axioms

MDPI

Article

Method for Automated Decomposition of Monolithic Software Systems Based on Graph Neural Networks

Yaroslav Kornaga [1], Oleksandr Hubariev [1,*], Serhii Yevseiev [2], Petro Yablonskyi [3] and Tetiana Pyrohovska [4]

1 Faculty of Informatics and Computer Engineering, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 03056 Kyiv, Ukraine; slovyan_k@ukr.net
2 Department of Cybersecurity, National Technical University "Kharkiv Polytechnic Institute", 61002 Kharkiv, Ukraine; serhii.yevseiev@gmail.com
3 Department of Descriptive Geometry, Engineering and Computer Graphics, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 03056 Kyiv, Ukraine; ypn@ukr.net
4 Department of Software Engineering for Power Industry, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 03056 Kyiv, Ukraine; t.pyrohovska@lll.kpi.ua
* Correspondence: gubarev.alexandr@gmail.com

Abstract

The purpose of this study is to improve the architectural quality of software systems with microservice architecture by investigating and improving methods for automated transition from monolithic architecture to microservice architecture. The study was performed using methods of static source code analysis, graph theory, algorithms for transforming graph structures (detecting strongly connected components, redundant and cyclic dependencies), cluster analysis algorithms, graph neural networks, as well as multi-criteria assessment of internal cluster consistency and inter-cluster connectivity. The results presented in the study are a comprehensive solution to the scientific problem of ensuring the automated transformation of monolithic software systems into a microservice architecture with controlled inter-service connectivity and a high level of internal consistency of architectural components.

Keywords: monolithic architecture; microservice architecture; software system decomposition; software architecture; static analysis; graph-based methods; machine learning; neural networks; distributed systems; information systems

MSC: 68T07; 05C90; 68R10; 68T05

Check for updates

Academic Editor: Arash Rafiey

Received: 9 May 2026
Revised: 10 June 2026
Accepted: 30 June 2026
Published: 2 July 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

In recent decades, software developers have increasingly abandoned monolithic systems and switched to microservices. Microservice architecture has better scalability, greater flexibility and ease of maintenance [1]. However, switching from monoliths to microservices is not an easy task as monoliths have hidden dependencies, modules are closely integrated with each other, domain boundaries are virtually absent, and the manual decomposition process of a monolithic system can be lengthy and lead to errors [2]. Therefore, the research and development of automated solutions for the transition from monolithic architecture to microservices are relevant today.

The problem of decomposition has already been studied within various scientific directions [3,4]. Heuristic and metric-oriented approaches [5–7] used dependency matrices, connectivity metrics, and rule sets to divide systems into services, which allowed

us to achieve a sufficient level of internal consistency of components; however, the resulting clusters only partially corresponded to subject domain boundaries. Later, graph approaches such as Service Cutter, IBM Mono2Micro, DacomiCo, and GDC-DVF began to be used, which used dependency graphs and increased modularity to improve the structural partition of the system.

On the basis of the results of previous studies, more modern approaches have been further developed, which use machine learning methods and aim at automatically detecting syntactic and semantic relationships between components of software systems [8]. Desai et al. in 2021 [9] proposed a graph neural network (GNN) to reduce the impact of abnormal components during refactoring of monolithic applications. Qian et al. in 2023 [10] proposed a deep clustering approach integrating two representations of the system architecture (dual-view fusion) to extract microservices. Yedida et al. in 2023 [11] used deep learning techniques to distribute cloud microservice applications. Sellami et al. in 2022 [12] used the hierarchical DBSCAN clustering algorithm for structural refactoring. Ziabakhsh et al. in 2025 [13] proposed a Mo2oM model—soft clustering (components may belong in part to multiple microservices) based on a graph neural network using deep semantic vector representations.

As a result of using these approaches, it was possible to significantly increase the modularity and semantic coherence of clusters. However, they exhibit sensitivity to data configuration, complexities when combining structural and semantic features, limited efficiency when eliminating redundant intermodular relationships, and difficulties in generalizing training models for heterogeneous software domains [14,15].

The proposed methodology contains several methodological contributions in the field of automated decomposition of monolithic systems into microservices. It provides an adaptive mechanism for the automatic selection of a clustering algorithm based on the structural characteristics of the dependency graph. A mechanism for automatic formation of training labels based on the results of heuristic graph analysis is proposed, which allows the training of a graph neural network without using expertly labeled decompositions or pre-prepared reference data sets. The heuristic procedure is used exclusively at the stage of training data preparation, while after training is completed, the decomposition is performed directly by the graph neural network without repeated application of the heuristic procedure. The proposed approach combines the structural characteristics of the graph, semantic information, and graph-neural representations within a single decomposition process. The model is optimized using a multi-criteria objective function that simultaneously takes into account classification error, cohesion, coupling, and semantic consistency. Together, these components form an adaptive end-to-end framework for automated software decomposition.

2. Materials and Methods

2.1. The Method of Structural Transformation of the Dependency Graph of the Software System

The study proposes a comprehensive approach to the structural analysis of software architecture, which is based on the step-by-step processing of the graph model of the system. At the first stage, the software architecture is analyzed, consisting of several consecutive steps:

1. Construction of an abstract syntax tree (AST).
2. Formation of a directed graph of dependencies between system components.
3. Calculation of the centrality of nodes (to determine structurally important modules of the system).
4. Identification of bridges and points of articulation that ensure the integrity of the architectural model.

5. Determination of weighting coefficients for edges and nodes.
6. Graph structure transformation.

Calculation of the centrality of nodes.

Degree Centrality:

$$DC(v) = DC_{\text{in}}(v) + DC_{\text{out}}(v), \tag{1}$$

$$DC_{\text{in}}(v) = \frac{\deg_{\text{in}}(v)}{n-1}, \tag{2}$$

$$DC_{\text{out}}(v) = \frac{\deg_{\text{out}}(v)}{n-1}, \tag{3}$$

where:

- $\deg_{\text{in}}(v)$ is the number of incoming connections of node v;
- $\deg_{\text{out}}(v)$ is the number of outgoing connections of node v;
- $n-1$ is the total number of nodes in the graph excluding node v itself.

Betweenness Centrality:

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{4}$$

where:

- σ_{st} is the total number of shortest paths between nodes s and t in the graph;
- $\sigma_{st}(v)$ is the number of such shortest paths passing through node v.

Identification of bridges and points of articulation that ensure the integrity of the architectural model.

First, the critical nodes of the dependency graph of the software system are determined. For this, a threshold analysis is used, according to which node v is considered critical if the following condition holds:

$$DC(v) > \mu_{DC} + \sigma_{DC}, \tag{5}$$

or

$$BC(v) > \mu_{BC} + \sigma_{BC}, \tag{6}$$

where:

- μ_{DC} and σ_{DC} are the mean value and standard deviation of Degree Centrality;
- μ_{BC} and σ_{BC} are the mean value and standard deviation of Betweenness Centrality.

The critical threshold value is determined dynamically for each analyzed project based on the statistical distribution of centrality values.

After identifying critical nodes, a depth-first search (DFS) algorithm is applied, which allows us to identify structurally important elements of the dependency graph by analyzing their impact on the overall connectivity of the system. Since the detection of bridges and articulation points is defined for undirected graphs, this analysis is performed on the undirected representation of the dependency graph.

