Migration to Microservices: A Comparative Study of Decomposition Strategies and Analysis Metrics

Meryam Chaieb
Laval University
Quebec, QC, Canada
meryam.chaieb.1@ulaval.ca

Mohamed Aymen Saied
Laval University
Quebec, QC, Canada
mohamed-aymen.saied@ift.ulaval.ca

***Abstract*—The microservices architectural style is widely favored for its scalability, reusability, and easy maintainability, prompting increased adoption by developers. However, transitioning from a monolithic to a microservices-based architecture is intricate and costly. In response, we present a novel method utilizing clustering to identify potential microservices in a given monolithic application. Our approach employs a density-based clustering algorithm considering static analysis, structural, and semantic relationships between classes, ensuring a functionally and contextually coherent partitioning. To assess the reliability of our microservice suggestion approach, we conducted an in-depth analysis of hyperparameter sensitivity and compared it with two established clustering algorithms. A comprehensive comparative analysis involved seven applications, evaluating against six baselines, utilizing a dataset of four open-source Java projects. Metrics assessed the quality of generated microservices. Furthermore, we meticulously compared our suggested microservices with manually identified ones in three microservices-based applications. This comparison provided a nuanced understanding of our approach's efficacy and reliability. Our methodology demonstrated promising outcomes, showcasing remarkable effectiveness and commendable stability.**

Additional technical details are available in the replication packages [1]

Keywords-microservices architecture; static analysis; clustering; decomposition.

I. Introduction

The monolithic architectures is one of the most widely utilized architectures for software design. In the realm of software architecture, the monolithic architecture stands as a prominent approach where an application is built as a single, indivisible unit. It encompasses all essential functionalities and components within a unified codebase, thereby presenting a tightly coupled system. This architectural style often involves a centralized database, user interface and business logic, rendering it self-contained and independent of external services. An exemplar of monolithic architecture, that we will use later in our evaluation process, can be observed in the context of the DayTrader [2] application, a virtual stock trading platform. In this monolithic setup, all trading functionalities, user management, and financial calculations are contained within a single application. While this approach simplifies development and deployment and despite being used since the early days of software systems, it can pose challenges when it comes to scalability, maintaining code integrity and accommodating changes or updates in individual components [1] [2] [3]. Monolithic architecture, in general, tends to experience performance issues when the amount of users exceeds a certain capacity level of these monolithic applications [1].

Many methods have arisen throughout time to solve these performance difficulties, such as migrating to new technologies, managing independent services and deploying more powerful servers. However, monolithic applications have evolved into massive, complex and often inefficient software systems over time, making them challenging to maintain. Additionally, they may not be able to support newer and more sophisticated technology [4] [5] [6] [7] [8] [9]. Moreover, adapting these systems to meet rising user demand is either unfeasible or necessitates wasteful workarounds such as the duplication of the whole monolith [10].

Microservices architecture, in the other side, is gaining in popularity and is projected to play a large role in developing scalable, easy to maintain software products by focusing on tightly defined, separated services inside a distributed system [1]. The microservice architecture emerges as a contemporary approach where an application is built as a collection of small, independent services. These services are designed to be modular, self-contained and focused on specific business functionalities. Unlike the monolithic architecture, microservices operate as autonomous units that communicate with each other through well-defined APIs. This architectural style enables teams to develop, deploy and scale individual services independently, fostering flexibility and maintainability. For these reasons, numerous firms have sought to rework their monolithic apps into a microservice based version choosing a viable option by altering these systems while preserving the same functionality [11] [12]. A noteworthy example of the microservice architecture can be found in the Netflix streaming platform. In this setup, various microservices handle distinct tasks such as user authentication, content recommendation, billing and media streaming. Each microservice can be developed, tested, deployed and scaled independently, allowing Netflix to rapidly innovate, adapt to changing demands and deliver a seamless streaming experience to its vast user base [13].

The transition from a monolithic design to a more durable

[1]https://anonymous.4open.science/r/Migration-to-microservicess-B67F/README.md

[2]https://github.com/WASdev/sample.daytrader7

and robust microservice architecture is based on the idea of finding contextually and functionally relevant modules and encapsulating them in a single service, while ensuring strong cohesion and low coupling between them. As Rosati pointed out in their research on the migration cost [14], transforming a mature monolithic software into microservices architecture may demand substantial investment in terms of time and cost. These difficulties have prompted academics to devise automatic decomposition methods that might ease the migration process. Such decomposition approaches seek to discover service boundaries and dependencies more efficiently and quickly, enabling for an easier transition [15]. The task of transitioning a monolithic application into a microservices architecture is treated as a clustering problem in the context of our project. Our suggested method entails a multi-step procedure that employs static examination of the source code to determine the functional and contextual links between its classes. This stage is crucial for detecting relationships between classes and identifying potential service boundaries. Upon identifying the functional connections, we employ an adaptive density-based clustering technique known as adapted-BMSC to partition the classes into several prospective microservices. This selection is motivated by its superior performance in similar clustering tasks, outperforming other state-of-the-art algorithms. These resultant clusters represent potential microservices for further evaluation.

We conducted an in-depth review utilizing a variety of metrics to measure the efficacy and efficiency of our method. We specifically evaluated the extracted microservices' quality to that of six other well-known decomposition baselines. To achieve a thorough review, we applied the evaluation criteria from several perspectives and evaluated them on seven example applications of diverse complexity.

We conducted an extensive comparison with two widely-used clustering techniques that solve this challenge. We compared the performance of our technique to existing algorithms using various settings of hyperparameters.

On the other side , in addtition, we also analyzed the efficiency of our technique by comparing the resultant microservices to those built by human specialists. This allowed us to assess how far our technique might automate the decomposition process while preserving the quality of human-designed microservices. As a culmination of these diligent efforts, our methodology has yielded a series of encouraging and promising outcomes as well as areas where more refinement and optimisation may be necessary.

