# Using Static Analysis to Aid Monolith to Microservice System Transformation: Tuning Fuzzy c-Means in a VAE-Based GNN Approach

Korn Sooksatra
korn_sooksatra1@baylor.edu
CS, Baylor University
Waco, Texas, USA

Md Showkat Hossain Chy
chym@arizona.edu
SIE, University of Arizona
Tucson, Arizona, USA

Md Ashfakur Rahman Arju
muhammadashfaku.arju@student.montana.edu
CS, Montana State University
Bozeman, Montana, USA

Tomas Cerny
tcerny@arizona.edu
SIE, University of Arizona
Tucson, Arizona, USA

Pablo Rivas
pablo_rivas@baylor.edu
CS, Baylor University
Waco, Texas, USA

## Abstract

Transitioning from monolithic systems to cloud-native based on microservice architecture is essential for organizations facing dynamic technological shifts and growing scalability demands. This paper explores a machine learning-driven approach to decompose monolithic systems into microservices, targeting maintainability and modularization. Utilizing static analysis, we extract critical dependency data from the monolith, which guides the configuration of a Variational Autoencoder (VAE) and fuzzy c-means clustering process. This approach enables precise tuning of hyperparameters to optimize the decomposition into highly independent, scalable microservices. Our findings highlight the effectiveness of integrating static analysis with machine learning to enhance the adaptability and efficiency of distributed systems, providing valuable insights into the nuanced impacts of hyperparameter adjustments on system performance. Furthermore, we provide a novel system multi-variant benchmark to the community.

## Keywords

Microservices, Mono-to-Micro, Static Analysis, Distributed Systems, Variational Autoencoder, Graph Neural Networks

**ACM Reference Format:**
Korn Sooksatra, Md Showkat Hossain Chy, Md Ashfakur Rahman Arju, Tomas Cerny, and Pablo Rivas. 2024. Using Static Analysis to Aid Monolith to Microservice System Transformation: Tuning Fuzzy c-Means in a VAE-Based GNN Approach. In *39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3691621.3694933

## 1 Introduction

In software architecture, monolithic systems have traditionally been favored for their simplicity and straightforward development process, consolidating all functionalities into a single, unified structure. Monoliths, characterized by their simplicity and ease of initial development, have played a pivotal role in the evolution of applications. However, as technology landscapes evolve and business requirements become more dynamic, the limitations of monolithic architectures have become apparent.

The motivation to transition from monoliths to microservices is rooted in the quest for enhanced agility, scalability, and adaptability. Monolithic architectures, while effective in their simplicity, often encounter challenges when it comes to accommodating rapid changes, selective service scalability, and facilitating efficient development practices [19]. Microservices offer a compelling alternative, allowing organizations to break down their applications into modular, independently deployable services. This transition enables teams to iterate faster, scale specific functionalities independently, and respond more effectively to evolving business needs. The motivation for this shift lies in the pursuit of a more dynamic, responsive, and scalable software ecosystem.

Amidst these architectural transformations, the role of static analysis has become increasingly significant. Static analysis provides a systematic examination of the codebase without executing the programs, offering insights into complex dependencies and potential architectural divisions. This technique is essential for identifying and understanding the tightly-knit components within monolithic architectures that must be carefully separated during the transition to microservices. By employing static analysis, developers can pinpoint service boundaries more accurately and define clearer interfaces between services, as discussed by Kalske et al. (2018) [13]. Chy et al. [6] introduces a methodology for transforming and optimizing existing microservice architectures using machine learning techniques, specifically Graph Neural Networks (GNN) and Variational Autoencoder (VAE). It aims to adapt methods initially designed for monolith-to-microservices migration to the optimization of microservice systems proposed by Sooksatra et al [21].

As the software engineering landscape continues to embrace advancements in artificial intelligence, the integration of machine learning into the process of decomposing monoliths adds a layer of intelligence and efficiency. The motivation for employing machine


*ASEW '24, October 27-November 1, 2024, Sacramento, CA, USA*

ACM ISBN 979-8-4007-1249-4/24/10
https://doi.org/10.1145/3691621.3694933

learning in this context stems from the desire to make decomposition decisions more informed, data-driven, and adaptive. By leveraging machine learning algorithms, organizations can automate the identification of optimal service boundaries and intelligently distribute functionalities across microservices.

This paper delves into the intricate details of the machine learning-based decomposition method [21]. The focal point of our investigation lies in utilizing static analysis insights from monolith systems to drive the nuanced process of tuning a crucial hyperparameter associated with the fuzzy *c*-means clustering algorithm recommending appropriate system division. As we explore this machine learning methodology, our research unveils the profound impact that fine-tuning this specific hyperparameter exerts on the overall performance of microservices. The empirical findings and analyses presented herein shed light on the intricate relationship between hyperparameter adjustments and the resulting efficacy of the decomposition process.

In essence, our study extends existing machine learning-based decomposition method [21] by providing nuanced insights into the role of hyperparameter tuning as a key determinant in optimizing microservices performance within the context of machine learning-based decomposition strategies. This work demonstrates that the approach [21] is not only theoretical but also practical and realistic, showcasing the feasibility and benefits of our proposed method. Furthermore, to validate the method, we use a large microservice system benchmark, convert it to a monolith, and decompose it based on our algorithm, followed by implementation, deployment, and testing, assessing our approach's feasibility and comparing the result with the original system baseline. Finally, given the community's lack of system multi-variant benchmarks to validate and compare mono-to-micro methods to foster advancements in research in this direction, we provide a new three-system-variant benchmark.

The manuscript is organized as follows: Section 2 briefly describes previous works similar to ours; Section 3 formulates the problem into an optimization problem; Section 4 describes our proposed solution; Section 5 shows how to apply our approach on an actual system, demonstrates the evaluation that compares our approach with several hyperparameters and illustrates the results; Section 6 concludes everything in the entire work.

## 2 Related Works

The transformation of monolithic applications into microservices architectures has been a dynamic journey, blending software engineering techniques with the innovative application of machine learning. The early exploration of microservices decomposition was marked by methodologies grounded in software engineering principles. In 2017, Chen et al. [5] presented a dataflow-driven approach, leveraging Data Flow Diagrams (DFD) to unearth potential microservices clusters. This approach focused on grouping similar operations and data types for effective decomposition and promoted cohesion within microservices. However, it heavily depended on DFDs, which risked leading to inconsistent interpretations. Expanding on this, Taibi and Systa [22] in 2019 proposed a detailed framework, integrating both static and dynamic analyses within a six-step schema, which, despite its comprehensiveness, revealed limitations due to its reliance on expert input. Additionally, it also lacked mechanisms for crucial tasks such as process identification and the evaluation of decomposition quality.

Progressing further, Krause-Glau et al. [15] in 2020 integrated the bounded-context concept from domain-driven design with static and dynamic analyses for microservices extraction. This approach meticulously refined the segmentation of monolithic systems into microservices. The resultant architecture exhibited a notable improvement in service cohesion and a marked reduction in coupling. In 2021, Auer et al. [2] introduced an industrial assessment framework, focusing on key metrics for transitioning to a microservices architecture, such as functional stability and maintainability.