During the DFS traversal, two parameters are calculated for each vertex v:

- $dfs[v]$—the discovery time of node v;
- $low[v]$—the minimum discovery time of any vertex reachable from v through DFS tree edges and back edges, including v itself.

Calculating these values is a key step in DFS analysis based on low-link values, which is used to detect bridges and articulation points. The value of $low[v]$ is determined as follows:

$$low[v] = \min\left(dfs[v], \min_{w \in children(v)} low[w], \min_{u \in back\text{-}edges(v)} dfs[u]\right), \tag{7}$$

where:

- $dfs[v]$ is the discovery time of node v during DFS;
- $low[w]$ is the low value of a child node w of vertex v in the DFS tree;
- $dfs[u]$ is the discovery time of a node u connected to v by a back edge.

The edge $e(u, v)$ is considered a bridge in the dependency graph if the following condition holds for the descendant node v:

$$low[v] > dfs[u], \tag{8}$$

This means that vertex v and its subtree cannot be reached from the rest of the graph without the edge $e(u, v)$; therefore, removing this edge disconnects the graph into separate components.

Vertex v is defined as an articulation point in the following cases:

1. If vertex v is the root of a DFS tree and has more than one child.
2. If the following condition holds for any child:

$$low[child] \geq dfs[v]. \tag{9}$$

This suggests that after removing vertex v, its child will lose its connection to other parts of the graph.

Bridge edges are denoted as critical connections between autonomous subgraphs, the removal of which leads to the rupture of the graph. Bridge edges in the dependency graph are not removed because they are critical to preserving system connectivity, but are seen as natural boundaries between future microservices because they connect autonomous subgraphs through a minimal number of interactions.

Articulation points are nodes through which communication between several subsystems is carried out, and such nodes should be marked as central routing elements or isolated into separate clusters to prevent excessive connectivity.

Determination of weighting coefficients for edges and nodes.

Formula for calculating the weight of an edge:

$$W(e_{ij}) = w_k, \tag{10}$$

where:

- w_k—weight of the corresponding type of edge.

Table 1 presents the edge types in the dependency graph, their assigned weight values, and descriptions of the corresponding relationships between software components.

The weights given in Table 1 reflect the relative strength of different types of architectural dependencies. Inheritance relationships receive the highest weight because they create strong structural coupling between components. The use of classes represents a direct implementation dependency and therefore also receives a relatively high weight. Meanwhile, function calls, use of interfaces, and interaction through a shared database receive lower weights because they typically reflect weaker or more indirect forms of coupling. The weights were chosen heuristically based on generally accepted principles of software architecture and should be considered as empirical indicators of the strength of dependencies rather than mathematically derived constants.

Table 1. Weights for different edge types.

Edge Type	Weight	Description
Function call	0.5	The main mechanism of interaction between modules.
Class usage	0.7	One module contains a class of another module.
Class inheritance	1.0	A class inherits from another class, forming a strong coupling.
Interface usage	0.4	Interaction between components through API.
Shared database usage	0.4	Modules interact through the database, but not directly.

The node weight $W(v)$ allows for a quantitative assessment of the architectural significance of a component, taking into account:

1. Its incoming and outgoing connections;
2. Its role in data flow between other modules;
3. Its frequency of use as a dependency in other modules.

Formula for calculating the node weight:

$$W(v) = DC_{\text{in}}(v) + DC_{\text{out}}(v) + BC(v), \tag{11}$$

To illustrate the graph-theoretic concepts used in the proposed approach, Figure 1 shows a simplified dependency graph of a hypothetical software system. The example demonstrates typical architectural relationships between controllers, services, repositories, entities, and external clients. Such a graph is used to explain centrality metrics, bridges, and articulation points and their role in the process of structural analysis of software architecture.

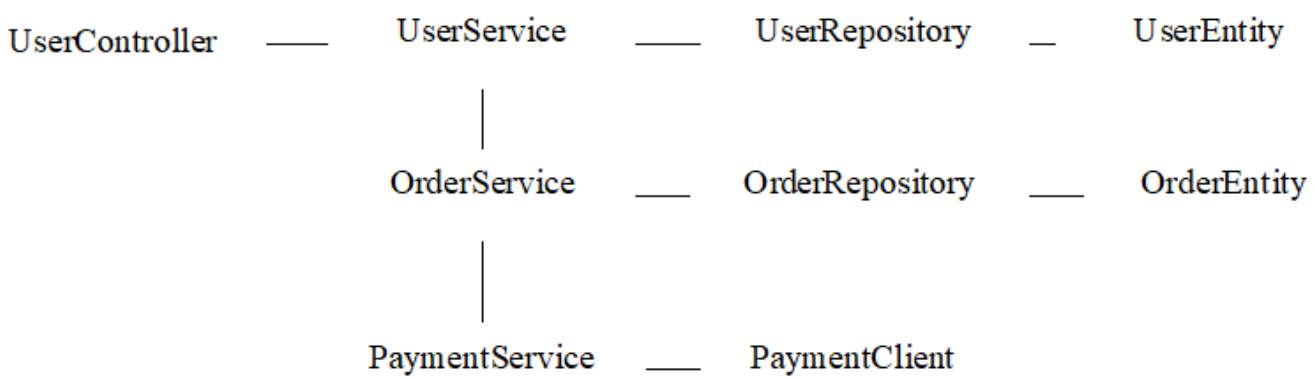

Figure 1. Simplified example of a software dependency graph used to illustrate graph-theoretic metrics and structural analysis concepts.

In the given example, the UserService node is a structurally important element as it connects several functional subsystems, which leads to a high Degree Centrality value. The UserService and OrdersService nodes also belong to a significant number of shortest paths between other components of the graph, which leads to increased values of Betweenness Centrality. The connection between UserService and OrdersService can be considered a bridge since its removal leads to a disruption of the connectivity of part of the graph. Similarly, the UserService node can act as an articulation point, since its removal increases the number of connected components of the graph. Such structural characteristics are used to identify critical elements of the architecture before performing a decomposition of the software system.

To illustrate the impact of dependency weights on the graph structure, different types of relationships should be assigned different weights according to Table 1. The re-

lationship UserRepository → UserEntity has a weight of w = 0.7 (class usage), the relationships OrderService → PaymentService and UserService → OrderService have a weight of w = 0.5 (function call). This weighting allows us to take into account the different architectural significance of the dependencies when constructing and analyzing the graph.

Graph structure transformation.

Using the results of the analysis carried out in the previous stages, a process of structural transformation of the graph is implemented, which includes detection and removal of weak and redundant connections, removal of cyclic dependencies, and combining nodes into more connected and structurally stable components.

In the first step, a search is made for subgraphs with high internal connectivity (strongly connected components (SCC)). Detection of SCC allows us to identify cyclic dependencies between modules, to see high cohesion groups that can be considered as single logical blocks, and to detect structural antipatterns, for example, when several modules constantly interact with each other (call methods, transmit data), forming a closely related set of dependencies.

To identify strongly connected components (SCCs) representing cyclic dependencies in the graph, Tarjan's Algorithm is used:

$$SCC(G) = \bigcup_{i=1}^{k} G_i, \quad G_i = (V_i, E_i), \tag{12}$$

where:

- $SCC(G)$ is the set of all strongly connected components in the graph, where for every pair of vertices $u, v \in V_i$, there exists a directed path from u to v and a directed path from v to u;
- $G_i = (V_i, E_i)$ is a separate subgraph (one SCC);
- k is the number of SCCs found.