The main contributions of our work are as follows:

1) The proposed approach combines density-based clustering and static analysis techniques to leverage the advantages of both methods. It considers the structural and semantic dependencies among classes in a given monolithic application.
2) A comparison between the resulting decomposition of the proposed algorithm and those of commonly used clustering algorithms in the field.
3) A comparison between the microservices produced through our proposed approach and those that were manually identified by human experts.

This paper is structured as follows: Section II presents the related work in the field of monolithic migration to microservices. Section III details the proposed methodology, including the clustering algorithms used. In Section IV, we discuss the findings of this effort and respond to different research questions. Section V outlines the threats to validity that were considered during the study. Finally, Section VI, concludes the work and discusses future research directions.

II. Related Work

Software engineering research has been tackling various issues in different phases of the software development lifecycle. However, the fast pace of evolution in the IT industry and the staggering growth of new technologies [3] based on APIs [12], [16], [17], containers [18], microservices [19]–[22], cloud and virtualization, put an increasing pressure on software development [2] and deployment [5], [9] practice to fully exploit this paradigm shift. This led to constant questioning of existing techniques [16] and results of software engineering research [23], [24], leading to investigating the use of AI and ML-based techniques to solve software engineering problems in topics related to software reuse [25], recommendation systems [26], mining software repositories [23], software data analytics and patterns mining [6], [7], [27], [28] , program analysis and visualization [8], [29], testing in the cloud environment, Edge-Enabled systems [30], microservices architecture [31] and mobile applications.

The majority of methods in the literature that address the microservices extraction problem may be divided into two major components.

The first component is concerned with the type of input provided to the solution and how it is handled. The methods suggested by MSExtractor [32], Bunch [33], and [34] , for example, take as input the source code of a monolithic program and apply various static analysis techniques to it. MSExtractor and Bunch, in particular, construct call graphs that encode the relationships between the classes in these systems. On the other hand, the approach described in [34] turns the source code into a collection of Abstract Syntax Trees, which are then fed into a code embedding model [35]. Static analysis, and more specifically the source code, is used with the assumption that structurally comparable classes or functions should be grouped together.

Some approaches have extracted the semantic relationships between the monolith's components from the source code. For example, the approach called HierDecomp [36] proposetwo types of measures : the structural similarity synthesised from the static calls between application's classes and the semantic similarity generated fromthe code text analysis. Brito and al. [37] identify the systems' topics, based on topic modeling techniques, which correlate to domain terms and reflect the legacy system's microservices. The collection of lexical information in the source code, notably method declarations,

variables, method and class names.., is used to infer such topic models.

Other approaches, such as Mono2Micro [38], FoSCI [39], and COGCN [40], are based on the study of monolithic system use cases and execution traces. These solutions try to bring together classes or methods that interact at run-time for each business need given as input. For example, based on the execution traces, Mono2Micro computes similarity metrics across classes.

These analytic approaches are not mutually exclusive and can be used to provide improved results. CO-GCN [40], for example, which, in addition to collecting execution traces, builds its model's architecture using the source code of the input application, assuming that each microservices contains classes with comparable domain concepts. Sellami and al [41] combine both static and dynamic analysis in order to cover the individual disadvantages of each of the analysis approaches.

There are, on either side, systems that employ different inputs, such as MEM [42], which analyses the git commit history of monolithic programs. This technique generates a graph from the git history that encodes the class similarity. Adding to that, Service Cutter [43] is a migration tool that uses the Unified Modeling Language (UML) diagrams to describe the various components of the monolithic application to be fragmented. The main drawback of this method is that the majority of existing programs lack representative diagrams, therefore it is up to the user to perform reverse engineering techniques to produce them and then transform them into the right format allowed by this approach. Nonetheless, Dehghan and al. [44] extended Service Cutter where the user will be required to provide the needed representative models as well as the source code to two distinct mechanisms: Service cutter [43] and MoDisco (Model Driven Reverse Engineering Framework) [45].

The second component of each approach takes the data processed in the previous step as input and applies an algorithm to it in order to build the decomposition. Most methods utilize clustering algorithms, such as [34] which feeds vectors derived from code embedding into an Affinity propagation clustering process [46]. The similarity metrics computed by an agglomerative single-linkage clustering method [47] are used by Mono2Micro [38]. Based on the graph it developed, MEM [42] provides its own clustering mechanism. Based on the similarity metrics, HierDecomp [36] and HyDecomp [41] employ a DBSCAN [48] density based clustering algorithm which ends by having a hierarchical microservices decomposition recommendation. Some methods suggest search algorithms to accomplish their goal. On the execution traces, MSExtractor [22] use the non-dominated sorting genetic algorithm (NSGA-II) [49] whereas FoSCI [39] employs both NSGA-II and hierarchical clustering. Bunch [33], on the other hand, employs a hill-climbing algorithm. A community discovery method is used by Service Cutter to provide a decomposition recommendation.

III. Proposed Approach

The task of extracting microservices from a monolithic application is approached as a clustering problem, with the application's source code as input. Figure 1 depicts our approach in detail, outlining the phases involved in our research. Our primary goal in this effort is to achieve granularity at the class level.

Our methodology initiates by extracting both semantic and structural information from the application through static analysis of the legacy application's source code. This initial phase is followed by a preprocessing step where we systematically assess all possible combinations, opting for specific choices from both semantic and structural preprocessing components. Subsequently, the combined representations generated in the previous step are fed into the first clustering subtask, where we employ the Mean Shift algorithm. The centers of density identified through Mean Shift are then utilized to compute a novel distance metric termed "iModes similarity." This newly derived metric is subsequently fed into the final clustering subtask, which is executed by the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm. The ultimate outcome of our approach is the resulting decomposition, achieved through the mapping of clusters initially detected by the mean shift algorithm and the final clusters generated by the DBSCAN algorithm. It is important to highlight that our methodology draws inspiration from the existing Boosted Mean Shift Clustering algorithm (BMSC), albeit with modifications to the calculation of iModes and adjustments in the input for the DBSCAN algorithm.