A significant paradigm shift occurred with the incorporation of machine learning techniques. IBM's Mono2Micro [12], introduced in 2021, employed hierarchical clustering for partitioning application components, complemented by a web interface for visualizing architecture and microservices boundaries. Eski and Buzluca [9] further advanced automated extraction strategies by combining analyses with agglomerative hierarchical algorithms validated against expert benchmarks.

The impact of machine learning extended to nuanced partitioning strategies. Abdullah et al. [1], in 2021, presented a URI-based partitioning method using the k-means algorithm, and Kalia et al.'s Mono2Micro framework [12] in 2020 optimized clustering for microservice segmentation using spatio-temporal decomposition, allowing for the dynamic collection and analysis of runtime call traces while preserving temporal relationships.

The advent of graph neural networks (GNNs) marked another milestone in microservices partitioning. Desai et al. [7] utilized graph convolution networks (GCN) for clustering, while subsequent studies, such as Mathai et al. [17], adopted heterogeneous graph neural networks, demonstrating the flexibility of this approach. Yedida et al. [25] optimized existing machine-learning-based methodologies, refining the partitioning process.

Recently, Trabelsi et al. [23] developed "MicroMiner", combining machine learning and semantic analysis for transitioning legacy systems into microservices. Filippone et al. [10] proposed an approach using graph clustering and the Louvain community algorithm for creating cohesive microservices, highlighting scaling challenges in larger systems. Moreover, Google's Service Weaver [11] emerged, automating application deployment to cloud-based microservices, focusing on maintaining business logic integrity. This contrasts with our approach, which provides a systematic guide for redesigning and restructuring microservices architectures, underscoring our contribution to this evolving field.

Sooksatra et al.[21] introduced a framework that utilizes the variational autoencoder[14] and the fuzzy *c*-means algorithm [3] to decompose a monolithic application into microservices whereas Chy et al. [6] proposed a similar approach for optimizing existing microservices. Our work is the first that investigates various hyperparameters of the fuzzy *c*-means algorithm to enhance the resulting decomposition and builds upon the framework in [21].

To provide an overview of the methodologies used in transforming monolithic applications into microservices architectures, we summarize key related works. The following table highlights the main techniques discussed in these studies. It includes approaches such as data flow analysis, static and dynamic analyses, machine

learning algorithms, and more. This comparison helps to understand the evolution and variety of strategies employed in this field.

## 3 Problem Formulation

The starting point involves dealing with a monolithic application consisting of $n$ classes, denoted as $C = c_1, c_2, \ldots, c_n$. The objective is to partition this monolithic application into sets of classes that form cohesive microservices. Achieving this requires balancing the granularity of the partitioning: too coarse-grained, and we reintroduce monolithic issues; too fine-grained, and we risk performance degradation due to excessive inter-service communication and interdependencies.

To address these challenges, we establish several objectives. First, we aim to ensure strong interdependence among classes within each microservice, meaning classes that frequently interact should ideally reside in the same microservice. Second, we seek to minimize inter-microservice communication, as excessive interactions between microservices can lead to performance bottlenecks. Third, while we allow some classes to be duplicated across multiple microservices to reduce inter-service communication, we also aim to minimize such duplications to maintain code manageability and reduce technical debt.

The decomposition process involves analyzing the interaction patterns among classes in the monolithic application. We model these interactions to quantify the relationships and communication needs between classes. Specifically:

- **Distance between classes ($d_1(c_1, c_2)$):** This metric measures the interaction frequency and dependency strength between classes $c_1$ and $c_2$. A lower distance indicates stronger interaction and higher interdependence.
- **Communication between microservices ($d_2(m_1, m_2)$):** This metric captures the extent of communication required between microservices $m_1$ and $m_2$. Lower communication requirements are preferable to reduce latency and overhead.
- **Duplicated classes ($d_3(c)$):** This metric counts the number of microservices in which a class $c$ is duplicated. Minimizing this count helps maintain code maintainability.

We formalize the problem as a multi-objective optimization problem, aiming to minimize a combination of these factors:

$$\min_{M} \sum_{m \in M} \sum_{c_1 \in m} \sum_{c_2 \in m} d_1(c_1, c_2) + \lambda_1 \sum_{m_1 \in M} \sum_{m_2 \in M} d_2(m_1, m_2) + \lambda_2 \sum_{c \in C} d_3(c),$$

where $M$ represents the set of microservices, $C$ represents the set of classes, $d_1(c_1, c_2)$ quantifies the distance between classes $c_1$ and $c_2$, $d_2(m_1, m_2)$ indicates the extent of communication between microservices $m_1$ and $m_2$, and $d_3(c)$ signifies the number of occurrences of class $c$ across microservices. The parameters $\lambda_1$ and $\lambda_2$ act as weights to balance the importance of each objective in the optimization problem.

Given the complexity of this problem, finding a global optimal solution is impractical. Instead, we employ fuzzy algorithms and neural networks to seek a local minimum, providing a feasible and effective approach to partitioning the monolithic application into well-defined microservices.

## 4 Our Approach

Our methodology draws inspiration from works presented in [8, 21], and this section outlines details. Our approach, visualized in Fig. 1, comprises three essential steps: Data Preparation, Embedding-Vector Creation, and Clustering. The input is a monolith application, and the output is microservices. Note that clusters in Clustering refer to microservices.

In the preliminary step focused on data preparation, a diverse set of tools is utilized to extract valuable insights and construct a dependency graph based on the application's components. This graph is then subjected to preprocessing to generate a feature matrix, providing a foundation for subsequent stages.

Proceeding to the subsequent stage, a graph convolution network is employed to process the information derived from the dependency graph. This process results in the extraction of embedding vectors for individual nodes within the graph, contributing to a deeper understanding of their interdependencies.

Finally, the clustering utilizes the fuzzy $c$-means algorithm to delineate and identify microservices based on the generated embedding vectors. This multi-step process forms the backbone of our approach, effectively partitioning the application into its respective microservices.

(1) **Data Preparation:** To facilitate the input for our machine learning model, we collected three distinct types of data from a monolithic application: a dependency graph, an entrypoint existence matrix, and an entrypoint co-existence matrix. The dependency graph can be derived using static analysis applied to the monolith system. In our notation, the set of entrypoints is represented as $P$. The dependency graph, denoted as $A$, takes the form of a matrix enumerating all classes upon which a specific class depends. The entrypoint existence matrix, labeled $E$, conveys the involvement of classes in paths initiated by different entrypoints. In this matrix, $E_{ij}$ is set to 1 when class $i$ is part of at least one path initiated by entrypoint $j$. The entrypoint co-existence matrix, denoted as $Co$, provides insights into the frequency of co-occurrence of two classes in paths initiated by the same entrypoints. Here, $Co_{ij}$ represents the number of instances where class $i$ and class $j$ are present within the same entrypoint-initiated paths.
To illustrate this process, consider a simplified monolithic application with the following classes and entrypoints:
- Classes: $C = \{c_1, c_2, c_3, c_3\}$
- Entrypoints: $P = \{p_1, p_2\}$

Assume the following dependencies between classes:
- $c_1$ depends on $c_2$
- $c_2$ depends on $c_3$
- $c_3$ depends on $c_4$

