and maintenance planning;

7) to define the validation strategy for assessing the proposed method, including comparison with proxy architectural references, expert assessment, and sensitivity analysis under different thresholds, feature weights, and historical windows.

The pipeline of the proposed method is depicted in Fig. 1. It is organized into successive stages that transform raw commit-level evolution data into a graph-based decomposition and an interpretable classification of domains. Each stage produces intermediate artifacts that support transparency and practical applicability in real-world repositories.

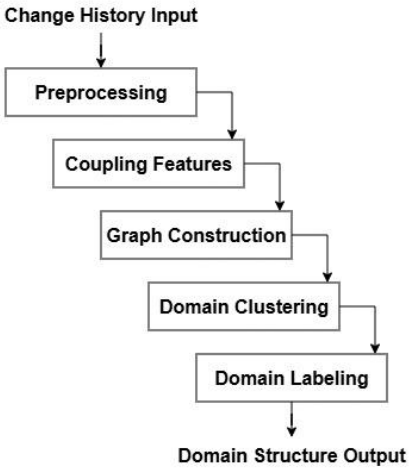


Fig. 1. **Pipeline transforming repository evolution data into an interpretable functional and technical domain structure model**
Source: **compiled by the authors**

Formally, the goal is to partition the set of code artifacts into cohesive groups that represent responsibility areas, while distinguishing between business-oriented and infrastructure-oriented domains. Artifacts may correspond to files, classes, packages, or other modular units, depending on the granularity of the analyzed system. The objective is not only to maximize cohesion within domains, but also to minimize unnecessary cross-domain coupling, thereby improving maintainability and supporting architectural reasoning.

The domain structuring task is considered correctly fulfilled when the resulting partition satisfies several evaluation conditions. First, artifacts inside the same domain should demonstrate stronger evolutionary cohesion than artifacts assigned to different domains. Second, the recovered domains should be sufficiently consistent with available proxy architectural references, such as packages,

modules, architectural layers, or ownership boundaries. Third, the functional or technical interpretation of each domain should be explainable through domain-level indicators, including ownership dispersion, reuse index, and cross-domain connectivity. Finally, the resulting structure should remain reasonably stable under moderate changes in cohesion thresholds, feature weights, and analyzed time windows. These conditions define whether the recovered domain model is not only formally constructed, but also meaningful for architectural understanding and maintenance-oriented reasoning.

Let $A = \{a_1, a_2, \dots, a_n\}$ denote the set of analyzed artifacts (files, classes, or packages), and $C = \{c_1, c_2, \dots, c_m\}$ denote the set of commits extracted from the repository history.

Each commit is represented as:

$$C_k = (T_k, A_k, U_k, M_k), \quad (1)$$

where T_k is the timestamp; $A_k \subseteq A$ is the set of modified artifacts; U_k is the author; M_k is the commit message.

The objective of domain structuring is to construct a partition $P = \{D_1, D_2, \dots, D_k\}$ such that:

$$\bigcup_{k=1}^{K} D_k = A, D_i \cap D_j = \emptyset (i \neq j), \quad (2)$$

where each D_k represents a candidate responsibility domain.

The partition is derived from evolutionary cohesion relations computed between artifacts.

CHANGE HISTORY AS INPUT DATA

The project change history stored in a version control system serves as the primary data source for the proposed method. Commits reflect which components were modified, when the changes occurred, and by whom they were introduced. The history is treated as a structured set of related entities, including the repository, commits, files, and authors.

The method does not require storing or processing the complete repository history in all cases. Instead, the analyzed change history can be limited by a configurable historical window, defined either by the most recent *N* commits or by a selected time interval. This restriction reduces data collection and processing costs, which is important when repository data are obtained through external services with API rate limits. It also helps avoid overemphasizing obsolete evolutionary relations that may no longer reflect the current system structure, development practices, or team composition.

In practice, commits are preprocessed to exclude irrelevant noise introduced by non-functional or automated changes. Artifacts can be analyzed at different abstraction levels (file-level or package-level), allowing the method to adapt to both fine-grained and coarse-grained architectural recovery settings. Automated collection and structuring of change history is assumed, ensuring the practical applicability of the approach to real-world repositories.

Regular co-changes between components may indicate their belonging to the same functional or technical domain. The method assumes sufficiently localized change history and consistent development practices to ensure that evolutionary patterns remain informative. Centralized merge commits may reduce the usefulness of co-change and authorship signals [19] by introducing noise into evolutionary dependencies. Therefore, optional filtering strategies can be applied to down-weight or exclude such commits when constructing evolutionary coupling relations.

In the preprocessing step, quantitative and semantic features are extracted from the change history to capture evolutionary relationships between components. These features combine frequency-based, temporal, semantic, and dependency-aware evidence, providing a richer representation of evolutionary cohesion than co-change counts alone.

Evolutionary coupling [20] is defined as the degree of interrelatedness between artifacts based on their shared evolution during development. Change frequency is computed for each component to highlight actively evolving and potentially critical areas. For each pair of artifacts (a_i, a_j) evolutionary relations are quantified through multiple normalized signals. Co-change frequency can be defined as:

$$f_{ij} = |\{ c_k \in C \mid a_i, a_j \in A_k \}|. \quad (3)$$

Co-change frequency measures how often two components are modified together within the same commits, which may indicate a shared functional responsibility.

Coupling strength is derived by normalizing co-change counts to distinguish stable evolutionary patterns from coincidental modifications.