We have separated this section into two subsections to enhance clarity and comprehensibility, each presenting a unique viewpoint on our methodology. We will start by defining the problem we want to solve and outlining how we represent the monolithic system in the preprocessing step, as shown in Figure 1. We will next move to the modeling step, where we will explain the primary clustering algorithms we employ.

A. Representation of the Monolithic Application

The input to our microservices extraction solution is a monolithic application, which is characterized as a set of Object Oriented Programming classes denoted as C_M=(c_1, c_2,.., c_N), where N represents the total number of classes in the application. In this context, our approach aims to partition the original monolithic application into a set of K microservices, with the output being a collection of microservices, M = (m_1, m_2,.., m_K). Each microservice, m_i, represents a subset of the original classes and is defined as m_i=(c_a, c_b,.., c_p), where c_j is the OOP class that constitute the microservice. By applying our approach, we aim to optimize the decomposition of the monolithic application into microservices, where each microservice is expected to be cohesive and loosely coupled, resulting in a more maintainable and scalable architecture.

The initial stage of our suggested solution focuses on representing the monolithic application and extracting the necessary information to build the microservices suggestions. To do this, we begin by creating an encoding scheme for each

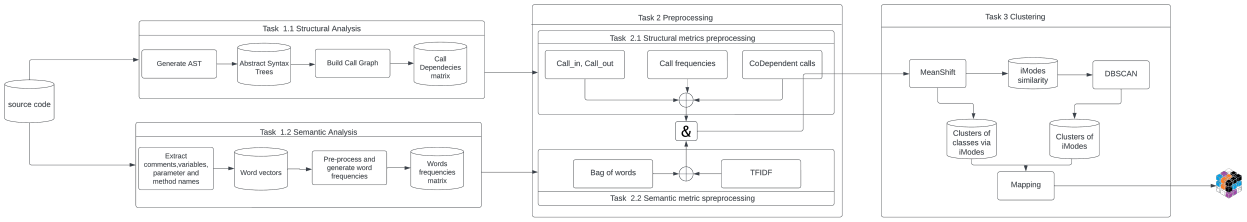

Fig. 1. Overview of the Microservices Extraction Process.

of the monolith's classes. The goal of our encoding approach is to capture the structural and semantic relationships that exist between the classes in the monolithic system as described in the source code. We want to find important links between the classes by concentrating on these dependencies, which will allow us to produce more accurate and effective microservices suggestions. This encoding phase is crucial because it serves as the foundation for the following stages of our strategy, in which we use clustering techniques to discover groupings of similar classes that may be encapsulated within individual microservices.

1) Structural encoding: Abstract Syntax Trees (ASTs) can be created after the source code has been evaluated using a static analysis tool, such as "Understand" [9]. These ASTs help us comprehend the code structure and may be used to extract call relationships between classes. This is accomplished by creating call graphs that depict the relationships between distinct classes and how they interact with one another via function calls. We can establish which classes are commonly called together and uncover the most significant relationships in the codebase by studying these call graphs.

As described in the diagram of figure 1 the structural information will be encodes using three different options:

- $Call_{in}$, $Call_{out}$: Each class in the monolithic application is encoded based on the dependency matrix derived from the static analysis phase. Specifically, we compute the sum of incoming and outgoing calls for each class. Our rationale for this encoding scheme is rooted in the observation that classes that exhibit frequent outgoing calls also tend to be called multiple times by other classes. By leveraging this insight, our approach seeks to group classes that interact frequently within the same cluster. This clustering strategy aims to reduce coupling between clusters while promoting greater cohesion within the resulting microservices
- Call frequencies: In contrast to the prior alternative, our second strategy tries to build more coherent clusters by encoding classes in greater depth. We analyse the frequency of calls between each pair of classes rather than just adding incoming and outgoing calls. By doing so, we hope to capture a more nuanced understanding of class connections, resulting in more coherent clusters.
- CoDependent calls: In this third and final structural encoding option, we take a more detailed approach to encode classes by considering the frequency of calls of classes that called both classes to encode each pair of classes, rather than just focusing on direct calls. This approach aims to group together classes that were involved in the same use case which will lead to produce even more cohesive clusters.
  To aid in understanding this concept, we provide an example in Figure 2.

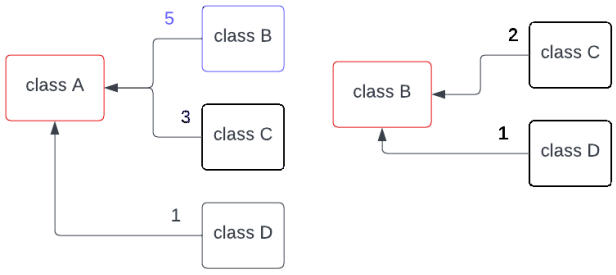

Fig. 2. Illustrative example of CoDependent calls metric.

In this scenario, there are four classes: A, B, C, and D. The objective is to encode the relationship between classes A and B. Figure 2 illustrates that class A is invoked five times by class B, three times by class C, and once by class D. In addition, class B is invoked twice by class C and once by class D. To encode the pair of classes A and B, the frequencies of calls originating from classes that invoked both A and B are summed. The encoding of the pair of classes A and B is the sum of incoming calls to A from the codependent classes C and D. The basic idea behind this technique is that classes that are frequently called together are usually used to handle the same functionality, and hence are required for the same use case. As a result, the goal is to create robust microservices that can solve a specific use case

each microservice.

2) Semantic encoding: Assume we are dealing with monolithic software projects that were created in accordance with industry norms. The names of classes, methods and variables are chosen based on functional principles in such projects and thorough annotations are included to indicate their intended use. As a result, the vocabulary employed in each software component can provide useful insights about the class's underlying domain, meaning and functionalities. It is critical to examine the semantic information associated with the classes when decomposing a monolithic program into microservices. This source of information gives a more in-depth insight of the underlying concepts and class relationships in the legacy system. We can ensure that the resulting microservices are resilient and coherent by integrating this knowledge.