After identifying strongly connected components (SCC), it is necessary to determine their density. The density of a component, D_{SCC}, indicates how tightly the nodes within the subgraph are interconnected.

The density is calculated using the formula:

$$D_{SCC} = \frac{|E_{SCC}|}{|V_{SCC}| \cdot (|V_{SCC}| - 1)}, \tag{13}$$

where:

- $|E_{SCC}|$ is the number of edges within the SCC;
- $|V_{SCC}|$ is the number of nodes within the SCC.

To identify strongly connected components with increased density, it is necessary to compute a global density threshold, $D_{\text{threshold}}$.

Threshold determination:

$$D_{\text{threshold}} = \bar{D} + \sigma_D, \tag{14}$$

SCC with $D_{SCC} < D_{\text{threshold}}$ are weakly connected (any cycle present is likely non-critical).

SCC with $D_{SCC} \geq D_{\text{threshold}}$ are dense and considered as potential candidates for transformation.

To calculate the set of all simple cycles in a directed graph without duplications, this study uses Johnson's algorithm, which is based on depth-first search (DFS) and is optimized by reducing the search space.

Once all cycles have been detected, the least important connections must be removed (considering the importance of both the connections themselves and the nodes they connect) for the graph to become acyclic.

For this, a combined metric is proposed that takes into account the weight of the connection, the weight of the target node, and the centrality of the target node (structural role in the graph).

Formula for selecting an edge to break a cycle:

$$e^* = \min_{e_{ij} \in C} \left(\frac{W(e_{ij})}{W(v_j) + BC(v_j) + \varepsilon} \right), \tag{15}$$

where:

- e_{ij}—a directed edge from node v_i to vertex v_j;
- C—the set of edges forming a specific cycle in the graph;
- $W(e_{ij})$—edge weight;
- $W(v_j)$—node weight of v_j;
- $BC(v_j)$—the centrality of node v_j in the graph structure;
- $\varepsilon \ll 1$—a constant to avoid division by zero.

In the next step, it is necessary to evaluate which of the strongly connected components (SCCs) should be combined into a single logical node, which will simplify the structure of the graph.

For each SCC, it is necessary to compute the weighted average strength of the connections between its nodes, taking into account the importance (weighting coefficients) of the nodes themselves.

To this end, we compute the average weight of internal edges within the SCC:

$$W_{SCC} = \frac{1}{|E_{SCC}|} \sum_{e_{ij} \in E_{SCC}} W(e_{ij}) \left(\frac{W(v_i) + W(v_j)}{2} \right), \tag{16}$$

where:

- E_{SCC}—the set of edges in the SCC;
- e_{ij}—an edge between nodes v_i and v_j in the SCC;
- $W(e_{ij})$—the weight of the edge between vertices v_i and v_j;
- $W(v_i)$ and $W(v_j)$—node weighting coefficients representing the importance or influence of each node;
- $\frac{W(v_i)+W(v_j)}{2}$—the average weight of the nodes connected by edge e_{ij}.

To identify SCCs with significantly higher internal connectivity compared to others, which can be merged into a single node, an edge weight threshold is computed:

$$T_W = \mu_W + \sigma_W, \tag{17}$$

where:

- T_W—edge weight threshold.
- μ_W—the mean value of W_{SCC} calculated across all detected strongly connected components;
- σ_W—the standard deviation of W_{SCC} values calculated across all detected strongly connected components.

SCCs in which the average weight of the internal edges $\bar{W}_{SCC_k}$ exceeds the threshold T_W are considered as candidates to join into a node because they exhibit high internal cohesion.

Additionally, to reduce excessive coupling between components of the software system, it is necessary to identify and eliminate redundant edges that duplicate information flow paths already established indirectly through other components. Specifically, if a direct connection W_{direct} exists between two modules, and an indirect connection W_{indirect} is also present, the redundant direct connection may be removed. Indirect paths are defined as simple directed paths from u to v whose length is greater than one edge and does not exceed four edges. The total indirect weight is calculated as the sum of the edge weights for all such paths.

Formula for checking a redundant edge:

$$W_{\text{direct}} < \sum W_{\text{indirect}}, \tag{18}$$

If this condition holds, the direct edge is removed.

Resulting graph after transformation:

$$G' = (V', E'), \tag{19}$$

where $G' = (V', E')$ is a subset of nodes and edges after structural transformation.

$$E' = E - \{e_{ij} \mid W(e_{ij}) < T\}, \tag{20}$$

where only the most important edges are preserved.

As a result of performing a given sequence of operations (removing cycles, merging strongly connected components, and eliminating redundant connections), a dependency graph $G' = (V', E')$ is formed, which preserves only structurally important connections between components.

Such a graph is characterized by a lower density and a better balanced topology, in which there are no information duplications, better visibility of modular boundaries and which is the basis for further high-precision clustering.

2.2. Dynamic Selection Method of the Clustering Algorithm

In the second stage, the graph is subject to further clustering. Clustering makes it possible to automatically isolate logically coupled subsystems which, given technological and architectural constraints, can be converted into independent microservices.

A directed dependency graph of a software system serves as the initial model for solving the clustering problem.

Let $C = \{C_1, C_2, \ldots, C_k\}$ be a partition of the set V into k disjoint subsets (clusters), such that:

$$\bigcup_{i=1}^{k} C_i = V, \qquad C_i \cap C_j = \varnothing, \quad \forall i \neq j, \tag{21}$$

Clustering is considered successful if:

(1) the intra-cluster connectivity (cohesion) within each C_i is maximized;

Cohesion measures how strongly the components within a service are related to each other, how many internal connections exist between them and how consistently they contribute to a common function.

For each cluster C, cohesion is defined as:

$$Cohesion(C) = \frac{\sum_{u,v \in C,\, u \neq v} W(u,v)}{|C|(|C|-1)}, \tag{22}$$

where:

- $W(u,v)$—the weight of the edge between classes u and v;
- $|C|$—the number of elements in the cluster.

(2) the inter-cluster connectivity (coupling) between C_i and other C_j is minimized.

Coupling measures the number of external connections a service has and the extent to which it depends on other clusters.

$$Coupling(C) = \sum_{\substack{u \in C \\ v \in V \setminus C}} W(u,v), \tag{23}$$

where:

- $W(u,v)$—the weight of the external edge between classes u and v.

Thus, coupling is defined as the total weight of all outgoing dependencies from a cluster to other clusters.

This is formulated as a multi-criteria problem:

$$\max_{C} \left[\sum_{i=1}^{k} (Cohesion(C_i) - Coupling(C_i)) \right], \tag{24}$$

Thus, the graph G partitioning is carried out into such clusters C_i, which are characterized by the highest value of the clustering quality metric.

Semantic consistency is used to estimate how much components within the same service belong to a common subject area and perform close functional roles, and unlike structural metrics (cohesion and coupling), which analyze only the dependency graph, semantic homogeneity allows checking the correspondence of clusters to logical domain boundaries. The structure of the graph can vary significantly for different software systems (density, size, the direction of edges in the graph, availability of shared resources, etc.), so the use of a single clustering algorithm for all cases is limited. Instead, a dynamic approach to choosing a clustering algorithm is proposed, which is based on local characteristics of the graph: number of nodes $|V_s|$, density $D(S)$, local FanOut, Shared Resource Ratio (SRR), and betweenness centrality of nodes $BC(v)$.