Normalized co-change strength can be defined as:

$$cc_{ij} = \frac{f_{ij}}{|\{c_k \mid a_i \in A_k\}| + |\{c_k \mid a_j \in A_k\}| - f_{ij}}. \quad (4)$$

Let ΔT_{ij} denote the average time interval between shared modifications. Temporal cohesion can be defined as exponential decay:

$$temp_{ij} = e^{-\lambda \Delta T_{ij}}, \lambda > 0, \quad (5)$$

where $\lambda > 0$ is a decay rate that controls how quickly the cohesion weight decreases over time.

EVOLUTIONARY COUPLING FEATURES EXTRACTION

After the change history is collected and preprocessed, the method proceeds to feature extraction as the first analytical step toward domain recovery. At this stage, the repository evolution is transformed into a set of measurable signals that describe how system components co-evolve over time.

Semantic similarity is incorporated through contextual information from commit messages or functionality-related terms, allowing the method to detect latent relationships even when direct co-change evidence is sparse.

Let $\emptyset(a_i) \in R^d$ denote a semantic embedding derived from commit messages or code tokens. Semantic similarity can be computed using cosine similarity:

$$sem_{ij} = \frac{\emptyset(a_i) \cdot \emptyset(a_j)}{\| \emptyset(a_i) \| \| \emptyset(a_j) \|}. \quad (6)$$

Dependency-aware signals further refine these relations by accounting for structural connections between components. Let $dep_{ij} \in [0,1]$ represent the normalized structural dependency strength between artifacts. The overall evolutionary cohesion between artifacts can be defined as the integrated evolutionary cohesion measure:

$$w_{ij} = a_1 cc_{ij} + a_2 temp_{ij} + a_3 sem_{ij} + a_4 dep_{ij}, \quad (7)$$

where $a_k \geq 0, \sum_{k=1}^{4} a_k = 1$ and dep_{ij} denotes the normalized structural dependency strength between artifacts, derived from static analysis of call, import, inheritance, or usage relations. Let D denote the set of structure dependencies. Then:

$$dep_{ij} = \begin{cases} 1 & if (a_i, a_j) \in D \\ 0 & else \end{cases}. \quad (8)$$

Note that $dep_{ij} = 0$ does not mean the lack of relation, but just the lack of explicit structural dependency.

This integrated measure reflects persistent multi-dimensional evolutionary connectivity rather than a single dependency signal.

Temporal coupling strengthens links that occur within short time intervals, reflecting tighter evolutionary coordination. The coupling measures emphasize persistent long-term relations rather than short-lived or accidental co-modifications.

The output of this stage is a set of weighted evolutionary links between components, which serves as the basis for subsequent graph construction and domain grouping.

GRAPH CONSTRUCTION AND DOMAIN GROUPING

Domain areas of the information system are structured by grouping components based on evolutionary coupling features. The weighted relationships obtained in the previous step are organized into a structural representation reflecting the intensity of shared evolution. This step effectively transforms the repository history into an evolutionary cohesion graph, where weak and noisy dependencies can be filtered out to highlight the most informative relations. System components are modeled as a graph, where edges capture the strength of evolutionary cohesion between artifacts. It is assumed that densely connected subsets of nodes correspond to potential functional or technical domains within the system.

The system is represented as a weighted, undirected evolutionary cohesion graph:

$$G = (A, E, W), \quad (9)$$

where E denotes the set of edges representing sufficiently strong evolutionary cohesion relations between artifacts, filtered by the filtering threshold τ: $E = \{ (a_i, a_j) \mid w_{ij} > \tau \}$, and $W(a_i, a_j) = w_{ij}$ denotes the strength of evolutionary cohesion. It means that an edge between artifacts a_i and a_j exists only if their integrated evolutionary cohesion w_{ij} exceeds a threshold τ. The threshold τ is treated as a configurable filtering parameter that controls the density of the evolutionary cohesion graph. Its value can be selected using one of three strategies depending on the available validation data and project characteristics.

First, τ may be defined empirically as a percentile of all non-zero cohesion weights, for example, by keeping only the strongest 20–30% of artifact relations.

Second, τ may be calibrated against proxy architectural references, such as package boundaries, modules, or ownership structures, by selecting the value that provides the best structural consistency.

Third, τ may be examined through sensitivity analysis, in which several threshold values are tested, and the stability of the resulting domains is compared.

In this work, τ is not treated as a fixed universal constant, because the distribution of cohesion weights depends on repository size, commit granularity, and development practices.

Domain structuring is formulated as a graph partitioning problem that maximizes internal cohesion and minimizes cross-domain connectivity. One possible objective function is a modularity maximization:

$$max_P Q(P), \quad (10)$$

$$Q(\mathcal{P}) = \frac{1}{2m}\sum_{ij}(w_{ij} - \frac{d_i d_j}{2m})\,\delta\big(D(a_i), D(a_j)\big), \quad (11)$$

where $D(a_i)$ denotes the domain assignment of the artifact a_i; d_i is the weighted degree of artifact a_i in the evolutionary cohesion graph; m is the total edge weight in the graph; and $\delta(\cdot,\cdot)$ is the Kronecker delta:

$$\delta(D(a_i), D(a_j)) = \begin{cases} 1, & \text{if } D(a_i) = D\big(a_j\big), \\ 0 & \text{- otherwise.} \end{cases} \quad (12)$$

Before grouping, coupling thresholds may be applied to remove coincidental links, improving the robustness of domain structuring. Domain grouping may then be performed using community detection or clustering techniques that partition the evolutionary cohesion graph into candidate domain areas. In the baseline configuration, modularity-based community detection approaches can be used as a practical grouping strategy due to their scalability and suitability for weighted evolutionary cohesion graphs. The resulting clusters represent candidate responsibility areas, which can be further analyzed or compared against existing modular boundaries in the system.