From these dependencies, we construct the dependency graph $A$:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Next, consider the paths initiated by the entrypoints:
- For $p_1$, assume the path is $c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4$
- For $p_2$, assume the path is $c_2 \rightarrow c_3 \rightarrow c_4$

| Approach | Chen et al. (2017) | Taibi and Systa (2019) | Krause-Glau et al. (2020) | Auer et al. (2021) | Kalia et al. (2021) | Eski and Buzluca (2021) | Abdullah et al. (2021) | Desai et al. | Mathai et al. (2021) | Yedida et al. | Trabelsi et al. | Filippone et al. | Grandl et al. (2023) | Sooksatra et al. (2022) | Chy et al. (2024) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Data Flow Diagrams (DFD) | ✓ | | | | | | | | | | | | | | |
| Static Analysis | | ✓ | ✓ | | | | | | | | | | | | |
| Dynamic Analysis | | ✓ | ✓ | | | | | | | | | | | | |
| Bounded- Context | | | ✓ | | | | | | | | | | | | |
| Hierarchical Clustering | | | | | ✓ | ✓ | | | | | | | | | |
| K-Means Algorithm | | | | | | | ✓ | | | | | | | | |
| Graph Convolution Networks (GCN) | | | | | | | | ✓ | | | | | | | |
| Heterogeneous Graph Neural Networks | | | | | | | | | ✓ | | | | | | |
| Semantic Analysis | | | | | | | | | | ✓ | ✓ | | | | |
| Louvain Community Algorithm | | | | | | | | | | | | ✓ | | | |
| Variational Autoencoder | | | | | | | | | | | | | ✓ | ✓ | ✓ |
| Fuzzy C-Means Algorithm | | | | | | | | | | | | | ✓ | ✓ | ✓ |
| Metrics for Microservices | | | | ✓ | | | | | | | | | | | |
| Advanced Partitioning | | | | | | | | | | ✓ | | | | | |

**Table 1: Comparison of related researches**

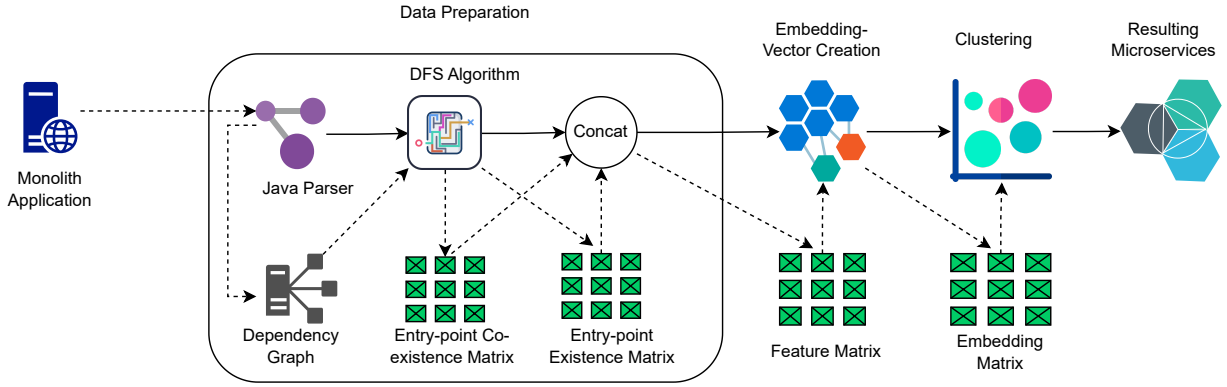



**Figure 1: The working flow of our approach**

The entrypoint existence matrix $E$ is then constructed by performing a depth-first search (DFS) from each entrypoint:

$$E = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

In this matrix, $E_{ij}$ is set to 1 if class $i$ is part of a path initiated by entrypoint $j$. For example, class $A$ is part of the path initiated by $p_1$ but not by $p_2$.

The entrypoint co-existence matrix $Co$ is built by counting the number of times each pair of classes appears in the same path:

$$Co = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 2 & 2 \\ 1 & 2 & 0 & 2 \\ 1 & 2 & 2 & 0 \end{bmatrix}$$

For instance, classes $c_2$ and $c_3$ appear together in paths initiated by both $p_1$ and $p_2$, so $Co_{c_1c_2} = 2$.

To extract information from the system source code, parsers are generally available and produce an Abstract Syntax Tree (AST) representation of the code. The AST serves as the foundation for extracting essential data from the code. We initially obtained a list of all classes for generating the dependency graph related to a specific class. Subsequently, we filtered this list to include only classes located within the project's root package, excluding those imported from external libraries.

The derivation of the entrypoint existence matrix $E$ involved implementing depth-first search (DFS) within the dependency graph. DFS was executed using class $i$ as the subject, designating entrypoint $j$ as the root. When class $i$ was encountered at any point during the DFS traversal, $E_{ij}$ was set to 1. In parallel, to construct the entrypoint co-existence matrix $Co$, DFS was again employed within the dependency graph. This process entailed enumerating all paths initiated by the entrypoints. Given class $i$ and class $j$, $Co_{ij}$ was set to the number of paths where these two classes concurrently existed. Following this data collection, we formed a feature matrix, represented as $\tilde{X}$, as the concatenation of $E$ and $Co$. Symbolically, this can be expressed as

$$\tilde{X} = E \odot Co,$$

where the operator $\odot$ signifies concatenation. As a result, the size of $\tilde{X}$ is determined as $|C| \times (|P| + |C|)$. Subsequently, we

employed graph convolutional network (GCN) techniques to normalize $\tilde{X}$ based on the adjacency classes within the graph. The normalization process is denoted as

$$X = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}\tilde{X}, \quad (1)$$

where $\tilde{A}$ is $A$ added to the identity matrix $I$, $\tilde{D}$ represents the degree diagonal matrix, and $\tilde{D}_{ii}$ is calculated as the summation of $\tilde{A}_{ij}$ for all $j$ in $C$.

(2) **Embedding-Vector Creation:** Building upon the dependency graph $A$ and feature matrix $X$, our approach employs a variational autoencoder (VAE) [14] to generate an embedding matrix $Z$. This matrix captures probabilistic information within the latent space, allowing for a more nuanced understanding of relationships between classes.
The process of generating the embedding vector using the VAE involves the following steps:
   (a) **Encoder:** The encoder network takes the feature matrix $X$ as input and maps it to a latent space to produce the mean $\mu$ and the standard deviation $\sigma$ of the latent variables. This is expressed as:
   $$\mu, \sigma = \text{Encoder}(X)$$
   (b) **Latent Space Sampling:** From the mean $\mu$ and standard deviation $\sigma$, we sample the latent variable $Z$ using the reparameterization trick:
   $$Z = \mu + \sigma \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$
   (c) **Decoder:** The decoder network then reconstructs the input feature matrix $X$ from the latent variable $Z$:
   $$\hat{X} = \text{Decoder}(Z)$$
   The VAE is trained to minimize a loss function that comprises three components:
   - **Reconstruction Loss:** This measures the difference between the original feature matrix $X$ and its reconstructed version $\hat{X}$ using Mean Squared Error (MSE):
   $$\sum_{i=1}^{n} ||X_i - \hat{X}_i||_2^2$$
   - **Graph Structure Loss:** This ensures that the latent variables $Z$ preserve the structure of the dependency graph $A$. We use the inner product of $Z$ to approximate $A$:
   $$\sum_{i=1}^{n} ||A_i - Z_i Z^T||_2^2$$
   - **KL Divergence Loss:** This regularizes the latent variables $Z$ to follow a prior distribution (assumed to be a normal distribution), using the Kullback-Leibler (KL) divergence:
   $$\text{KL}(Z, p(Z))$$
   The overall loss function for the VAE is:
   $$\sum_{i=1}^{n} ||X_i - \hat{X}_i||_2^2 + \sum_{i=1}^{n} ||A_i - Z_i Z^T||_2^2 + \text{KL}(Z, p(Z)), \quad (2)$$
   where $p(Z)$ is the prior distribution of $Z$ assumed to follow a normal distribution, $\text{KL}(\cdot, \cdot)$ is the KL divergence function between two distributions, and $M_i$ denotes row $i$ of matrix $M$.
   Incorporating the graph structure into the latent space embedding allows the VAE to capture both the feature information from $X$ and the structural information from $A$, resulting in a comprehensive representation of the classes.