Table 2 presents the dynamic clustering algorithm selection strategy based on graph metrics, including the corresponding conditions and the recommended clustering method for each graph structure scenario.

The threshold values given in Table 2 were chosen empirically during research experiments and should be interpreted as practical decision-making criteria rather than as theoretically optimal limits.

The absolute values of the decomposition quality metrics depend not only on the chosen clustering algorithm but also on the complexity, density, structure of relationships, and existing architectural dependencies of the software system under study. Therefore, the proposed mechanism for dynamic algorithm selection is aimed at determining the most suitable clustering strategy for a specific structure of the dependency graph. Its goal is to adapt the decomposition process to the specifics of the analyzed system rather than providing predetermined values of the quality metrics.

Table 2. Dynamic clustering algorithm selection strategy.

Conditions Based on Graph Metrics	Recommended Clustering Algorithm
$\begin{cases} 0.3 < D(G) < 0.5 \\ 0.2 \leq SRR \leq 0.5 \\ \lvert V \rvert > 300 \\ \overline{BC}_{norm} > 0.15 \\ 2 \leq FanOut_{avg} \leq 5 \end{cases}$	Spectral Clustering
$\begin{cases} \overline{BC}_{norm} > 0.15 \\ D(G) > 0.3 \\ \lvert V \rvert > 300 \\ FanOut_{avg} > 5 \\ SRR > 0.5 \end{cases}$	Infomap
$\begin{cases} 0.05 < \overline{BC}_{norm} \leq 0.15 \\ 0.1 \leq D(G) \leq 0.3 \\ 100 < \lvert V \rvert < 300 \\ 2 \leq FanOut_{avg} \leq 5 \\ 0.2 \leq SRR \leq 0.5 \end{cases}$	Louvain (all other cases)
$\begin{cases} \overline{BC}_{norm} < 0.05 \\ D(G) < 0.1 \\ \lvert V \rvert > 300 \\ FanOut_{avg} < 2 \\ SRR < 0.2 \end{cases}$	Label Propagation
$\begin{cases} \lvert V \rvert < 100 \\ D(G) < 0.1 \\ \overline{BC}_{norm} > 0.15 \\ 2 \leq FanOut_{avg} \leq 5 \\ 0.2 \leq SRR \leq 0.5 \end{cases}$	Girvan–Newman
$\begin{cases} 100 < \lvert V \rvert < 300 \\ 0.2 \leq SRR \leq 0.5 \\ 0.1 \leq D(G) \leq 0.3 \\ 0.05 < \overline{BC}_{norm} < 0.15 \\ 2 \leq FanOut_{avg} \leq 5 \end{cases}$	Markov Clustering (MCL)

2.3. Cluster Structure Refinement Method

To improve the architectural structure and strengthen modular insulation, it was proposed to apply a model that will take into account the structural and semantic indicators of the quality of cluster partitioning.

Thus, the problem reduces to finding a partition $C_1, C_2, \ldots, C_K$, that maximizes the overall objective function:

$$\max_{(C_1,\ldots,C_K)} \left[\frac{1}{K} \sum_{i=1}^{K} Cohesion(C_i) - \frac{1}{|E|} \sum_{i \neq j} Coupling(C_i, C_j) \right], \quad \text{subject to: } \forall i,\ Sem(C_i) \geq \delta, \tag{25}$$

where:

- δ—fixed threshold.

To implement the model, iterative search is used to gradually improve the initial partitioning of system components, where at each iteration one of the nodes of the graph is randomly selected and the feasibility of moving it to one of the neighboring clusters is checked. The proposed strategy limits the search space within the structurally close environment, which ensures better convergence of the algorithm due to minimal interference with the general structure.

To prevent the degradation of architectural integrity, the model sets a limit—the node cannot be moved to a new cluster if the semantic consistency of the target cluster then becomes lower than the given threshold $\delta = 0.5$.

The quality assessment of each potential change is carried out on the basis of a weighted metric, which is formally given as:

$$temp_score = 2.0 \cdot cohesion - 0.1 \cdot coupling \tag{26}$$

In a given function, a factor of 2.0 is applied for cohesion to form tightly coupled subsystems, whereas the coefficient of 0.1 for coupling allows us to maintain control over modular insulation and avoid excessive fragmentation of clusters.

The coefficients were chosen empirically to enhance the effect of cohesion improvement during local node movement while preventing coupling-related changes from dominating the temporal estimate. They should be interpreted as heuristic calibration parameters rather than theoretically derived constants.

2.4. Graph Neural Network-Based Decomposition Model

In the next stage of the research, a graph neural network model was employed to automate the software system decomposition process.

A system, represented as a graph of dependencies between its components, shows a network structure (nodes as classes or modules, edges as calls, dependencies or interactions between them) and requires models that can process not only feature vectors but also take into account the context of the node's placement in the graph structure (topological environment and types of connections with other nodes).

Therefore, a modification of the Graph Convolutional Network (SimpleGCN) was chosen for modeling, which works directly on the graph structure and aggregates information from the local context of each node through a normalized adjacency matrix. This model allows learning internal representations of components, taking into account their own properties and the structural environment in which they function.

The neural model is presented in Figure 2.

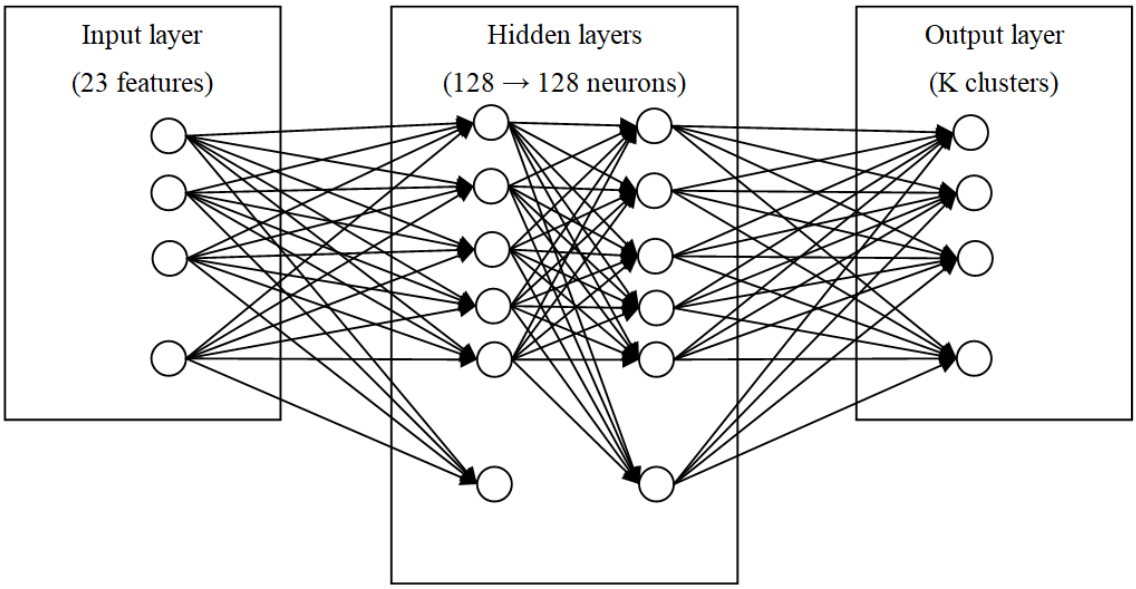

Figure 2. Neural network architecture.