The method remains independent of a specific grouping algorithm. The graph is built on an integrated evolutionary cohesion measure, so the resulting domains reflect shared evolution rather than formal structural separation.

Scalability considerations include evaluating how system size and graph density affect the complexity of graph construction and grouping procedures, which is critical for large projects. The output of this stage is a set of candidate domain areas, representing an evolutionary structural decomposition of the system.

DIFFERENTIATION BETWEEN FUNCTIONAL AND TECHNICAL DOMAINS

After candidate domain areas are identified, they are interpreted and structured into functional and technical domains of the information system. Automatic grouping reveals organization, but practical usage requires structuring, which areas correspond to business functionality, and which represent infrastructural subsystems.

Functional domains are associated with application logic, business processes, and the system's problem domain. Technical domains cover infrastructure modules, shared services, and supporting components that provide system-wide functionality.

Domain classification relies on combined indicators reflecting evolution, role, interaction patterns, and reuse characteristics. This separation is a key contribution, since most existing approaches produce clusters without functional or technical interpretation. The proposed method provides both domain formation and explainable classification by combining ownership dispersion, infrastructural reuse signals, and cross-domain connectivity.

For each recovered domain D_k the following quantitative indicators are computed.

1. Ownership dispersion, which is defined as:

$$OD_k = \frac{|\,U_k\,|}{\Sigma_{a\in D_k} commit_count(a)}, \quad (13)$$

where U_k is the set of distinct contributors.

2. Reuse index, which is defined as:

$$RI_k = \frac{\Sigma_{i\in D_k, j\notin D_k} w_{ij}}{\Sigma_{i,j\in D_k} w_{ij}}. \quad (14)$$

3. Cross-domain connectivity, which is defined as:

$$CDC_k = \frac{\Sigma_{i\in D_k, j\notin D_k} w_{ij}}{\Sigma_{i,j\in A} w_{ij}}. \quad (15)$$

To formulate the domain type decision function, let z_k denote a feature vector of the domainD_k:

$$z_k = (OD_k, RI_k, CDC_k). \quad (16)$$

The probability of a domain being technical is defined as:

$$P(technical \mid D_k) = \sigma(\beta_0 + \beta^T z_k), \quad (17)$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$; Here β_0 is the intercept term, specifying the model's baseline tendency to attribute domains to technical even when all features are zero, and β is the parameter vector weighting the domain-level indicators:

$$\beta = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}, \quad (18)$$

where β_1 is weight of the ownership dispersion, β_2 is weight of the reuse index, β_3 is weight of the

cross-domain connectivity. The notation β^T denotes the transpose of the vector β. Since

$$z_k = (OD_k, RI_k, CDC_k), \quad (19)$$

thus

$$\beta^T z_k = \beta_1 OD_k + \beta_2 RI_k + \beta_3 CDC_k. \quad (20)$$

The weights of the domain-level indicators can be specified in two ways. In the initial version of the method, they may be assigned expert-based values reflecting the expected contribution of ownership dispersion, reuse, and cross-domain connectivity to the technical role of a domain. In an empirical setting, these parameters can be learned from a small manually labeled subset of domains using weakly supervised or supervised calibration. The threshold value 0.5 is used as a baseline decision boundary for binary classification; however, it can be adjusted during validation to balance false functional and false technical assignments.

A domain is classified as technical if $P(technical \mid D_k) > 0.5$, otherwise as functional.

The correctness of functional and technical domain assignment is expected to be validated against proxy ground truth sources, such as existing package structures, architectural layers, module ownership, and developer or architect assessment. Additional validation can include sensitivity analysis under different threshold values and indicator weights, as well as comparison between automatically assigned labels and manually reviewed domain labels.

Ownership patterns may indicate domain type: technical modules often involve a broader set of contributors, while functional domains may remain more localized. Components that are reused across many areas or implement general mechanisms (logging, data access, security) are treated as candidates for technical domains. Interaction context further supports classification: technical domains typically exhibit a high number of links to other parts of the system, whereas functional domains tend to remain more domain-specific and locally cohesive.

The output of this stage is a domain structure model where each area is interpreted as functional or technical, improving the clarity of the recovered decomposition.

OUTPUT DOMAIN STRUCTURE REPRESENTATION

The identified domains are represented in a structured form suitable for further analysis and practical use. The primary output of the method is a domain map, which assigns information system components to the corresponding functional or technical domains. Each domain entry can be enriched with summary information that supports interpretability. In addition, an interaction graph is constructed to represent relationships and dependencies between domains derived from evolutionary change patterns.

This representation provides a more transparent system decomposition and can support architectural analysis by highlighting critical isolations or excessive coupling between subsystems. Such representations can be integrated into architectural analysis and maintenance workflows, enabling practitioners to prioritize refactoring or risk mitigation actions. The resulting domain model serves as a foundation for maintenance optimization, change planning, refactoring, and support of risk and security management processes. The outputs can be integrated into technical debt management and change impact control.

ILLUSTRATIVE EXAMPLE

A sample scenario illustrates the workflow of domain structuring based on change history and reflects a typical enterprise repository structure that combines application and infrastructural modules. The example is intended solely for methodological illustration of the proposed workflow and does not represent a full empirical validation on an industrial-scale repository. A representative fragment of an information system is considered, including both functional and technical components.

Assume the system consists of the following five modules:

– *OrderProcessing;*
– *PaymentIntegration;*
– *UserMgmt;*
– *DataAccess;*
– *MonitoringAndLogging*.