By including semantic information into the encoding process, we can determine the essential links between classes and the functionality they provide, facilitating the ability to combine them into coherent and self-contained microservices. This guarantees that each microservice serves a specific use case. Finally, this method results in a more modular and scalable system.

As a result, the semantic information of each class is composed of a collection of terms that are used in different parts such as comments, parameter names, field names, method names, and variable names. To preprocess these words, we separate them using CamelCase, filter out stop words and normalise them using stemming. This method guarantees that we have a good comprehension of the class and its functionalities.

As seen in Figure 1, the semantic information received from this preprocessing phase will be represented in two options:

- Bag of Words (BoW) : One option for class encoding involves incorporating the frequencies of terms found within the vocabulary of the application. By doing so, we can ensure that the terms with higher frequencies are more closely related to the domain of the class. By considering the frequency of terms within each class, we can create a more refined understanding of the class and its intended purpose. This information can be used to group similar classes together into cohesive microservices, improving the overall organization and functionality of the resulting software system.
- Term Frequency-Inverse Document Frequency (TF-IDF): Utilizing TF-IDF instead of simple Bag of Words can improve class clustering in a variety of ways. To begin, TF-IDF considers not only the frequency of a word in a specific class, but also the inverse document frequency, which assesses how unique a term is to a class in comparison to the total corpus of classes. This means that unusual and unique terms that are exclusive to a class will have a larger weight in the TF-IDF calculation and will be more informative of the domain and purpose of the class. Second, utilising TF-IDF can help limit the influence of common keywords that are not specific to any single class, such as other generic terms, which can distort clustering results when Bag of Words are implemented. Overall, by encoding the classes using TF-IDF, the resultant feature vectors will be more representational of the classes' distinct properties, resulting in more accurate and effective clustering findings.

B. Clustering algorithms

The objective is to extract microservices by encoding classes structurally and semantically using different combinations of options. To achieve this, the Adapted Boosted Mean Shift Clustering (BMSC) [50] algorithm, along with other well-known clustering algorithms such as Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [51] and Mean Shift [52], is experimented with. The goal is to compare the abilities of adapted-BMSC against those of the other two algorithms. In the following section, we will provide a detailed description of each algorithm.

1) DBSCAN algorithm: DBSCAN is a clustering technique used in spatial databases to detect clusters and noise. The user must specify two hyperparameters, Eps and MinPts. The method uses these parameters to arrange densely related points into a single cluster. One major benefit of DBSCAN is that the user doesn't need to define the number of clusters. Alternatively, based on the data and the provided hyperparameters, the number of clusters can be arbitrary detected , leading in more accurate clusters [48].

Hyperparameters :

- **Eps (ϵ) :** refers to the radius of the neighbourhood surrounding the cluster's central point. This parameter determines the densest area in the data collection.
- **MinPts :** This is the bare minimum of points required to build a cluster.

The value of the Eps parameter in DBSCAN can effect the number of dense clusters and the number of identified clusters. A higher Eps value, in particular, might result in fewer dense clusters being detected, which can reduce the overall number of clusters identified by the method. While implementing DBSCAN, it is important to avoid setting the MinPts parameter too low, as it is with the Eps parameter. A low MinPts value may cause the algorithm to generate an excessive number of less dense clusters.

After executing the DBSCAN algorithm on a dataset, the results may be classified into three classes of points, as illustrated in Figure 3. A core point is one that has at least MinPts number of points within an Eps radius. A Border point is any point that is close to Eps and possesses one or more Core points. Lastly, a Noise point is any point that is neither Core nor Boundary.

To build clusters, the DBSCAN algorithm goes through numerous phases. It begins by picking an arbitrary point in the database to serve as the first Core point. It then collects data points within a distance equal to Eps. A cluster is produced if the total number of points acquired is more than or equal to the minimum number of points necessary (MinPts). To enlarge the original cluster, this procedure is repeated for each cluster point. During this step, the algorithm creates the first cluster. The procedure is then repeated after removing all of the points

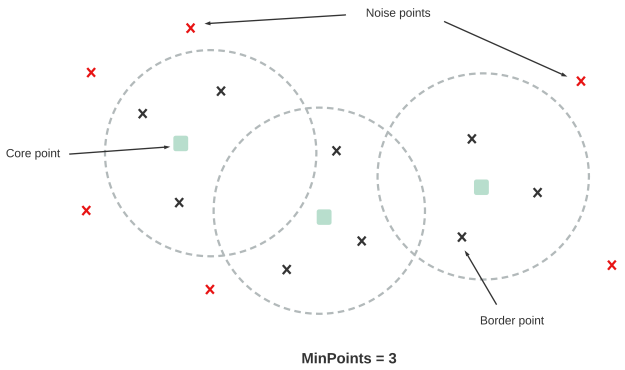

Fig. 3. DBSCAN algorithm showcase.

that composed it from the database. When no further clusters can be produced with the provided parameters, the algorithm stops. The rest of the points are labelled as Noise.

Despite the fact that DBSCAN is widely used in our problem, it still has limitations. As demonstared in previous studies on extracting microservices from monolithic software relied on DBSCAN, this algorithm is highly sensitive to its hyperparameters, leading to significant variation in microservices' quality. Moreover, DBSCAN-based approaches may not work well with datasets with varying densities or non-globular shapes.

2) Mean Shift: The Mean Shift method is a cluster analysis technique that does not need any assumptions about the underlying distribution of the data. Based on the data, it can automatically detect non-linearly formed clusters and compute the number of clusters [52].

Figure 4 depicts a succession of steps taken by the Mean Shift algorithm to find clusters. It begins by identifying a region of interest, which is indicated by the red area in the picture. The centre of density or centre of mass for that region, shown by the blue point, is then calculated. The mean shift vector is then generated, and the centre of the area is shifted along the vector until it corresponds with the centre of mass point. This procedure is performed until convergence is reached. Members of the same group are points that converge to the same center of mass.