A two-layer architecture was chosen because it provides a balance between depth of analysis and maintaining feature clarity. The first layer of the GCN processes the closest

connections between classes that form the local structure, while the second layer takes into account interactions at a distance of two steps (connections between neighboring subsystems, where domain logic manifests itself). This is enough for the network to identify important architectural context, but at the same time, it does not average features, which would lead to a blurring of distinctions between classes.

The width of both layers is set to 128 hidden components. This is sufficient to reproduce medium- and large-scale architectural structures, while limiting the risk of overfitting.

For each node of the graph, a feature vector was constructed, which included structural and semantic characteristics. The following structural features were added:

1. In-degree centrality of the node;
2. Out-degree centrality of the node;
3. Betweenness centrality of the node;
4. Local clustering coefficient:

$$C(v) = \frac{2e_v}{k_v(k_v - 1)}, \tag{27}$$

where:

- e_v—the number of edges between the neighbors of node v;
- k_v—the number of neighbors.

5. PageRank coefficient:

$$PR(v) = \frac{1-d}{|V|} + d \sum_{u \in N(v)} \frac{PR(u)}{\deg(u)}, \tag{28}$$

where:

- d—damping factor;
- $|V|$—the number of nodes in the dependency graph;
- $N(v)$—the set of incoming neighbors of vertex v, i.e., vertices from which edges are directed to v;
- $\deg(u)$—the number of edges between node u and other components of the dependency graph.

6. Node weight;
7. A binary feature indicating resource usage, which captures whether a component participates in data access operations and is important for modeling data sharing across clusters:

$$is_resource_v = \begin{cases} 1, & \text{if component } v \text{ represents a common data layer element} \\ & \text{used by multiple domain components,} \\ 0, & \text{otherwise.} \end{cases} \tag{29}$$

When propagating information through a graph, the neural network takes into account the node type and the weight of the connection and forms a representation of each component that takes into account its structural and data context. If the components of the system are connected by the same resource, their vector representations converge, which indicates a potential dependency through a common data model. Due to the lower weight, resource connections do not prevail over business logic connections, and the neural network forms service boundaries that take into account both programmatic interaction and connections to the database.

After receiving the services, an analysis of the distribution of resources between them is performed; for each resource, the service that has the closest connection to the resource is determined by calculating the total weight of the connections between the component

and the corresponding resource. Such a service is considered the owner of the resource; however, if the resource is used equally by several services, then such a case is recorded as a conflict and requires additional analysis during migration.

Semantic features were also included in the vector representation of nodes. Class names were converted to lowercase and compared to a predefined dictionary of domain tags using a case-insensitive substring search. If a matching tag was found in the class name, the node was assigned a binary semantic vector that reflected its semantic category. In the absence of a match, a null vector was used.

The semantic representation is given as a vector:

$$s(v) = (s_1, s_2, \ldots, s_k), \quad s_i \in \{0, 1\}, \tag{30}$$

where k corresponds to the number of defined semantic categories.

The loss function was enriched with three structural and semantic components. The first component is aimed at increasing the density of connections between nodes (cohesion) within each cluster. The second component is aimed at reducing inter-cluster connections (average FanOut). The third component is aimed at increasing semantic consistency (estimated as the average cosine deviation between vector representations of nodes of the same cluster).

Multi-criteria loss function:

$$L = CE + FanOut_{avg} - Cohesion - SemanticConsistency, \tag{31}$$

where:

- CE—cross-entropy, which measures the difference between the predicted and target probability distributions of clusters;
- $FanOut_{avg}$—average number of output dependencies of cluster nodes;
- $Cohesion$—average number of internal connections between nodes of the same cluster;
- $SemanticConsistency$—determined based on the vector similarity of class names, tags, and attributes, i.e., the extent to which objects in the same cluster have a similar subject entity.

Cohesion and *SemanticConsistency* are presented as normalized values in the range [0,1]. $FanOut_{avg}$ is logarithmically scaled before aggregation to reduce the impact of extreme values.

3. Results

3.1. Experimental Setup

In order to test the proposed methods of structural analysis of software systems, a test environment and an experimental stand were built. The Google Colaboratory cloud environment (Google Colab) was used for the experiments, which provides an integrated Python 3.12.13 environment with pre-installed libraries for data processing and the ability to restart experiments.

To verify the proposed approach and methods of automated transformation of software architecture, experimental studies were conducted using graph models obtained from the source code of an industrial logistics process management system implemented in Java using the Spring framework. The system is a large-scale monolithic corporate application containing several interconnected business modules. Due to confidentiality and intellectual property protection restrictions, the source code cannot be made public. The experiments were aimed at assessing the quality of the resulting cluster partition and checking the feasibility of dynamically selecting a clustering algorithm that depends on the structural characteristics of the dependency graph.

The experimental evaluation combined a detailed study of an industrial software system with a comparative evaluation on a set of test software systems. Accordingly, the presented metrics should be interpreted as indicators of the quality of decomposition for the studied software systems and not as statistical estimates obtained on the basis of multiple experimental tests.

3.2. Experimental Evaluation of Dynamic Clustering Algorithm Selection

To evaluate the method, graphs were constructed for 30 projects of varying scale. On the same graphs, all selected clustering algorithms were run in turn. Coupling and cohesion were calculated for each result. The algorithm that gives the best balance of these indicators is considered the most suitable for this project.

In parallel with this, the method of dynamic selection of the clustering algorithm was used, and it was checked whether this selection coincides with the most suitable algorithm that was found by exhaustive search. Next, it was calculated how often the method accurately chooses the most suitable algorithm, as well as how often the correct algorithm falls into the top two options. Separately, it was analyzed how much the indicators of coupling and cohesion deteriorate or improve if the choice was not the best.

Table 3 presents the results of the experimental evaluation of the dynamic clustering algorithm selection method for different graph size groups, including Top-1 and Top-2 accuracy, number of mistakes, and average deviations of cohesion and coupling values.

Table 3. Results of experimental evaluation of the method of dynamic selection of the clustering algorithm.

Group (Number of Nodes)	N	Top-1 (pcs)	Top-1 (%)	Top-2 (pcs)	Top-2 (%)	Mistakes (pcs)	Average Deviation of the Cohesion Value	Average Deviation of the Coupling Value
All	30	22	73.3	26	86.7	8	−0.004	+0.14
$n < 100$	10	8	80.0	8	80.0	2	−0.003	+0.08
$100 < n < 300$	10	7	70.0	9	90.0	3	−0.004	+0.14
$n > 300$	10	7	70.0	9	90.0	3	−0.004	+0.18

According to the results of an experimental study, the dynamic selection method provided an exact match with the result of a complete search in 73.3% of cases, while in 86.7% of cases the correct algorithm was among the two best options. The mean deviation of cohesion at the non-best choice is negligible (about −0.004), while the mean increase in intercluster connectivity is about 0.14, which confirms the stability of the method regardless of the size of the graph.

3.3. Experimental Evaluation of the Graph-Based Decomposition Approach

In the first step, a syntactic and semantic analysis of the source software code is performed to construct a graph of dependencies between classes. 518 files were processed (excluding test files) and a directed system graph was formed:

1. Number of nodes (classes/modules)—345;
2. Number of oriented edges (dependencies)—641.

After applying the proposed graph structural transformation method, the number of edges decreased to 452, that is, 189 edges were removed from the graph (about 29% of the original number of edges).