A small commit history fragment exhibits shared evolution patterns:

– *Commit 1*: modified *OrderProcessing* and *PaymentIntegration*;
– *Commit 2*: modified *OrderProcessing* and *PaymentIntegration*;
– *Commit 3*: modified *UserMgmt*;
– *Commit 4*: modified *DataAccess* and *MonitoringAndLogging*;
– *Commit 5*: modified *DataAccess* and *MonitoringAndLogging*;
– *Commit 6*: modified *DataAccess* and *UserMgmt*

Co-change frequency reveals strong evolutionary links between modules:

– a functional relationship between *OrderProcessing* ($P = 0.2$) and *PaymentIntegration* ($P = 0.3$);

– a technical dependency between *DataAccess* ($P = 0.7$) and *MonitoringAndLogging* ($P = 0.8$).

UserMgmt represents a boundary functional component. While it is assigned to the functional domain, its domain-level indicators show limited proximity to technical infrastructure due to shared service interactions. In this illustrative case, intermediate indicator weighting ($P = 0.5$) reflects its functional membership together with maximal closeness to the technical area. The presented indicator weights and threshold values are not empirically calibrated and are used only to illustrate the interpretation mechanism of the proposed method.

For example, after computing the integrated evolutionary cohesion measure and applying the filtering threshold $\tau = 0.5$, the illustrative evolutionary cohesion graph can be represented as follows:

– *OrderProcessing* - *PaymentIntegration* = 0.86;

– *OrderProcessing* - *UserMgmt* = 0.63;

– *PaymentIntegration* - *UserMgmt* = 0.58;

– *DataAccess* - *MonitoringAndLogging* = 0.82;

– *DataAccess* - *UserMgmt* = 0.46.

With $\tau = 0.5$, the retained graph contains four edges: *OrderProcessing* - *PaymentIntegration*, *OrderProcessing* - *UserMgmt*, *PaymentIntegration* - *UserMgmt*, and *DataAccess* - *MonitoringAndLogging*. The link *DataAccess* - *UserMgmt* is excluded because its weight is below the threshold. The retained edges form two clearly interpretable areas: a functional domain consisting of *OrderProcessing*, *PaymentIntegration*, and *UserMgmt*, and a technical domain consisting of *DataAccess* and *MonitoringAndLogging*. The numerical cohesion values in this example are illustrative and are introduced solely to demonstrate the behavior of the proposed grouping procedure.

Based on these cohesion signals, candidate domains are structured as follows:

– Functional domain: *OrderProcessing* + *PaymentIntegration* + *UserMgmt*;

– Technical domain: *DataAccess* + *MonitoringAndLogging*.

This example demonstrates that stable co-evolution patterns can naturally induce domain areas even in the absence of explicit architectural documentation.

Thus, the illustrative scenario explains the practical logic of the approach and leads to the overview of the proposed method shown in Fig. 2.

Empirical evaluation on real repositories is a part of the next research step, including:

– domain stability assessment under different time windows and thresholds;

– comparison against proxy ground truth (folders, modules, architectural layers, owner teams);

– qualitative expert evaluation through developer surveys or manual labeling of a limited subset.

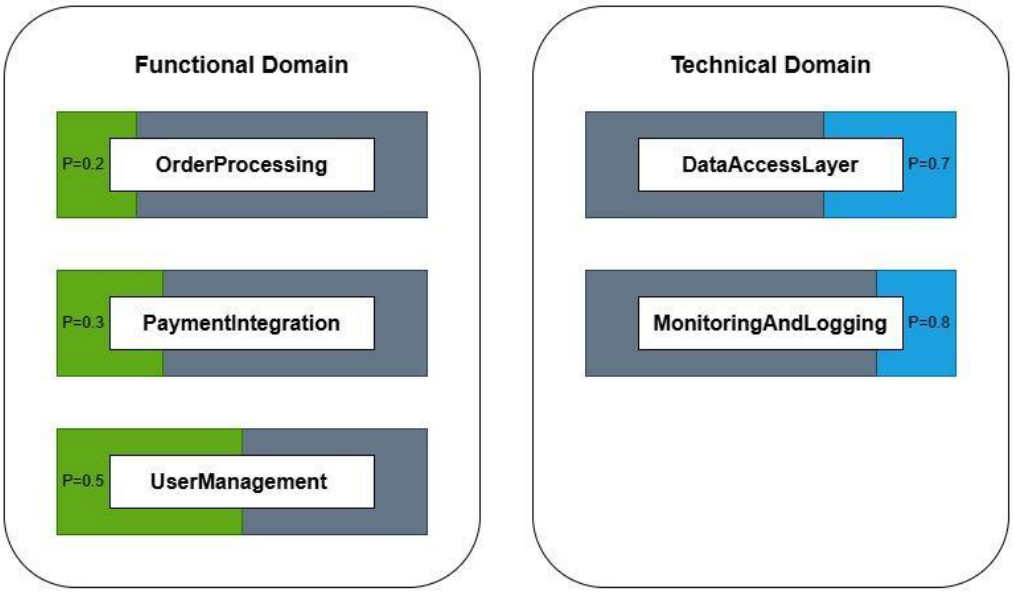


Fig. 2. **Functional and technical domain areas with a boundary module structured at the commit level**
Source: **compiled by the authors**

BENEFITS OF THE PROPOSED METHOD

The recovered domain structure provides several practical benefits for system maintenance and continuous evolution:

– localized change analysis and reduced effort: instead of inspecting the entire system, developers focus on the affected domain and its closest neighbors, significantly narrowing the scope of investigation;

– explainable impact estimation: domains serve as an intermediate abstraction layer for impact analysis, helping predict which component areas are most likely to be influenced by a modification;

– risk-aware maintenance through domain interaction modeling: the domain interaction graph supports an interpretable impact radius, where changes in one domain pose risks to domains with strong evolutionary or structural coupling;

– support for architectural refactoring and remodularization: clusters with high internal evolutionary cohesion indicate natural modular boundaries, while domains with excessive external connectivity reveal architectural smells and blurred responsibilities;

– improved robustness under noisy or incomplete change history: when co-change signals are sparse or unreliable, semantic and dependency-aware features strengthen domain recovery and improve recommendation quality;

– foundation for responsibility and ownership reasoning: the domain map provides a basis for modeling responsibility zones and making ownership dispersion and infrastructural reuse patterns explicit.