(3) **Clustering:** The clustering stage employs the fuzzy $c$-means (FCM) algorithm [3], chosen for its flexibility in handling scenarios where classes may exhibit varying degrees of membership to multiple microservices. Note that this algorithm has two hyperparameters: the number of microservices and the fuzzy value $m$. The algorithm produces a membership matrix $W$ and a centroid matrix $\Gamma$. No other clustering algorithm offers this advantage naturally.
This stage is crucial for determining the microservice assignments for each class. The FCM algorithm minimizes the objective function, which is a weighted sum of squared distances and membership values:
$$\sum_{i=1}^{k} \sum_{j=1}^{n} W_{ji} ||Z_j - \Gamma_i||_2^2,$$
where $k$ is the number of microservices, $n$ is the number of classes, $Z_j$ is the embedding of class $j$, $\Gamma_i$ is the centroid of microservice $i$, and $W_{ji}$ represents the membership value of class $j$ in microservice $i$.
The resulting assignments are based on memberships greater than a specified maintainability threshold. This threshold is a parameter that ensures classes are assigned to microservices in a way that balances code maintainability and performance. Specifically, the maintainability threshold controls the extent to which a class can belong to multiple microservices. A higher threshold means a class must have a stronger association with a microservice to be assigned to it, reducing the number of duplications across microservices. Conversely, a lower threshold allows more flexibility, increasing the likelihood of a class being shared across multiple microservices, which might improve performance but could also increase maintenance complexity. The maintainability threshold is therefore critical in influencing the final microservice architecture by balancing the trade-offs between code duplication and inter-microservice communication. By adjusting this threshold, we can control how strictly the classes are partitioned, ensuring an optimal balance that aligns with the desired maintainability and performance goals.

Overall, we use the graph convolutional network to create the feature matrix ($X$) and then apply the variational autoencoder to obtain the embedding matrix ($Z$) for classes. At last, we utilize the fuzzy c-means to group all the classes based on their embedding vectors in $Z$.

In this work, we focus on tuning the hyperparameter $m$ of the FCM algorithm in the clustering step to show the significant influence of $m$ on the microservices' performance.

## 5 Case study

Exploring Mono2Micro approaches poses a significant challenge due to the scarcity of large, realistic systems available in both monolithic and microservices versions. Existing system benchmarks, developed primarily for showcasing microservice systems, lack the intention to stress-test a monolith. In response, our semi-automated

methodology not only tackles this challenge but also introduces a novel testing benchmark tailored for the Mono2Micro community.

## 5.1 Case Study Objectives

The selection of an appropriate microservice system for our case study involved a meticulous evaluation of options within the microservices architecture domain [18]. Identifying a critical and complex microservice system that authentically mirrors real-world applications proved to be a formidable task.

In our pursuit of an ideal benchmark, we extensively surveyed available options, considering their attributes. While individual microservices [18] exhibit varying levels of complexity, they often lack the holistic representation found in comprehensive applications. This scarcity of suitable alternatives is a common challenge in microservices architecture research.

Furthermore, simpler microservices designed for experimentation may fall short of capturing the intricacies of real-world systems. Conversely, highly intricate systems may impose impractical resource demands, limiting the scope of extensive research. Enter the train-ticket microservice, striking a balance by embodying real-world intricacies while facilitating comprehensive study.

## 5.2 Selected Microservice System

Our choice for the case study is the train-ticket microservice system [27]. This system stands out for its capacity to represent real-world intricacies while offering a conducive environment for comprehensive study. Moreover, a testing benchmark for this system has been published for full end-to-end test coverage [20].

Widely acknowledged in existing literature [16, 26], the train-ticket system serves as an indispensable reference for evaluating novel methodologies, including the one presented in this study. Comprising 47 microservices, the system orchestrates a train-ticket booking platform with a diverse technological landscape. Programming languages such as Java, Node.js, Python, and Go, coupled with frameworks like Spring Boot/Cloud, Express, Django, and Webgo, contribute to its richness.

For our experimentation and analysis, we focus on a specific subset: the 42 Java-based microservices. This targeted subset forms the core foundation of our research, allowing for a meticulous and in-depth examination of their composition, dependencies, and structural intricacies.

To initiate our experiment, we need an application monolith, and thus, we have manually converted the train ticket system into a monolith version. To validate the correctness of this system version concerning the functionality and offered features, we have performed the full end-to-end test coverage [20] benchmark on both system versions, with all tests passing. This monolith version will be publicly available with a full version of this manuscript.[1]

Moreover, for both systems, we also extracted the system intermediate representation [4] of the service endpoints and the canonical data model (context map) using static analysis; while this gives approximation, it allowed us to evaluate matching data model and assess match of system endpoints.

[1]We have made available the monolithic version of the train-ticket system benchmark in https://zenodo.org/records/11215085

## 5.3 Our Approach on Train-Ticket Application

We navigate through the intricacies of the decomposition process illustrated in Data Preparation, Embedding-Vector Creation, and Clustering as depicted in Fig. 1. The systematic breakdown unfolds as follows:

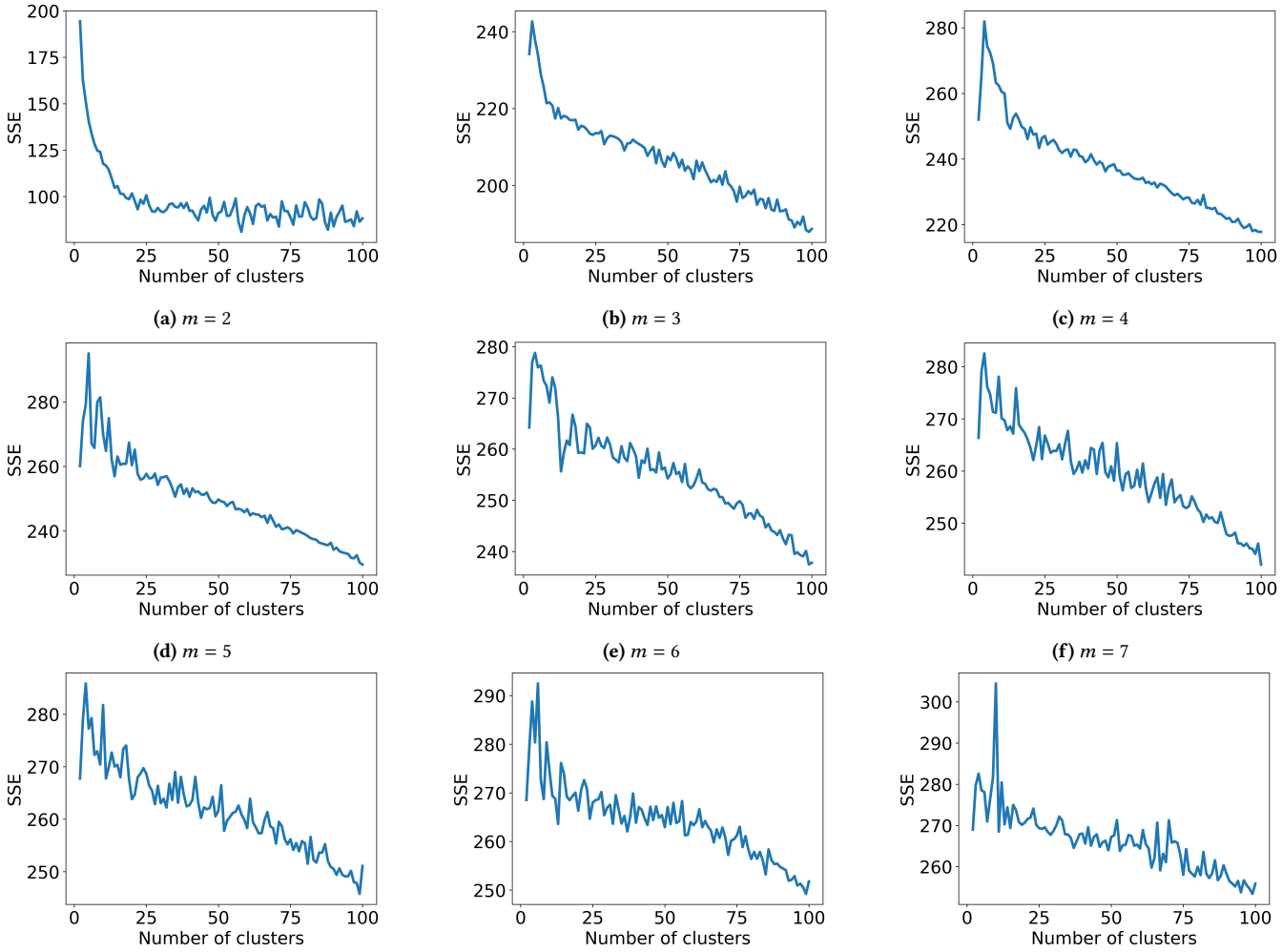
(1) **Data Preparation:** To initiate the decomposition journey, the monolithic version of the train-ticket application undergoes the discerning gaze of Javaparser. This inspection yields a dependency graph $A$, laying the groundwork for subsequent operations. The subsequent steps involve the creation of the entrypoint existence matrix $E$ and the entrypoint co-existence matrix $Co$ through the adept application of the DFS algorithm on $A$. Merging these matrices using (1) crafts the feature matrix $X$, a critical artifact in our quest for microservices.
(2) **Embedding-Vector Creation:** Building upon the foundation laid by $X$, the subsequent stage witnesses the ascendancy of our Variational Autoencoder (VAE). This architectural maestro transforms $X$ into an embedding matrix $Z$, governed by the intricate dance of the loss function encapsulated in (2). The resultant $Z$ emerges as a key player, poised to influence the unfolding narrative. It's noteworthy that, for comparative analysis, we also employ an autoencoder, introducing a nuanced exploration of alternative methodologies.
(3) **Clustering:** During the crucial phase, the FCM algorithm plays a central role, organizing the clustering process on $Z$. Each row in $Z$ represents an embedding vector, capturing the characteristics of each class. We use the elbow method, as explained in our experiments, to help us decide the best number of microservices. Classes are allocated to clusters (or microservices) based on their membership scores exceeding a certain maintainability threshold.

## 5.4 Number of Microservices Selection

In our investigation of the hyperparameters associated with the FCM algorithm, specifically focusing on the number of microservices and the fuzziness coefficient $m$, we conducted an in-depth exploration to discern their impact on the decomposition process. The hyperparameters play a crucial role in influencing the efficiency and effectiveness of the algorithm in partitioning a monolithic application into microservices.

To assess the performance of different configurations, we employed the computation of the sum of squared errors (SSE) as a key metric. The SSE provides valuable insights into the quality of the decomposition, with lower values indicating better results. Our analysis considered a range of settings for the number of microservices, and varying values of the fuzziness coefficient $m$.

Fig. 2 visually represents the SSE across the spectrum of the number of microservices for each distinct value of $m$. A notable observation emerged in the range of 0 to 100 microservices, where the algorithm exhibited optimal SSE minimization with $m = 2$. This finding implies that, within this microservice count range, the fuzziness coefficient of 2 yields a decomposition with minimal errors. It's worth highlighting that exceeding 100 microservices might lead to