Although the Mean Shift method has shown excellent results, [50] demonstrates that adapted-BMSC outperformed Mean Shift in a similar clustering problem with more stable clustering.

Given the difficulties involved with clustering algorithms in situations where there is no obvious separation between clusters or where the number of clusters is uncertain, we decided to investigate alternatives to standard techniques. We picked the Boosted Mean Shift Clustering (BMSC) technique since it has showed higher performance in similar situations, and we compared the results of this adapted clustering algorithm to those obtained using other algorithms, including some that employ DBSCAN.

Algorithm 1 adapted-Boosted Mean Shift Clustering

Require: X, width, height, Eps.
Ensure: the final clustering results cl_final.

```
Initialize Grid( X,width,height)
                    ▷ Distribute X over G = width × height cells.
iModes ← ∅
counter ← 1
while counter! = 3 do
    for j ← 1 to G do
        newiModes ← MeanShift(cellData_i)
        iModes.Append(newiModes)
                    ▷ collect the iModes of each cell of the Grid
    end for
    ConfidenceAssignement(Semantic_similarity)
                ▷ Assign confidence values to classes in each cell

    for j ← 1 to G do
        CollectedData ← CollectNeighborhoodData(j, neighborhood_structure) ∪ cellData_j
        cellData_j ← WeightedSampling(CollectedData)
                                                ▷ update cellData_j
    end for
        cl_iModes, numberOfClusters ← DBSCAN (iModes_similarity, Eps)
                ▷ cl_iModes is the clustering results of the iModes
    if numberOfClusters == lastnumberOfClusters then
        counter++
    else
        counter ← 1
    end if
end while
cl_final ← DataAssignement(X,cl_iModes)
```

3) Adapted-BMSC algorithm: The adapted-Boosted Mean Shift Clustering algorithm is a hybrid clustering technique that combines two well-known clustering techniques: Mean Shift and DBSCAN. It is a density-based clustering methodology that overcomes some of the limitations of both approaches and can find clusters of any form and size with varied densities without the need for a predetermined number of clusters. [50]

The adapted-BMSC first applies the Mean Shift algorithm on the dataset to generate a set of initial cluster centres. The centres of these clusters are used as input for the subsequent steps, which implements the DBSCAN algorithm. Adapted-BMSC selects a sample of the data that captures the skeleton of the clusters in order to properly identify the data's underlying structure. Essentially, the goal of adapted-BMSC is to overcome the limits of individual clustering algorithms by combining the capabilities of Mean Shift and DBSCAN, resulting in a more powerful and accurate clustering approach.

Algorithm 1 outlines the steps involved in applying the adapted-BMSC algorithm. The first step is to divide the data uniformly into cells of a grid, where the grid size is specified by the user. Once the grid is initialized, the Mean Shift algorithm is applied independently to the data in each cell, as shown in Figure 4. This produces a list of intermediate mode points (iModes) for each cell. The next step in the adapted-BMSC algorithm is to disperse the data of each cell using a specific mechanism. This re-sampling mechanism involves each grid cell interacting with a limited number of cells in

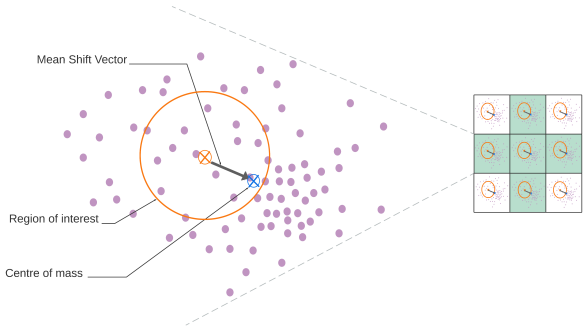

Fig. 4. Mean Shift Algorithm showcase.

its vicinity, which are defined as its neighborhood based on a previously determined neighborhood structure. The BMSC paper [50] presents various neighborhood structures, which are depicted in Figure 5. In our work, we adopt the linear 5 neighborhood structure.

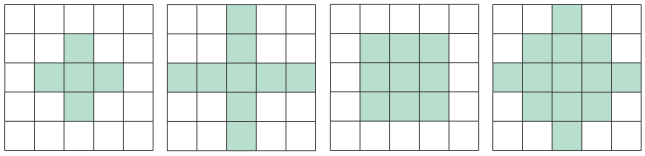

Fig. 5. Potential neighbourhood structures.

Upon completing the initial step of re-sampling, the subsequent stage of the adapted-BMSC algorithm involves calculating the distances between all data points in the parent cell and those in its neighboring cells, relative to the intermediate modes (iModes) generated by the Mean Shift algorithm. To determine the similarity between the class and its corresponding iMode, we adopt a semantic similarity metric that assesses the confidence level of each relationship. By incorporating this metric, we can effectively identify the semantic association between the two points during this preliminary stage. After that, adapted-BMSC algorithm utilizes the list of intermediate modes (iModes) obtained from the Mean Shift algorithm to run DBSCAN in order to identify clusters of densely packed iModes, which in turn generates clusters of the original data points at a lower level.
In our particular scenario, we utilize an aggregation function to transform the iModes produced by the Mean Shift algorithm into a format similar to that of the legacy application's classes. More specifically, we represent each cluster center by summing the structural encodings of the classes assigned to its cluster, thus capturing the structural aspect of the mode point. Additionally, we compute the semantic part of the vector by summing the term frequencies of words used in those specific classes.

For the purpose of extracting reliable microservices, we adopt a novel approach inspired from the work of Sellami and al [36] where we don't directly input the encoders of iModes into the DBSCAN algorithm. Instead, we provide the connections between each pair of iModes. To achieve this, we employ the iModes similarity measures that capture the structural and semantic relationships between the iModes. This approach aims to produce microservices that are consistent from both the implementation and use cases perspectives.
The iModes similarity is calculated as follows :

- **iModes Similarity (MS) :** It is a weighted sum of two similarity metrics, as provided by equation 1.

$$MS(m_i, m_j) = \alpha Sim_{str}(m_i, m_j) + \beta Sim_{sem}(m_i, m_j) \quad (1)$$

With :

- $\alpha \in$ *[0,1]*,
- $\beta \in$ *[0,1]*,
- $\alpha + \beta = 1$.