COMPARATIVE ANALYSIS OF THE PROPOSED METHOD AGAINST EXISTING APPROACHES

The proposed method is positioned within software architecture recovery, as it derives structural views through dependency-based clustering. However, it differs from classical SAR [4], [21] in objective and output abstraction: instead of primarily producing packaging-level decompositions, it emphasizes domain areas interpreted as functional versus technical responsibility units.

Unlike single-signal modularization [22] studies that mainly optimize cohesion, coupling, or similarity to ground truth, this approach targets explainable domain interpretation, explicitly separating functional from technical domains as an outcome.

While multi-dependency recovery [23] emphasizes integrating heterogeneous dependency models, the proposed method extends this principle at the evolutionary evidence level by combining co-change, semantic, dependency-aware, and temporal signals [24] and interpreting them beyond packaging recovery. Related studies [25] highlight the need for unified structural and semantic representations of software repositories to support system-level analysis and interpretation. Graph-based dependency representations have been applied for software quality and reliability assessment in systems of high architectural complexity.

In contrast to co-change prediction methods that optimize Top-K recommendation accuracy [11], the focus here lies on reconstructing a domain structure that explains why related changes emerge, rather than predicting isolated artifact pairs.

Compared to existing architecture recovery and co-change analysis approaches, the proposed method provides several additional advantages. First, it combines structural, semantic, temporal, and evolutionary evidence within a unified evolutionary cohesion model instead of relying on a single dependency type. Second, the method introduces explainable differentiation between functional and technical domains, improving recovered-structure interpretability for maintenance and architectural reasoning. Third, the approach remains clustering-algorithm independent, which improves adaptability across projects with different repository characteristics and development practices.

Since this paper introduces the proposed method and does not include empirical evaluation, the comparative analysis focuses on qualitative positioning and evaluation criteria that can be used in future validation rather than unsupported numerical results. The proposed method can be compared with architecture recovery, multi-dependency clustering, co-change-based reconstruction, and evolutionary coupling approaches using several metrics: MoJoFM, modularity, and MQ for structural recovery quality; accuracy, precision, recall, F1-score, and expert agreement for functional or technical domain classification; and stability under different cohesion thresholds, feature-weight configurations, and time windows for robustness assessment.

As summarized in Table 1, the key distinctions concern the input evidence, main goal, interpretability focus, and output level. Since clustering effectiveness depends on project characteristics, the method remains algorithm-independent, with the grouping technique as a configurable parameter for future empirical evaluation. The proposed method aligns with architecture recovery goals, while extending prior work through evolutionary signal integration and an explicit functional-technical domain interpretation.

Table 1. **Positioning of the proposed method against related approaches**

Aspect	Classical SAR	Co-change prediction	Proposed method
Input evidence	Structural dependencies	Change history signals	Structural + evolutionary + semantic temporal signals
Main goal	Package/module recovery	Top-K change recommendation	Domain structure reconstruction
Interpretation	Mainly structural grouping	Predictive accuracy	Explainable functional and technical domains
Output	Packaging decomposition	Ranked related artifacts	Responsibility-oriented domain areas

Source: **compiled by the authors**

LIMITATIONS

The recovered domains represent a living decomposition derived from shared component evolution rather than a direct reflection of formal package or directory structures. This perspective remains especially relevant under architectural degradation, where continuous change gradually erodes the original modular organization and documentation fails to preserve an up-to-date structural view. The resulting domain map, therefore, serves as an interpretive navigation artifact that highlights evolutionary relationships between responsibility areas. However, automatic clustering produces candidate domains whose semantic meaning still requires expert validation, since some clusters reflect architectural anomalies or mixed responsibilities rather than clean boundaries.

A key threat to validity arises from noise in a change history. Commits may include unrelated files due to formatting edits, reorganizations, or bulk updates, which introduce spurious co-change links and may artificially merge domains without shared responsibility. Filtering logical dependencies through strength thresholds and cleaning rules, therefore, remains necessary to reduce coincidental associations. The informativeness of recovered structures also depends on development practices: repositories with more atomic commits and fewer large merge commits tend to yield clearer evolutionary signals, while rarely modified parts of the system may remain weakly represented, and some logical dependencies may never manifest in recorded history.

Generalization across architectural patterns constitutes another limitation. In strongly layered systems, generic cohesion or coupling criteria and historical signals do not always recover engineering-oriented layering, particularly in the presence of cross-cutting infrastructural mechanisms. Functional versus technical domain separation remains challenging because technical concerns are often scattered and tangled across multiple modules, producing domains with mixed characteristics rather than purely functional or infrastructural boundaries. As a result, the distinction is not treated as a strict ground-truth label, but as an interpretation supported by explainable indicators such as infrastructural role signals, interaction breadth in the domain graph, and ownership dispersion patterns. Misclassification of cross-cutting technical domains may obscure shared dependencies and underestimate change propagation risk, which highlights the need for careful responsibility-oriented interpretation beyond structural grouping outcomes.