**(a)** $m = 2$ **(b)** $m = 3$ **(c)** $m = 4$

**(d)** $m = 5$ **(e)** $m = 6$ **(f)** $m = 7$

**(g)** $m = 8$ **(h)** $m = 9$ **(i)** $m = 10$

**Figure 2: Sum of square error produced by the FCM algorithm for each number of microservices and $m$.**

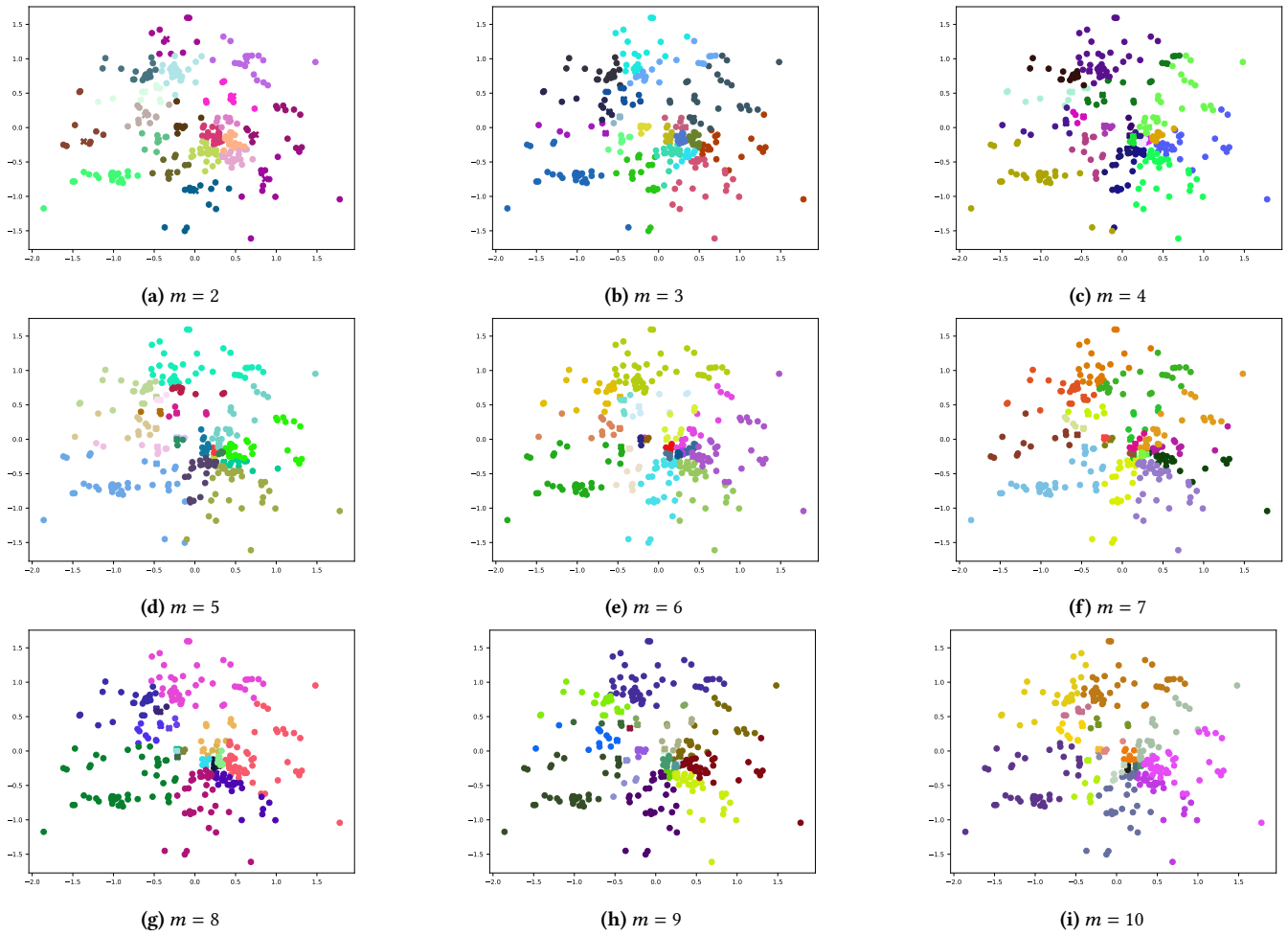
increased maintenance difficulties, prompting a practical constraint on the upper limit of the microservice count.

Interestingly, as we progressed to higher values of $m$, particularly starting from $m = 3$, the characteristic elbow shape, typically utilized in the elbow method to identify an optimal number of clusters, became less evident in the trend. This phenomenon posed a challenge in applying the elbow method to discern the ideal number of microservices for these configurations. Despite the absence of a distinct elbow shape, it is worth noting that such a structural change might become discernible when the number of microservices surpasses 100. Nevertheless, it is essential to acknowledge that such a large number of microservices is not anticipated in practical scenarios.

Our exploration of hyperparameters reveals that the fuzziness coefficient $m = 2$ showcases superior SSE minimization within a reasonable range of microservices, and practical considerations discourage exceeding a certain threshold, emphasizing the need for a balance between decomposition quality and maintainability.

Upon meticulous analysis, a noteworthy finding emerged when $m = 2$, where the number of microservices reached a distinct elbow point at 20 microservices, as illustrated in Fig. 2a. This observation signifies a crucial juncture in the curve where the trade-off between intra-microservice cohesion and inter-microservice separation is optimized. As a result, we have chosen this specific number of microservices, 20, as the basis for subsequent experiments, aiming for a balance between granularity and comprehensibility in the decomposed system.

Fig. 3 vividly depicts the microservice assignments for various values of $m$. Each color represents a distinct microservice, providing a visual representation of the decomposition results. It becomes evident that as the fuzziness coefficient $m$ increases, the microservice assignment becomes more nuanced and blurred. This is prominently illustrated in Fig. 3i, where $m = 10$, showcasing overlapping microservices such as the grey microservice intertwining with several



(a) $m = 2$ (b) $m = 3$ (c) $m = 4$

(d) $m = 5$ (e) $m = 6$ (f) $m = 7$

(g) $m = 8$ (h) $m = 9$ (i) $m = 10$

**Figure 3: Scatter plot of 20 assigned microservices produced by the FCM algorithm for each $m$.**

others (e.g., orange, black, and purple microservices). This observation underscores the substantial impact of $m$ on the granularity and clarity of microservice assignments.

In summary, our exploration highlights the critical role of the fuzziness coefficient $m$ in shaping the microservice assignment. The choice of $m = 2$ and the corresponding elbow point at 20 microservices serve as a pivotal configuration for striking an optimal balance between cohesion and separation in the decomposed system, showcasing the nuanced influence of hyperparameter choices on the resulting microservices.

## 5.5 Evaluation Metrics

We do not generate codes for the resulting microservices. Hence, we cannot run and evaluate it in the running time. Therefore, we only use metrics that evaluate the architecture of the microservices for comparison. The metrics are briefly described as follows:

- **Structural Modularity (SM):** This metric demonstrates how strong the connection among members in a microservice is, compared to the connection between the members and the classes outside the microservice. We can formulate this as follows:

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

  where $k$ is the number of clusters, $e_i$ is the number of edges inside cluster $i$, $n_i$ is the number of classes inside cluster $i$, and $e_{i,j}$ is the number of edges between cluster $i$ and cluster $j$. We aim to obtain a high SM for the resulting clusters.
- **Interface Number (IFN):** This metric directly counts the number of external connections of each microservice and determines the average of the external connections. Therefore, a good microservice should have a low IFN.

## 5.6 Results

The outcomes of our evaluation are visually depicted in Fig. 4 and 5, showcasing the Structural Modularity (SM) and Interface Number (IFN) scores, respectively. These figures provide a comprehensive insight into the relationship between maintainability thresholds, fuzziness coefficient ($m$) values, and the resulting SM and IFN metrics.

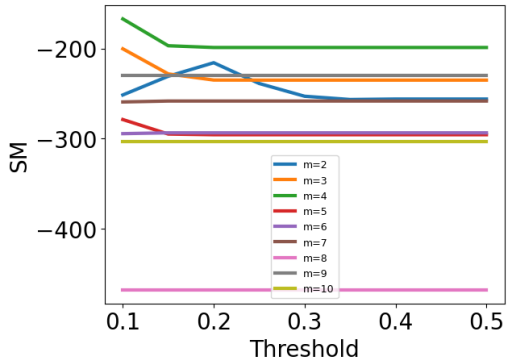



**Figure 4: SM scores for the resulting 20 microservices from the FCM algorithm with several maintainability thresholds and $m$ values**

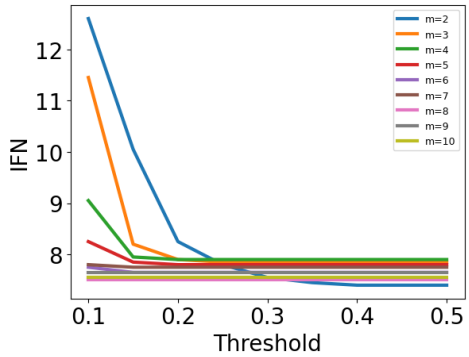



**Figure 5: IFN scores for the resulting 20 microservices from the FCM algorithm with several maintainability thresholds and $m$ values**

A crucial observation emerges as we scrutinize the impact of varying maintainability thresholds in conjunction with different $m$ values on the SM scores. Intriguingly, the relevance of the maintainability threshold diminishes as $m$ increases. This is perceptible in the graphical representation where the curve transforms into a linear trajectory around $m = 6$ for both SM and IFN scores. While the results with $m = 4$ exhibit superior performance in terms of SM scores, an interesting revelation surfaces concerning IFN scores. Specifically, results obtained with $m = 2$ and a maintainability threshold exceeding 0.35 outshine others in the IFN domain. This dynamic interplay between $m$ and the maintainability threshold underscores the nuanced considerations that users need to weigh based on their priorities and the metrics they prioritize.