Each one of the similarities is computed as follow:

- **Structural similarity** (Sim_{str}) : We calculate this measure based on the number of method calls that are common between two iModes. This allows us to encode their level of interdependence and evaluate their similarity from a functional perspective.
  The structural similarity of two given iModes m_i and m_j is determined using equation (2):

$$sim_{str}(m_i, m_j) = \begin{cases} \frac{1}{2}(\frac{call(m_i,m_j)}{call_{in}(m_j)} + \frac{call(m_i,m_j)}{call_{in}(m_i)}) & If call_{in}(m_i) \neq 0 and call_{in}(m_j) \neq 0 \\ \frac{call(m_i,m_j)}{call_{in}(m_j)} & If call_{in}(m_i) = 0 and call_{in}(m_j) \neq 0 \\ \frac{call(m_i,m_j)}{call_{in}(m_i)} & If call_{in}(m_i) \neq 0 and call_{in}(m_j) = 0 \end{cases} \quad (2)$$

With:

- call(m_i , m_j): refers to the number of times that method m_i has called method m_j;
- $call_{in}(m_i)$: refers to the number of incoming calls in m_i.

The structure similarity values range between 0 and 1, with 1 denoting that the iModes m_i and m_j are highly similar in functionality, and 0 indicating complete independence between the two.

- **Semantic Similarity** (Sim_{sem})**:** In order to evaluate the similarity between the domain semantics of two iModes,a TF-IDF model is used, as in equation 3. The semantic similarity metric between two classes is represented by the cosine similarity between their respective vectors [53].

$$sim_{sem}(m_i, m_j) = \frac{\vec{m_i}.\vec{m_j}}{||\vec{m_i}||.||\vec{m_j}||} \quad (3)$$

With:

- $\vec{m_i}$: represents the TF-IDF vector of iMode m_i.
- $||\vec{m_i}||$: represents the Euclidean norm of the vector $\vec{m_i}$.

The values of Sim_{sem} range from 0 to 1, with 1 indicating that both classes employ the same vocabulary and so fulfil the same use case.

Afterwards, we employ DBSCAN algorithm on the iModes similarity metric obtained from the previous step. The clustering process is iterated until DBSCAN algorithm produces the same number of clusters for three consecutive iterations.

Because of its unique combination of Mean Shift and DBSCAN, the adapted-BMSC method has outperformed conventional clustering approaches. This combination keeps both algorithms' non-parametric character, resulting in more robust clustering.

IV. Evaluation

This section provides a summary of the evaluation process of our approach in discovering suitable microservices, which are available in the replication packages [3]. Table I summarizes the characteristics of the monolithic applications used to assess different aspects of our approach.

Project	Version	SLOC	# of classes
Plants [4]	1.0	7,347	40
DayTrader [5]	1.4	18,224	118
JPetStore [6]	1.0	3,341	73
AcmeAir [7]	1.2	8,899	86

TABLE I
- Characteristics of monolithic applications

A. Research Questions

The goal of our experimental investigation is to address a set of research questions (RQs):

RQ1: What is the most effective and promising configuration among the various choices in our approach that leads to favorable outcomes?

RQ2: How does the stability and robustness of our approach compare to that of Mean Shift and DBSCAN in relation to hyperparameter variation?

RQ3: How does our solutions perform in terms of partitioning quality when compared with state-of-the-art baselines?

RQ4: How well do the extracted microservices compared to those that were manually identified by software engineers?

B. Evaluation metrics

We used a set of metrics specified in [38] to analyse various aspects of the extracted microservices without relying on the ground truth microservices:

- **Structural Modularity (SM) :** Determined by measuring the structural cohesiveness of classes inside a partition m_i (scoh) and the coupling (scop) between partitions (M), as illustrated by equation 4.

$$SM = \frac{1}{M}\sum_{i=1}^{M} scoh_i - \frac{1}{(M(M-1))/2} scop_{ij} \quad (4)$$

Where :

- $scoh_i = \frac{\mu_i}{m_i^2}$;
- μ_i refers to the number of calls internal to the partition m_i;
- $scop_{ij} = \frac{(\gamma_{ij})}{2*(m_i*m_j)}$;
- γ_{ij} refers to the number of calls being made between partitions m_i and m_j.

The higher SM value, the better the decomposition.

- **ICP :** Depicts the percentage of calls that occur between two divisions as shown by equation 5.

$$icp_{ij} = \frac{c_{ij}}{\sum_{i,j=0j/=i}^{M} c_{ij}} \quad (5)$$

Where :

- c_{ij} refers to the number of calls detected between partitions i and j.

The lower the ICP value, the better the recommendation.

- **Interface Number (IFN) :** This metric, denoted as IFN, is used to count the number of interfaces present in a microservice m_i. An interface is defined as a class within m_i that is invoked by a class within another microservice m_j.
The calculation of IFN is described by Equation 6.

$$IFN = \frac{1}{N}\sum_{i=1}^{N} ifn_i \quad (6)$$

Where :

- N refers to the total number of microservices;
- ifn_i is the number of interface classes in the microservice m_i.

The lower the IFN value, the better the recommendation.

- **Non-Extreme Distribution (NED) :** This metric assesses the distribution of classes within microservices and aims to ensure that a microservice is neither too large nor too small. According to the study [38], a microservice is considered non-extreme if it contains a number of classes within the range of [5, 20]. The metric is calculated using Equation 7.

$$NED = 1 - \frac{\sum_{k=0}^{N} n_k}{|N|} \quad (7)$$

Where :

- n_k refers to the number of the non-extreme microservices;
- N presents the total number of microservices.