EVALUATION OUTLOOK

The paper introduces the method itself along with its analytical estimation, while a full empirical evaluation is presented as an outlook for future research. A baseline strategy will compare recovered domains against proxy ground truth sources, such as package or folder structures, architectural layers, and ownership information. Structural recovery quality can be assessed using MoJoFM, modularity, and MQ, which are commonly used to evaluate recovered decompositions and clustering quality in software architecture recovery.

Future evaluation includes ablation analysis to assess how co-change, structural, semantic, and temporal signals influence the reconstructed architecture, both individually and in combination. Domain stability can also be examined by varying temporal windows and filtering thresholds, since evolutionary relationships may shift across development phases.

The sufficiency of the selected historical depth will be evaluated through sensitivity analysis over different commit-based and time-based windows. In this evaluation, domain structures reconstructed from different values of N or different time intervals will be compared using structural similarity, stability of domain assignments, and consistency with proxy

architectural references and expert assessment. If increasing the historical window does not substantially change the recovered domain structure, the selected depth can be considered sufficient for effective structuring in a given project context.

The correctness of functional and technical domain classification will be evaluated using accuracy, precision, recall, F1-score, and agreement with expert labels, while robustness will be assessed through stability analysis under different time windows, cohesion thresholds, and feature-weight configurations.

The approach is compared against structural-only clustering, co-change-only clustering with filtering, and ensemble or multi-dependency recovery techniques to demonstrate the added value of the integrated measure and domain interpretation. Expert validation remains essential, as clustering-based recovery often requires human interpretation; lightweight developer assessment, therefore, is considered to confirm the semantic correctness of the produced domain map.

Subsequent research is expected to extend the method toward learning cohesion weights and clustering strategies, weakly-supervised functional/technical classification using third-party metadata, and semantic enrichment through code embeddings when historical signals are sparse.

Therefore, the research goal will be achieved if the method produces cohesive, interpretable, and stable domain structures that are consistent with proxy architectural references and expert assessment.

CONCLUSIONS

This paper introduces a method for recovering the functional and technical domain structure of an information system from change history. By treating commits as evolutionary signals, the approach constructs an integrated evolutionary cohesion graph that captures co-change, semantic, dependency-aware, and temporal relationships between artifacts. Candidate domains are identified through graph-based grouping, while an interpretive layer classifies them as functional or technical responsibility areas using explainable indicators such as ownership dispersion, reuse index, and cross-domain connectivity.

This evolutionary perspective allows domain boundaries to be inferred from actual development activity rather than relying solely on static architectural descriptions. Furthermore, such a representation helps highlight responsibility areas and their interactions, supporting more informed architectural reasoning during system evolution. In contrast to approaches that rely primarily on static structural information, the proposed method derives the structure of system domains directly from observable evolutionary behavior recorded in the repository history.

The resulting domain map serves as a living decomposition that remains meaningful under architectural degradation and can support maintenance, refactoring, and change impact analysis. Compared to approaches based solely on static dependencies or isolated evolutionary signals, the proposed method provides an interpretable and maintenance-oriented representation of system responsibility areas.

Thus, the method yields an interpretable domain structure that can potentially support architectural understanding, maintenance planning, and change impact reasoning, while full empirical confirmation of these benefits remains a part of future work. The obtained domain representation provides a higher-level abstraction of system organization and helps developers analyze responsibility boundaries and evolutionary dependencies between system components.

This study also discusses the limitations of the proposed method, while comprehensive empirical validation is deferred to future work.

In future work, the method will be extended toward learned cohesion weighting, weakly supervised functional–technical classification, and richer semantic enrichment for sparse historical signals.

REFERENCES

1. Filippone, G., Mehmood, N. Q., Autili, M., Rossi, F. & Tivoli, M. “From monolithic to microservice architecture: an automated approach based on graph clustering and combinatorial optimization”. In *IEEE 20th International Conference on Software Architecture (ICSA)*. 2023. p. 47–57. DOI: https://doi.org/10.1109/ICSA56044.2023.00013.

2. Baabad, A., Zulzalil, H. B., Hassan, S. A. & Baharom, S. B. “Software architecture degradation in open source software: A systematic literature review”. *IEEE Access*. 2020; 8: 173681–173709. DOI: https://doi.org/10.1109/ACCESS.2020.3024671.

3. Stavropoulou, I., Grigoriou, M. & Kontogiannis, K. “Case study on which relations to use for clustering-based software architecture recovery”. *Empirical Software Engineering*. 2017; 22 (4): 1717–1762. DOI: https://doi.org/10.1007/s10664-016-9459-z.

4. Sozer, H. “Evaluating the effectiveness of multi-level greedy modularity clustering for software architecture recovery”. In: *European Conference on Software Architecture*. 2019. p. 71–87. Cham: *Springer International Publishing*, https://www.scopus.com/pages/publications/85072833482. DOI: https://doi.org/10.1007/978-3-030-29983-5_5.

5. Puchala, S. P. R., Chhabra, J. K. & Rathee, A. „Ensemble clustering based approach for software architecture recovery”. *International Journal of Information Technology*. 2022; 14 (4): 2013–2019. DOI: https://doi.org/10.1007/s41870-021-00846-0.

6. Altınışık, M., Sozer, H. & Gürsun, G. “Software Architecture Recovery from Multiple Dependency Models”. In: *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*. 2024. p. 1185–1192. DOI: https://doi.org/10.1145/3605098.3635917.