The divergence in performance across SM and IFN metrics for different $m$ values accentuates the trade-offs inherent in the decomposition process. Users are confronted with the challenge of balancing the structural cohesion captured by SM with the efficiency of external connections reflected in IFN. The dependency on user priorities becomes paramount in determining the most suitable configuration, whether it be favoring superior SM scores

| Microservice 1 |
| --- |
| "UserRepository", "EmailConfig", "AuthUserRepository", "TrainServiceImpl", "PriceController", "JWTUtil", "PriceConfigRepository", "JsonUtils", "DeliveryRepository", "NotificationController", "ExecuteControlller", "FoodController", "RouteInfo", "JWTProvider", "AssuranceType", "Type", "ItemNameUtil", "VerifyCodeService", "TrainTicketApplication", "BaseException", "SecurityConfig", "CommonOrderAlterInfo", "ConsignController", "SecurityRepository", "Trigger", "InsidePayment", "NotifyRepository" |

**Figure 6: Example microservice 1 generated by our approach with 20 clusters and $m = 2$ that allow duplication and with the maintainability threshold of 0.25**

| Microservice 2 |
| --- |
| "TrainTicketApplicationTests", "VerifyCodeController", "TextMapUtil", "InfoConstant", "OrderSecurity", "AdminRouteController", "ConsignPriceController", "ExceptionUtils", "DocumentType", "Account", "FoodOrder", "AdminTrip", "AssuranceTypeBean", "RabbitSend", "Queues", "SwaggerConfig", "GetAccountByIdResult", "TrainService", "AuthUserDetailsServiceImpl", "UserOperationException" |

**Figure 7: Example microservice 2 generated by our approach with 20 clusters and $m = 2$ that allow duplication and with the maintainability threshold of 0.25**

with $m = 4$ or prioritizing optimal IFN results with $m = 2$ and a specified maintainability threshold.

In essence, our findings underscore the complexity of decision-making in microservice decomposition, where users must navigate the intricate interplay between maintainability thresholds and fuzziness coefficient values to align the outcomes with their specific objectives and metric preferences.

## 5.7 Recommended System Decomposition

Our approach recommended to decompose the system into 20 microservices. It was configured with an $m$ value of 2, and a maintainability threshold of 0.25. Figs. 6 and 7 depict two microservices generated through our approach. It is noteworthy that the classes featured in the microservice illustrated in Fig. 6, originally dispersed across 18 distinct microservices, have harmoniously converged into a cohesive new microservice.

Moreover, Fig. 7 showcases another instance of the effectiveness of our approach. That is, the approach efficiently consolidates classes from 14 different microservices into a singular, optimized entity, strategically enhancing both functionality and maintainability. This amalgamation is not arbitrary; it is a product of meticulous parameter tuning. The resulting microservice is not merely a container for diverse classes but a well-structured, cohesive unit. It skillfully intertwines functionalities from its constituent classes, potentially reducing inter-service communication overhead and streamlining operational flows within the emerging microservice.

## 5.8 Assessing Recommended Decomposition

To assess the feasibility of our approach and validate our recommendation, we followed the proposed decomposition to implement the new system versions. Our approach recommended microservice responsibilities expressed through suggested components (classes with annotations in the Spring framework). Throughout the transformation process, we identified that the original system exhibited

microservice anti-patterns where service components and data entities were inappropriately injected into various microservices through libraries. While such a mechanism could be aimed at encapsulation for reuse, it is known to be a bad practice in cloud-native [24] since such injection introduces dependencies across microservices and limits their independent evolution. Our algorithm's recommendations, implemented in the new version, effectively eliminated these injections, enhancing the architectural integrity. At the same time, there were limitations in the proposed decomposition. In particular, a small set of controllers overlapped because of the limited algorithm granularity. The original controllers' methods had low cohesion when assigned. These were identified as part of the algorithm's output but needed to be further refined in our implementation.

### 5.9 Resulting Implementation Assessment

We critically evaluated the transformation from the original 47-microservice train-ticket system, focusing on the 42 Java-based Spring Boot services, to a monolithic system version and subsequently to a more modular 20-microservice version. This transition, driven by our proposed machine learning algorithm, aimed to address modularity and key architectural concerns. The new system versions' source code is shared as an anonymous package[2].

Considering all system versions, the system intermediate representation was extracted and analyzed [4]. The service endpoints and the canonical data model matched across versions. Functional equivalence with the original system was verified using the available testing benchmark [20], with all tests successfully passed. This ensured that the new microservices maintained the same functionalities as the original.

At the same time, when we compare the source code analysis (SourceMonitor[3]) of the 20 and 42 microservice system versions, the 20 microservices have more lines of code and more classes in total, which is given by the elimination of the shared resources through a library anti-pattern and by our algorithm allowing class replication. The results on cyclomatic complexity are skewed, given the anti-pattern elimination. Yet, the design change to 20 microservices provides greater autonomy to individual microservice evolution, eliminating ripple effects and change propagation. This streamline updates and modifications that do not promote multiple modules, which makes it more efficient to adapt or change requirements. It is also more cohesive as the naming of the services and domain knowledge is logically organized (related functionalities in the same modules), enhancing the system's clarity and coherence.

The new system version shows better maintainability and evolvability with more independent microservices. At the same time, it must be considered from the context where a more thorough comparison and assessment needs to be performed, which is beyond the scope of this work.

Using dynamic analysis, our testing assessment considered CPU and Memory usage. The 42-microservice system version needed 16.83GB of RAM and 57.02% CPU to run whereas the 20-microservice version required 9.94 GB and 37.01% CPU. We conducted a total of 11 tests using Selenium, covering all user interfaces in the original repository. On average, the 42-microservice version took 4.07 minutes and the 20-microservice version took 3.2 minutes for test execution.

[2]The source code of the monolith, 20-microservice, and 42-microservice system version benchmark are available in https://zenodo.org/records/11215085; moreover, we share the screenshots of passing end-to-end tests benchmark and the user interfaces examples from running system in the package

[3]SourceMonitor https://www.derpaul.net/SourceMonitor

## 6 Conclusions

In the culmination of our research, we center our focus on the intricate process of transitioning from monolithic to microservices structures. The crux of our investigation lies in the adaptation and application of an extended machine learning methodology originally conceived for navigating the complexities of this transformative journey.