The lower the NED value, the better the recommendation.

C. Evaluation and Results for RQ1

1) Evaluation protocol: The objective of this research question is to assess the quality of extracted microservices resulting from different combinations of structural and semantic information from classes within monolithic applications. The distinct strategies within our approach, elaborated in Figure

[3]https://anonymous.4open.science/r/Migration-to-microservicess-B67F/README.md

1, are independently applied and yield diverse inputs for subsequent clustering tasks.

To enhance comprehension of these combinations, we have assigned abbreviations to each one as follows:

- Configuration 1 : $Call_{in}$,$Call_{out}$+ Bag of Words.
- Configuration 2 : $Call_{in}$,$Call_{out}$+ TFIDF
- Configuration 3 : Call Frequencies+ Bag of Words.
- Configuration 4 : Call Frequencies+ TFIDF.
- Configuration 5 : CoDependant calls+ Bag of Words.
- Configuration 6 : CoDependant calls+ TFIDF.

The aim is to contrast the outcomes of various Configurations and pinpoint the most efficient one in terms of representing the monolithic application for the decomposition task. As an initial phase of the evaluation, hyperparameters were set based on existing literature.

- kernel bandwidth : Is set using the estimate bandwidth function from scikit-learn, which estimates the value of the bandwidth based on the provided data [54].
- MinPts : Is set to its default value (minPts = 1).
- Epsilon (ϵ) : The critical hyperparameter ϵ is set using a k-distance graph [21].

The DayTrader application is used to answer the first research question and its metadata is presented in Table I for the purpose of evaluating and comparing the various strategies.

2) Results: The presented Table II offers an overview of the evaluation results obtained from assessing the DayTrader Application across six distinct Configurations, each representing a unique combination of features.

Upon scrutinizing the diverse Configurations, the Structural Modularity metric (SM) values provide crucial insights into how effectively the generated microservices encapsulate the inherent structural relationships within the DayTrader Application. It is noteworthy that Configuration 4 significantly stands out with the highest SM value of 0.56. This prominence indicates that the fusion of Call Frequencies and TF-IDF in Configuration 4 yields microservices that adeptly capture the underlying structural dependencies. The elevated SM value indicates a strong alignment between Configuration 4's decomposition and the application's internal structure, suggesting the potential for more cohesive and organized microservices.

Shifting focus to the Interface Number (IFN) metric, which quantifies a microservice's interface dependencies, Configuration as long as Configuration 1 and 4 take the lead with the lowest IFN value of 0.93. This outcome implies that these strategies combinations effectively group features with fewer external interface dependencies. This attribute has the potential to foster the creation of self-contained and modular microservices.

Significantly, all configurations exhibit closely aligned Inter Call Percentage (ICP) values. Reducing communication between different components of the decomposition leads to a decrease in coupling, which can enhance the modularity and isolation of microservices, ultimately promoting component isolation.

Furthermore, the pivotal Non-Extreme Distribution (NED) metric aims to strike a balance between excessively large and excessively small microservices. Across the Configurations, NED values are closely clustered, signifying a favorable distribution that contributes to the overall quality of microservices' suggestion. Additionally, examining the number of detected microservices unveils variability among the Configurations, with Configuration 1 detecting the highest count (30) and Configuration 6 detecting the lowest count (22). A similar trend is apparent in the size of the largest microservice, with Configuration 6 featuring the largest (17) and Configuration 4 the smallest (12), both of which are within the classification recommandations outlined in the study [38].

In essence, the intricate interplay of these metrics underscores the multifaceted nature of the microservices decomposition task. Each Configuration presents distinct strengths and trade-offs. The culmination of these metrics guides the selection of the most effective Configuration aligned with specific project goals and desired outcomes. Notably, Configurations utilizing TF-IDF in the semantic part showcase favorable evaluation values. Additionally, the utilization of CoDependent calls as a structural representation and TF-IDF as a semantic representation leads to the lowest NED and ICP metrics.

To conclude, the meticulous evaluation of these metrics showcases how various attributes influence microservices' quality. While each Configuration showcases noteworthy aspects, the harmony between structural coherence, interface independence, coupling reduction and a well-distributed size spectrum, as captured by NED, makes Configuration 6 a standout choice.

Metrics	Configuration 1	Configuration 2	Configuration 3	Configuration 4	Configuration 5	Configuration 6
SM	0.41	0.52	0.43	0.56	0.44	0.40
IFN	0.93	1.2	1.07	0.93	1.12	1.3
ICP	0.65	0.64	0.62	0.64	0.63	0.63
NED	0.73	0.67	0.69	0.75	0.67	0.63
# microservices	30	25	26	32	25	22
size of the largest micro	14	15	14	12	14	17

TABLE II
EVALUATION RESULTS OF DAYTRADER APPLICATION.

> Configuration 6 proves to be the optimal approach, utilizing the CoDependent calls metric for structural information and TF-IDF vectors for semantic information. Consequently, our work will continue to focus on this strategy.

D. Evaluation and Results for RQ2

1) Evaluation protocol: Within this protocol, our experimental design encompasses two primary phases, each serving a distinct purpose. The overarching objective is to meticulously examine the performance and gauge the sensitivity of the adapted-BMSC algorithm in comparison to the individual performances of DBSCAN and Mean Shift algorithms.

In the initial stage of our experimentation, we conducted individual tests for each algorithm. Subsequently, we embarked on a comprehensive exploration of hyperparameters, a pivotal facet in algorithmic performance. To ensure a thorough assessment, we systematically delineated the potential values for each hyperparameter. With a focus on rigorous control,

we kept the other hyperparameters constant while varying a specific hyperparameter. For each conceivable value within the defined range, we executed the respective algorithms, recording the extracted microservices at each iteration. To comprehensively evaluate the results, we employed the suite of evaluation metrics, while concurrently plotting the metric values at each step to visualize their trends.
Our investigation centered around the DayTrader monolithic project, a widely recognized benchmark within this domain.