7. Yang, T. H. & Huang, C. Y. “Improving software modularization quality through the use of multi-pattern modularity clustering algorithm”. In: *IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. 2023. p. 696–706. DOI: https://doi.org/10.1109/QRS60937.2023.00073.

8. Kretsou, M., Arvanitou, E. M., Ampatzoglou, A., Deligiannis, I. & Gerogiannis, V. C. “Change impact analysis: A systematic mapping study”. *Journal of Systems and Software*. 2021; 174: 110892, https://www.scopus.com/pages/publications/85098739462. DOI: https://doi.org/10.1016/j.jss.2020.110892.

9. Zhou, D., Wu, Y., Xiao, L., et al. “Understanding evolutionary coupling by fine-grained co-change relationship analysis”. In *IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 2019. p. 271–282, https://www.scopus.com/pages/publications/85072325334. DOI: https://doi.org/10.1109/ICPC.2019.00046.

10. Zhou, D., Zhang, J., Yu, P. & Guo, W. “Enhancing Change Impact Prediction by Integrating Evolutionary Coupling with Software Change Relationships”. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2024, October. p. 49–60. DOI: https://doi.org/10.1145/3674805.3686668.

11. Zagane, M., Azar, A. T. & El-Shafai, W. “Deep semantic embeddings for scalable co-change recommendation in software systems”. *IEEE Access*. 2026. DOI: https://doi.org/10.1109/ACCESS.2026.3656097.

12. Sohn, J. & Papadakis, M. “Cement: on the use of evolutionary coupling between tests and code units. a case study on fault localization”. In *IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. 2022. p. 133–144. IEEE. DOI: https://doi.org/10.1109/ISSRE55969.2022.00023.

13. Stana, A. D. & Sora, I. “Refining software clustering: The impact of code co-changes on architectural reconstruction”. *IEEE Access*. 2025. DOI: https://doi.org/10.1109/ACCESS.2025.3592777.

14. de Brito, M. C., Bianchini, C. P. & Silva, L. A. “Discovery of modularity in monolithic Java Project codes using complex networks”. *Journal of Web Engineering*. 2025; 24 (6): 911–942. DOI: https://doi.org/10.13052/jwe1540-9589.2463.

15. Meng, F., Wang, Y., Chong, C. Y., Yu, H. & Zhu, Z. “CLUE: Customizing clustering techniques using machine learning for software modularization”. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*. 2024. p. 189–198. DOI: https://doi.org/10.1145/3671016.3674816.

16. Belle, A. B., Boussaidi, G. E., Lethbridge, T. C., Kpodjedo, S., Mili, H. & Paz, A. “Systematically reviewing the layered architectural pattern principles and their use to reconstruct software architectures”. *arXiv preprint*. 2021. DOI: https://doi.org/10.48550/arXiv.2112.01644.

17. da Silva Carvalho, L. P., Mendes, T. S., de Souza Gomes, F. G., Freire, S., Novais, R. L. & Mendonça, M. G. “Using third-party components' metadata to analyze cross-cutting concerns”. *Journal of Software Engineering Research and Development*. 2024; 12 (1). DOI: https://doi.org/10.5753/jserd.2024.3086.

18. Dobrean, D. & Diosan, L. “Detecting model view controller architectural layers using clustering in mobile codebases”. In *ICSoFT*. 2020. p. 196–203. DOI: https://doi.org/10.5220/0009884601960203.

19. Wen, F., Nagy, C., Lanza, M. & Bavota, G. “Quick remedy commits and their impact on mining software repositories”. *Empirical Software Engineering*. 2022; 27 (1): 14. DOI: https://doi.org/10.1007/s10664-021-10051-z.

20. Kirbas, S., Hall, T. & Sen, A. “Evolutionary coupling measurement: Making sense of the current chaos”. *Science of Computer Programming*. 2017; 135: 4–19. DOI: https://doi.org/10.1016/j.scico.2016.10.003.

21. Kumar, N., Singh, R. & Rathee, A. “MOIF-SAR: A multi-objective information fusion-based software architecture recovery approach”. In *AIP Conference Proceedings. AIP Publishing LLC*. 2025; 3343 (1): 020020. DOI: https://doi.org/10.1063/5.0292690.

22. Tendean, S., Siahaan, D. & Yuniarti, A. “Modularization in object oriented software: A comparative study”. In *International Conference on Artificial Intelligence, Blockchain, Cloud Computing, and Data Analytics (ICoABCD)*. 2024. p. 291–296. DOI: https://doi.org/10.1109/ICoABCD63526.2024.10704544.

23. Ibrahim, K., Hassan, H., Wassif, K. T. & Makady, S. “Context-Aware Expert for Software Architecture Recovery (CAESAR): An automated approach for recovering software architectures”. *Journal of King Saud University-Computer and Information Sciences*. 2023; 35 (8): 101706. DOI: https://doi.org/10.1016/j.jksuci.2023.101706.

24. Kurinko, D. D. & Kryvda, V. I. „Uncertainty-aware multi-objective refactoring for code duplication”. *Herald of Advanced Information Technology*. 2025; 8 (3): 301–315. DOI: https://doi.org/10.15276/hait.08.2025.19.

25. Syromiatnikov, M. V. & Ruvinskaya, V. M. “A system internals modelling and annotation language for large language model-driven software engineering”. *Applied Aspects of Information Technology*. 2026; 9 (1): 103–121. DOI: https://doi.org/10.15276/aait.09.2026.08.