By removing the edges, the graph has become less cyclic and the noise level has decreased, which should improve the quality of clustering.

At the next stage, the structural characteristics of the directed graph were calculated:

- Global graph density—0.0038;
- $FanOut_{avg}$—1.310;
- Shared Resource Ratio (SRR)—0.0638;
- Normalized betweenness centrality $(\overline{BC}_{norm})$—0.000726.

Based on the obtained metrics, the graph can be classified into the following set of conditions:

$$\overline{BC}_{norm} < 0.05 \wedge D(G) < 0.1 \wedge |V| > 300 \wedge FanOut_{avg} < 2 \wedge SRR < 0.2 \quad (32)$$

According to the classification scheme for selecting clustering algorithms, Label Propagation is the most suitable method for this type of graph.

After applying this clustering algorithm, 32 clusters were formed and cohesion, coupling and semantic consistency were calculated to assess the quality of the decomposition.

The average level of cohesion in clusters is only 0.0682. This indicates that clusters have weak internal connectivity of components, that is, the number of strong structural or functional dependencies between classes grouped into one module is low.

Coupling is 119.0, meaning that the system has many external connections between clusters, making it difficult for modules to develop independently, and it is a source of violations of the principle of loose coupling.

The semantic consistency of clusters is 0.4883, that is, clusters partially correspond to the functional content of software components, but there are also a significant number of modules that belong to different domains.

3.4. Evaluation of Cluster Structure Refinement Method

The same industrial software project that was used in the study of the heuristic graph approach was used to evaluate the cluster structure improvement method. After applying the improvement method, the average cohesion value increased from 0.0682 to 0.0752, the total number of intercluster dependencies decreased from 119.00 to 88.20, and the semantic coherence value of the clusters increased from 0.4883 to 0.6358.

The results obtained confirm the feasibility of applying the cluster structure improvement stage after the initial graph decomposition. The results generated using the heuristic graph approach and the cluster improvement method are further used to form the reference decomposition and cluster labels necessary for training the graph neural network.

3.5. Evaluation of the Graph Neural Network-Based Decomposition Model

To train the graph neural network, open-source Java projects were used as a training set: Spring PetClinic, Broadleaf Commerce, OpenMRS Core, and Apache OFBiz (Project repositories: https://github.com/spring-projects/spring-petclinic (accessed on 29 June 2026), https://github.com/BroadleafCommerce/BroadleafCommerce (accessed on 29 June 2026), https://github.com/openmrs/openmrs-core (accessed on 29 June 2026), https://github.com/apache/ofbiz-framework (accessed on 29 June 2026).), covering different architectural approaches and scales (from 250 to over 600 classes).

The graph neural network was trained on a graph where each node had a predefined cluster membership, i.e., the system learned to reproduce the clustering structure that was obtained in the previous steps using a heuristic approach.

The goal of training the model was to form a vector representation of the nodes of the architectural graph, in which nodes from the same structural or functional context are located closer to each other, and nodes from different contexts are located further apart in the feature space. By using a modified loss function, the model learned a generalized representation of the structure of the software system rather than reproducing specific cluster labels.

As a result of training a graph neural network, a space of vector representations (embeddings) was obtained, where each node of the graph corresponds to a vector that summarizes its structural characteristics, semantic properties, and the context of interaction with other nodes in the dependency graph.

In the testing phase, the model was applied to an industrial project that was used to evaluate the application of the heuristic approach. For the test project, an architectural dependency graph was first constructed and initial feature vectors of nodes were created, then the model performed a direct traverse of the graph and calculated class embeddings, which served as input data for the subsequent clustering phase.

After applying the neural network, the semantic consistency improved compared to the heuristic approach. Its value increased from 0.4883 to 0.6358, which means that the semantic homogeneity of components belonging to the same domain increased. It can be concluded that the model can not only preserve the structure of the initial clustering but also improve it in terms of subject consistency. The average cohesion value, compared to the initial clustering, increased across the system from 0.0752 to 0.0890, or 18.3%. This means that the internal density of connections within the clusters has increased. Comparison of the internal consistency values of individual clusters for heuristic and GNN approaches is presented in Figure 3.

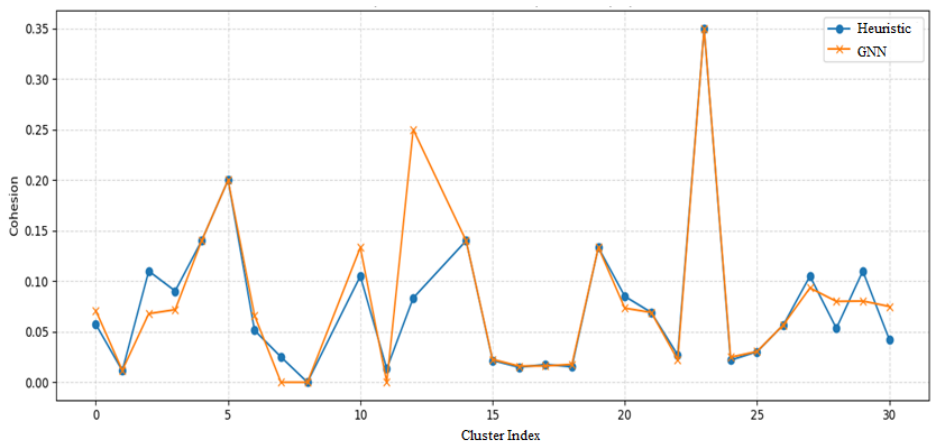

Figure 3. Comparison of cluster internal consistency values for heuristic and GNN approaches.

An even more important result was the reduction in coupling. The average coupling decreased from 4.20 to 3.36 (i.e., by 20%), and the total number of inter-cluster connections in the system decreased from 88.2 to 67.2, i.e., by 23.8%. Comparison of the inter-service connectivity values of individual clusters for heuristic and GNN approaches is presented in Figure 4.

3.6. Comparative Analysis with Existing Approaches

At the next stage, a comparative analysis of the proposed approach with modern graph methods—CoGCN, DEEPLY, HyDEC, GDC-DVF and Mono2Micro—was carried out. Following the experimental protocol described in the literature, the basic methods were evaluated using implementations and hyperparameter settings recommended by their authors. This ensured consistent experimental conditions and correct comparison of the obtained results. The Structural Modularity (SM) and Inter-Call Percentage (ICP) metrics are used for comparison [13].

The SM metric is used to assess the structural quality of a monolithic system decomposition and characterizes the balance between a cluster's cohesion and intercluster dependencies. Higher SM values correspond to better structural modularity of the decomposition.