Within our study, we introduce a machine learning-based approach that harnesses the power of graph neural networks, variational autoencoders, and the FCM algorithm. Our exploration delves into the nuanced impact of hyperparameters, particularly the number of microservices and the fuzzy value $m$, on the performance of the resulting microservices. This elucidation underscores the critical role that thoughtful parameter tuning plays in achieving optimal outcomes.

The empirical demonstration of our approach's effectiveness is manifested through concrete examples showcasing the resultant microservices. Therefore, our exploration reveals the complexity of choosing hyperparameters, highlighting the tradeoffs involved. In contrast to previous efforts [21], our work provides users with more alternatives using fuzzy values. These examples provide concrete proof of our methodology's ability to intelligently group relevant classes into coherent microservices.

Our approach on an established system benchmark makes our work not only theoretical but also validated by actual system implementations that deploy, run, and pass tests. Such broad effort allowed us to perform realistic assessments and prove our approach's feasibility and benefits. Moreover, it must be recognized that our work builds cross-domain bridges and brings significant advancement to the interdisciplinary community in the form of microservice system benchmarks that a) invite ML experts to further improve system decomposition to optimize various system qualities using the system monolith version as input and the original system as an assessment baseline, and b) invite software engineers to further validate qualities of provided system benchmark versions.

Looking forward, future avenues of research beckon us to expand the scope of information incorporated into our machine learning algorithms. A prospective enhancement involves integrating the actual codes of the classes into the input of machine learning models, such as Transformers. This augmentation empowers machine learning models to make more informed decisions, facilitating the grouping of classes into microservices with a heightened level of accuracy and relevance.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. OISE-1854049, OISE-2409933.

## References


[1] Muhammad Abdullah, Waheed Iqbal, and Abdelkarim Erradi. 2019. Unsupervised Learning Approach for Web Application Auto-Decomposition into Microservices. https://doi.org/10.1016/j.jss.2019.02.031

[2] Florian Auer, Valentina Lenarduzzi, Michael Felderer, and Davide Taibi. 2021. From monolithic systems to Microservices: An assessment framework. *Information and Software Technology* 137 (2021), 106600. https://doi.org/10.1016/j.infsof.2021.106600

[3] James C Bezdek, Robert Ehrlich, and William Full. 1984. FCM: The fuzzy c-means clustering algorithm. *Computers & geosciences* 10, 2-3 (1984), 191–203.

[4] Vincent Bushong, Diptal Das, and Tomas Cerny. 2022. Reconstructing the Holistic Architecture of Microservice Systems using Static Analysis. In *Proceedings of the 12th International Conference on Cloud Computing and Services Science - CLOSER*. 149–157.

[5] Rui Chen, Shanshan Li, and Zheng Li. 2017. From Monolith to Microservices: A Dataflow-Driven Approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 466–475. https://doi.org/10.1109/APSEC.2017.53

[6] Md Showkat Hossain Chy, Korn Sooksatra, Jorge Yero, and Tom Černý. 2024. Benchmarking Micro2Micro transformation: an approach with GNN and VAE. *Cluster Computing* (05 2024). https://doi.org/10.1007/s10586-024-04526-z

[7] Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth Tamilselvam. 2021. Graph Neural Network to Dilute Outliers for Refactoring Monolith Application. *CoRR* abs/2102.03827 (2021). arXiv:2102.03827 https://arxiv.org/abs/2102.03827

[8] Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth Tamilselvam. 2021. Graph neural network to dilute outliers for refactoring monolith application. In *Proceedings of 35th AAAI Conference on Artificial Intelligence (AAAI'21)*.

[9] Sinan Eski and Feza Buzluca. 2018. An Automatic Extraction Approach: Transition to Microservices Architecture from Monolithic Application *(XP '18)*. Association for Computing Machinery, New York, NY, USA, Article 25, 6 pages. https://doi.org/10.1145/3234152.3234195

[10] Gianluca Filippone, Nadeem Qaisar Mehmood, Marco Autili, Fabrizio Rossi, and Massimo Tivoli. 2023. From monolithic to microservice architecture: an automated approach based on graph clustering and combinatorial optimization. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. 47–57. https://doi.org/10.1109/ICSA56044.2023.00013

[11] Robert Grandl. 2023. A Quick Introduction to Service Weaver.

[12] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. 2021. Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices. *CoRR* abs/2107.09698 (2021). arXiv:2107.09698 https://arxiv.org/abs/2107.09698

[13] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. 2018. *Challenges When Moving from Monolith to Microservice Architecture*. 32–47. https://doi.org/10.1007/978-3-319-74433-9_3

[14] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[15] Alexander Krause, Christian Zirkelbach, Wilhelm Hasselbring, Stephan Lenga, and Dan Kröger. 2020. Microservice Decomposition via Static and Dynamic Analysis of the Monolith. *CoRR* abs/2003.02603 (2020). arXiv:2003.02603 https://arxiv.org/abs/2003.02603

[16] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. 2022. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering* 27 (01 2022). https://doi.org/10.1007/s10664-021-10063-9

[17] A. Mathai, S. Bandyopadhyay, U. Desai, and S. Tamilselvam. 2021. Monolith to Microservices: Representing Application Software through Heterogeneous Graph Neural Network. *arXiv preprint* arXiv:2112.01317 (2021).

[18] Davide Rahman, MI.and Taibi. 2019. A curated Dataset of Microservices-Based Systems. In *Joint Proceedings of the Summer School on Software Maintenance and Evolution* (Tampere, Finland). CEUR-WS.

[19] C. Richardson. 2018. *Microservices Patterns: With examples in Java*. Manning. https://books.google.com/books?id=UeK1swEACAAJ

[20] Sheldon Smith, Ethan Robinson, Timmy Frederiksen, Trae Stevens, Tomas Cerny, Miroslav Bures, and Davide Taibi. 2023. Benchmarks for End-to-End Microservices Testing. In *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 60–66. https://doi.org/10.1109/SOSE58276.2023.00013

[21] Korn Sooksatra, Rokin Maharjan, and Tomas Cerny. 2022. Monolith to Microservices: VAE-Based GNN Approach with Duplication Consideration. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 1–10.

[22] Davide Taibi and Kari Systä. 2019. From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining. https://doi.org/10.5220/0007755901530164

[23] Imen Trabelsi, Manel Abdellatif, Abdalgader Abubaker, Naouel Moha, Sébastien Mosser, Samira Ebrahimi-Kahou, and Yann-Gaël Guéhéneuc. 2022. From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis. *Journal of Software: Evolution and Process* 35 (09 2022). https://doi.org/10.1002/smr.2503

[24] Adam Wiggins. 2017. The Twelve-Factor App. https://12factor.net/ https://12factor.net/, last accessed 1/2/2022.

[25] Rahul Yedida, Rahul Krishna, Anup Kalia, Tim Menzies, Jin Xiao, and Maja Vukovic. 2021. Partitioning Cloud-based Microservices (via Deep Learning).

[26] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Wenhai Li, Chao Ji, and Dan Ding. 2018. Delta debugging microservice systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 802–807. https://doi.org/10.1145/3238147.3240730

[27] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking Microservice Systems for Software Engineering Research *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 323–324. https://doi.org/10.1145/3183440.3194991