Conflicts of Interest: The authors declare that they have no conflict of interest regarding this study, including financial, personal, authorship, or other interests, which could influence the research and its results presented in this article

Received 14.04.2026
Received after revision 12.06.2026
Accepted 19.06.2026

DOI: https://doi.org/10.15276/hait.09.2026.23
УДК 004.415:004.4'24

Метод структурування функціональних і технічних доменів інформаційної системи на основі історії змін проєкту

Галамай Віталій Степанович[1)]
ORCID: https://orcid.org/0009-0001-9960-4263; vitalii.s.halamai@lpnu.ua
Мельник Віктор Анатолійович[2)]
ORCID: https://orcid.org/0000-0002-5046-8002; viktor.melnyk@kul.pl. Scopus Author ID: 57200786767
[1)] Національний університет «Львівська політехніка», вул. Степана Бандери, 12. Львів, 79013, Україна
[2)] Люблінський католицький університет ім. Івана Павла II, Al. Racławickie 14. Lublin, 20-950, Польща

АНОТАЦІЯ

Актуальність: Інформаційні системи постійно еволюціонують унаслідок внесення комітів і модифікацій, що поступово знижує архітектурну прозорість і розмиває межі відповідальності між компонентами. У довготривалих системах статична архітектурна документація часто стає застарілою, тоді як історія змін репозиторію зберігає об'єктивні дані про спільну еволюцію артефактів і формування еволюційних залежностей. **Мета статті**. Метою роботи є розроблення методу автоматичного структурування функціональних і технічних доменів інформаційної системи на основі історії змін проєкту та сигналів еволюційної зв'язаності, отриманих із репозиторію. **Завдання**: Основними завданнями дослідження є формалізація моделі історії змін проєкту, визначення показників еволюційної зв'язаності, побудова інтегрованого графа еволюційної зв'язаності, розроблення процедури графового групування доменів, диференціація відновлених доменів на функціональні та технічні області відповідальності, а також визначення форми представлення результатів і стратегії валідації методу. **Методи**. В дослідженні використано методи аналізу репозиторіїв, аналізу еволюційної зв'язаності, графової кластеризації, аналізу семантичної подібності, аналізу часової близькості та моделювання з урахуванням структурних залежностей. Метод поєднує частоту спільних змін, часову близькість, семантичну подібність і сигнали структурних залежностей в інтегровану міру еволюційної зв'язаності, представлену у вигляді зваженого графа. **Наукова новизна**: Наукова новизна полягає у розробленні методу відновлення доменної структури не на основі статичної архітектури чи документації, а на основі спостережуваної еволюційної поведінки, зафіксованої в історії репозиторію. Запропоновано інтегровану міру еволюційної

зв'язаності та механізм пояснюваної диференціації доменів на функціональні й технічні області відповідальності з використанням показників розпорошеності володіння, повторного використання і зв'язків між доменами. **Практична значимість**. Запропонований метод підвищує архітектурну прозорість і підтримує планування супроводу, рефакторинг, аналіз впливу змін, управління технічним боргом та аналіз зон відповідальності в еволюційних інформаційних системах. Сформована доменна карта та граф взаємодій допомагають виявляти сильно зв'язані області та аналізувати еволюційні залежності між підсистемами. **Результати**: У роботі запропоновано повний конвеєр перетворення даних еволюції репозиторію в інтерпретовану модель доменної структури. Метод формує граф еволюційної зв'язаності, визначає кандидатні домени за допомогою графового групування та класифікує їх на функціональні й технічні області на основі пояснюваних індикаторів. Наведений ілюстративний приклад демонструє, що стабільні патерни спільної еволюції дозволяють формувати інтерпретовані доменні області навіть за відсутності явної архітектурної документації. **Висновки**. Запропонований метод забезпечує відновлення інтерпретованої структури функціональних і технічних доменів на основі історії змін проєкту шляхом поєднання сигналів спільних змін, семантичної подібності, структурних залежностей і часової близькості. Отримана доменна декомпозиція підтримує архітектурне розуміння та аналіз супроводу системи, зменшуючи залежність від неповної або застарілої документації. Подальші дослідження будуть спрямовані на емпіричну валідацію та вдосконалення механізмів класифікації й налаштування вагових коефіцієнтів.

Ключові слова: еволюційна зв’язаність; виявлення спільнот; аналіз репозиторіїв; граф зв’язності; декомпозиція доменів; функціональні домени; технічні домени

ABOUT THE AUTHORS



Vitalii S. Halamai - PhD student, Department of Electronic Computing Machines. Lviv Polytechnic National University, 12, Stepana Bandery Street. Lviv, 79013, Ukraine
ORCID: https://orcid.org/0009-0001-9960-4263; vitalii.s.halamai@lpnu.ua
Research fields: Computer engineering, code analysis

Галамай Віталій Степанович - аспірант кафедри Електронних обчислювальних машин. Національний університет «Львівська політехніка», вулиця Степана Бандери, 12. Львів, 79013, Україна



Viktor A. Melnyk - Doctor of Engineering Sciences, Professor, Department of Social and Technical Sciences. John Paul II Catholic University of Lublin, Al. Racławickie 14, Lublin, 20-950, Poland
ORCID: https://orcid.org/0000-0002-5046-8002; viktor.melnyk@kul.pl. Scopus Author ID: 57200786767
***Research fields*:** Computer systems architecture research and design; IP cores design; high-performance reconfigurable computer systems design; computer data protection; cryptographic processors design and implementation; wireless sensor network security

Мельник Віктор Анатолійович - доктор технічних наук, професор факультету Соціальних і технічних наук. Люблінський католицький університет ім. Івана Павла II, Al. Racławickie, 14, Lublin, 20-950, Польща