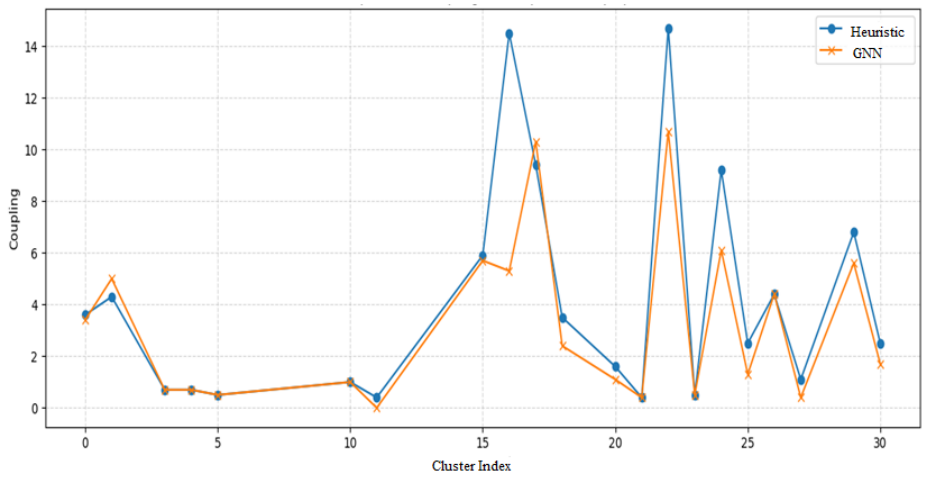

Figure 4. Comparison of inter-service connectivity of clusters for heuristic and GNN approaches.

The value of the metric is determined by the formula:

$$SM = \frac{1}{k}\sum_{i=1}^{k}\frac{e_i}{n_i^2} - \frac{2}{k(k-1)}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\frac{e_{ij}}{2n_i n_j}, \tag{33}$$

where:

- k—the number of clusters;
- e_i—the number of edges inside cluster i;
- n_i—the number of classes in cluster i;
- e_{ij}—the number of edges between clusters i and j;
- n_j—the number of classes in cluster j.

Inter-Call Percentage (ICP) is used to evaluate the degree of interaction between clusters and represents the proportion of inter-cluster calls relative to the total number of calls in the system. Lower ICP values indicate weaker coupling between clusters and a higher quality decomposition.

The metric is computed as follows:

$$ICP = \frac{E_{external}}{E_{internal} + E_{external}}, \tag{34}$$

where:

- $E_{external}$—the total number of calls (edges) between different clusters;
- $E_{internal}$—the total number of calls (edges) within clusters.

The methods were applied to an industrial project. The results are presented in Table 4.

Table 4. Comparative results of structural decomposition: Industrial project.

Method	SM	ICP
CoGCN	0.027	0.559
DEEPLY	0.040	0.546
HyDEC	0.080	0.425
GDC-DVF	0.041	0.446
Mono2Micro	0.030	0.658
Proposed approach	0.129	0.295

The results obtained for the industrial project demonstrate the superiority of the proposed approach over all the compared methods in terms of structural modularity and inter-

cluster connectivity. SM = 0.1295 is higher than CoGCN (0.027), DEEPLY (0.040), GDC-DVF (0.041) and Mono2Micro (0.030) and also exceeds the result of HyDEC (0.080). ICP = 0.2950 is the lowest among all the considered approaches. CoGCN (0.559), DEEPLY (0.546), Mono2Micro (0.658) and HyDEC (0.425) demonstrate a significantly higher level of inter-cluster dependencies.

3.7. Complexity and Scalability Analysis

The proposed approach works with a directed dependency graph containing V nodes and $|E|$ edges. Graph construction, dependency filtering, and most of the preprocessing and analysis steps of the graph are performed with linear or quasi-linear complexity relative to its size. A sparse representation is used to store the graph, so the memory requirements are $O(|E|)$ for the graph structure and $O(V \cdot d)$ for the node feature matrix, where d is the size of the feature vector.

The SimpleGCN model propagates information directly along the edges of the dependency graph. For each layer of the graph convolution, the computational complexity is proportional to the number of edges in the graph and the size of the hidden representation. Thus, the total training complexity is estimated as $O(epochs \cdot |E| \cdot h)$, where h is the size of the hidden layer. Since dependency graphs of software systems are usually sparse, computational costs grow mostly linearly with increasing number of dependencies, which makes the proposed approach suitable for analyzing large enterprise-level software systems.

4. Discussion

The obtained experimental results confirm the initial working hypothesis that the complex structural use of graph analysis, dynamic clustering algorithm selection, cluster structure improvement, and graph neural network provide an increase in the quality of decomposition of monolithic software systems on a microservice.

The initial decomposition of the software system was obtained using a heuristic approach that used the structural characteristics of the dependency graph and a set of rules for forming initial clusters. For the industrial system under study, this approach provided a cohesion value of 0.0682, a total number of intercluster dependencies of 119.00, and a semantic consistency of 0.4883. The results obtained confirm the ability of the heuristic approach to form an initial decomposition, but there remains significant potential for improving both the structural and semantic quality of the clusters.

At the next stage, a cluster improvement procedure was applied. The application of the cluster structure improvement method made it possible to significantly increase the internal connectivity of modules and reduce the number of external dependencies between clusters. The average value of cluster cohesion increased from 0.0682 to 0.0752, which means strengthening the internal structural integrity of clusters; coupling decreased from 119.00 to 88.20, which means increasing modular isolation; at the same time, the semantic consistency of clusters increased from 0.4883 to 0.6358, which confirms a better alignment of the structural decomposition with the domain logic.

Further application of the graph neural model to an industrial software project allowed us to take into account not only the structural characteristics of the dependency graph but also the semantic features of the software components. As a result, the proposed approach provided higher cohesion compared to the heuristic method (0.0890 vs. 0.0752, 18.3%), which had a lower number of intercluster dependencies (67.2 vs. 88.2, 23.8%). The results obtained indicate that the use of a graph neural network allows us to more effectively identify hidden connections between software components and form a higher-quality decomposition of the software system.

Compared to the basic and modern approaches, the proposed method demonstrated consistently higher values of structural modularity (SM) and lower inter-service coupling (ICP) indices, which indicates a more balanced and architecturally justified decomposition.

For an industrial software project, the proposed approach achieved an SM value of 0.129, which significantly exceeds the results of alternative methods, in particular IBM Mono2Micro (0.030), CoGCN (0.027), DEEPLY (0.040) and GDC-DVF (0.041). The structural modularity index characterizes the level of internal integrity of the formed services while minimizing unnecessary dependencies between them. Accordingly, the growth of this indicator indicates that the defined service boundaries better correspond to the real business functions and technical areas of responsibility of the system. This is especially important for corporate information systems, where poor-quality decomposition often leads to excessive inter-service interaction, duplication of logic and complication of further support.

At the same time, the proposed method demonstrated the lowest ICP value (0.295) among all the compared approaches. Since this indicator characterizes the level of structural dependence between the allocated services, its reduction means a reduction in the number of inter-service calls and a higher level of functionality encapsulation. From a practical point of view, this can provide better scalability, simplify independent deployment of components, and reduce the risk of cascading failures in a distributed environment. Compared to Mono2Micro (0.658) and CoGCN (0.559), the obtained advantage is significant, which confirms the limitations of approaches focused exclusively on clustering or similarity of components without taking into account hidden structural dependencies.

From the perspective of previous studies, the results obtained are consistent with the current trend of using graph models for analyzing software architectures. Early methods, such as Service Cutter, mainly relied on manually defined criteria for the relationships between modules, while later approaches such as CoGCN or CHGNN integrated graph neural networks to learn from the structure of dependencies. However, a significant part of existing solutions uses fixed clustering strategies or static graph partitioning schemes. The results obtained indicate the feasibility of using an adaptive multi-step approach, which includes the choice of a clustering algorithm that depends on the structural characteristics of the graph, further improvement of the cluster structure, and the use of a graph neural network to automate the process of decomposition of software systems.

The results also support the hypothesis that combining structural and semantic information is more effective than using only source code dependencies. In real software systems, individual classes may be technically related but implement different business functions or vice versa; they belong to the same subject area with minimal direct dependencies.

However, some limitations of the study should be noted. First, despite the use of an industrial project, additional testing on very large corporate monoliths with thousands of classes would allow for a better assessment of the scalability of the method. Second, structural metrics such as SM and ICP, while important, do not take into account organizational, operational, and team factors that also affect migration success. Third, the use of graph neural models may require additional computational resources and parameter tuning for heterogeneous code bases.

Further research can be directed in several promising directions. First of all, it is advisable to include dynamic data—execution traces, transaction logs, API call statistics—in the graph model to combine static and behavioral analysis. Another promising direction is the use of large language models to semantically enrich source code elements and more precisely define the boundaries of subject areas. Of particular interest is the application of multi-criteria optimization methods that simultaneously take into account modularity, interaction delays, fault tolerance, and deployment cost. Also important are long-term industrial case studies, within which it is possible to evaluate not only architectural metrics

but also business indicators: development speed, number of defects, release stability and operational efficiency.

Thus, the results obtained demonstrate that the proposed approach is not only competitive with existing solutions but also represents a practically significant step forward in the field of automated microservices allocation. The combination of graph analytics, adaptive clustering and machine learning methods forms a reliable basis for the further development of intelligent software modernization systems.

5. Conclusions

The study proposes an approach to the decomposition of monolithic software systems, which combines the heuristic formation of initial clusters, their further improvement based on the structural characteristics of the graph and the use of a graph neural network to automate the clustering process. The proposed approach takes into account the structural dependencies between software components, the semantic characteristics of classes and the topological properties of the dependency graph; this allows for the formation of a higher-quality decomposition of the software system. The trained graph neural model not only reproduces the patterns identified during the operation of the heuristic algorithm but also provides additional improvement of the results in terms of cohesion, coupling and semantic consistency, which is confirmed by the results of an experimental study on an industrial software system.

Comparative analysis with existing approaches showed the competitiveness of the proposed method and its effectiveness for the decomposition of complex software systems. The obtained results demonstrated a high level of structural modularity and a low level of inter-cluster dependencies, which confirms the feasibility of using the proposed approach for the automated removal of microservices from monolithic software systems.

Prospects for further research include expanding the capabilities of the proposed approach, which consists of applying the method of dynamic analysis of program execution, improving semantic models by including information from documentation and API interfaces in the feature vector, and exploring the possibility of using neural network models for the automated distribution of the database structure between services [16–18].

Author Contributions: Conceptualization, Y.K. and S.Y.; methodology, Y.K., S.Y. and O.H.; software, O.H.; validation, Y.K., S.Y. and P.Y.; formal analysis, Y.K. and O.H.; investigation, Y.K., S.Y., O.H. and P.Y.; writing—original draft preparation, O.H.; writing—review and editing, Y.K., S.Y., P.Y. and T.P.; visualization, O.H.; supervision, Y.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Dragoni, N.; Giallorenzo, S.; Lluch Lafuente, A.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 195–216.
2. Mazlami, G.; Cito, J.; Leitner, P. Extraction of Microservices from Monolithic Software Architectures. In *2017 IEEE International Conference on Web Services (ICWS)*; IEEE: New York, NY, USA, 2017.
3. Martínez Saucedo, A.; Rodríguez, G.; Gomes Rocha, F.; Pereira dos Santos, R. Migration of Monolithic Systems to Microservices: A Systematic Mapping Study. *Inf. Softw. Technol.* **2024**, *177*, 107590. [CrossRef]
4. Seedat, M.; Abbas, Q.; Ahmad, N. Systematic Mapping of Monolithic Applications to Microservices Architecture. *arXiv* **2023**, arXiv:2309.03796.

5. Abdelfattah, A.S.; Cerny, T. The Microservice Dependency Matrix. In *Service-Oriented and Cloud Computing*; Springer: Berlin/Heidelberg, Germany, 2023; Volume 14183, pp. 276–288.
6. Zhong, C.; Zhang, H.; Li, C.; Huang, H.; Feitosa, D. On Measuring Coupling Between Microservices. *J. Syst. Softw.* **2023**, *200*, 111670. [CrossRef]
7. Taibi, D.; Systä, K. A Decomposition and Metric-Based Evaluation Framework for Microservices. In *Software Architecture*; Springer: Berlin/Heidelberg, Germany, 2019.
8. Trabelsi, I.; Mahmoudi, B.; Minani, J.B.; Moha, N.; Guéhéneuc, Y.-G. A Systematic Literature Review of Machine Learning Approaches for Migrating Monolithic Systems to Microservices. *IEEE Trans. Softw. Eng.* **2025**, *51*, 2972–2995. [CrossRef]
9. Desai, U.; Bandyopadhyay, S.; Tamilselvam, S. Graph Neural Network to Dilute Outliers for Refactoring Monolith Application. *Proc. AAAI Conf. Artif. Intell.* **2021**, *35*, 72–80. [CrossRef]
10. Qian, L.; Li, J.; He, X.; Gu, R.; Shao, J.; Lu, Y. Microservice extraction using graph deep clustering based on dual view fusion. *Inf. Softw. Technol.* **2023**, *158*, 107171. [CrossRef]
11. Yedida, R.; Krishna, R.; Kalia, A.; Menzies, T.; Xiao, J.; Vukovic, M. An Expert System for Redesigning Software for Cloud Applications. *Expert Syst. Appl.* **2023**, *219*, 119673. [CrossRef]
12. Sellami, K.; Saied, M.A.; Ouni, A. A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering (EASE '22)*; Association for Computing Machinery: New York, NY, USA, 2022.
13. Ziabakhsh, A.; Rezaee, M.; Eskandari, M.; Goudarzi, M. Mo2oM: Extracting Overlapping Microservices from Monolithic Code via Deep Semantic Embeddings and Graph Neural Network–Based Soft Clustering. *arXiv* **2025** arXiv:2508.07486.
14. Chaieb, M.; Saied, M.A. Migration to Microservices: A Comparative Study of Decomposition Strategies and Analysis Metrics. *arXiv* **2024**, arXiv:2402.08481.
15. Maharjan, R.; Sooksatra, K.; Cerny, T.; Rajbhandari, Y.; Shrestha, S. A Case Study on Monolith to Microservices Decomposition with Variational Autoencoder-Based Graph Neural Network. *Future Internet* **2025**, *17*, 303. [CrossRef]
16. Krause, A.; Zirkelbach, C.; Hasselbring, W.; Lenga, S.; Kröger, D. Microservice Decomposition via Static and Dynamic Analysis of the Monolith. In *IEEE International Conference on Software Architecture Companion (ICSA Companion)*; IEEE: New York, NY, USA, 2020; pp. 9–16.
17. Mohottige, T.I.; Polyvyanyy, A.; Fidge, C.; Buyya, R.; Barros, A. Reengineering Software Systems into Microservices: State-of-the-Art and Future Directions. *Inf. Softw. Technol.* **2025**, *183*, 107732.
18. Oumoussa, I.; Saidi, R. Evolution of Microservices Identification in Monolith Decomposition: A Systematic Review. *IEEE Access* **2024**, *12*, 107732. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.