To elucidate the hyperparameter exploration process, we specifically targeted two hyperparameters of significance:

- **Kernel Bandwidth:** The estimate bandwidth function from the scikit-learn library informed our estimation of the central kernel bandwidth value. Subsequently, we systematically varied this parameter across a range that encompassed the estimated value, allowing for a thorough assessment of its impact on algorithmic performance.
- **Epsilon (ϵ):** With the aim of probing the influence of this hyperparameter, we systematically traversed the spectrum of Epsilon values, ranging from 0 to 1 with increments of 0.05.

By meticulously investigating these hyperparameters, our protocol endeavors to unravel the intricate dynamics of algorithmic performance and sensitivity, contributing to a nuanced understanding of adapted-BMSC, DBSCAN and Mean Shift within the context of software clustering.

2) Results: In our initial analysis experimentation, the comparison results among the three algorithms are illustrated in Table III where the results were detected using the estimated bandwith and the epsilon using K- distance Graph while keeping the MinPts set to 1. This comparison of clustering algorithms' evaluation outcomes offers captivating insights into their performance within the context of decomposing monolithic applications.Remarkably, Mean Shift exhibits a notably high structural modularity (SM) value of 0.87, signifying its proficiency in capturing structural relationships. Nevertheless, a more nuanced evaluation of its suitability for the task is warranted. Interestingly, despite adapted-BMSC displaying a lower SM value of 0.53, which is similar to the DBSCAN value of 0.54, further explanation and observations reveal adapted-BMSC as the leading algorithm due to its highly effective decomposition outcomes.

Turning our attention to Interface Number (IFN), DBSCAN boasts the lowest value at 0.34, suggesting its adeptness in generating microservices with fewer dependencies on external interfaces, a pivotal trait for effective modularization.

The assessment of Inter-Call Percentage (ICP) emphasizes the significance of reducing coupling. All approaches, namely DBSCAN, adapted-BMSC, and Mean Shift, maintain ICP values of 0.65, 0.63, and 0.61, respectively, demonstrating their commitment to minimizing communication between distinct components.

An illuminating aspect arises when delving into the number of microservices and the size of the largest microservice. Adapted-BMSC stands out with the detection of 22 microservices, showcasing its proficiency in skillfully segmenting the application into manageable components. This count starkly contrasts with DBSCAN's 88 and Mean Shift's 14 microservices. Additionally, adapted-BMSC demonstrates a well-balanced size distribution among its largest microservices, each with a size of 17, in stark contrast to Mean Shift and DBSCAN, where the largest microservices encompass 104 classes for Mean Shift and 9 for DBSCAN. This balance notably enhances the overall quality of the decomposition process.

In essence, adapted-BMSC's success in achieving the fundamental objectives of microservices decomposition, despite its seemingly lower structural modularity (SM) value compared to the results of other algorithms, underscores its holistic approach. By effectively tackling inherent issues in monolithic applications and attaining optimal interface independence, coupling reduction, and balanced distribution, adapted-BMSC emerges as the preferred algorithm for this specific decomposition task.

Metrics	Mean Shift	DBSCAN	adapted-BMSC
SM	0.87	0.54	0.53
IFN	1.21	0.34	1.03
ICP	0.61	0.65	0.63
NED	1	0.97	0.72
# microservices	14	88	29
size of the largest micro	104	9	16

TABLE III
EVALUATION RESULTS COMPARISON OF CLUSTERING ALGORITHMS USING DAYTRADER APPLICATION.

Transitioning to the second phase of analysis, we delve into the examination of hyperparameter sensitivity. The insights garnered from Figure 6 provide clear evidence that adapted-BMSC exhibits higher sensitivity compared to DBSCAN when subjected to variations in hyperparameters. Specifically, when assessing the epsilon hyperparameter ($BMSC_{eps}$ vs. $DBSCAN_{eps}$) and the bandwidth hyperparameter ($BMSC_{band}$ vs. $Mean\ shift_{band}$) across all metrics, adapted-BMSC's responsiveness is notably more pronounced.

Despite DBSCAN's superior performance in terms of SM and IFN metrics in comparison to adapted-BMSC, it yields a substantial number of microservices, averaging around 115 for an application containing 118 classes. Regrettably, this outcome does not align with our objectives. In contrast, adapted-BMSC demonstrates a more tempered sensitivity when varying the bandwidth hyperparameter compared to its sensitivity when altering the epsilon hyperparameter ($BMSC_{band}$ vs. $BMSC_{eps}$). This distinction arises from the fact that varying the bandwidth has the potential to generate differing numbers of modes, which are subsequently interconnected via DBSCAN. In contrast, changes in the epsilon hyperparameter directly influence the ultimate count of microservices, a fact underscored by the observed variations in the number of microservices.

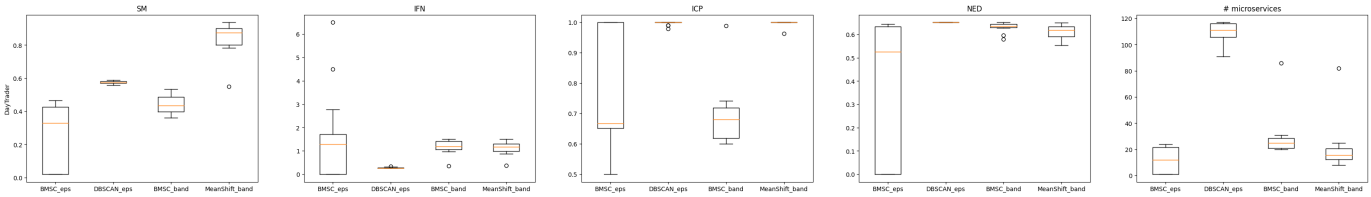

Fig. 6. Evaluation metrics for different hyperparameters values when extracting microservices from the project DayTrader.

> In contrast to the findings in [50], our analysis suggests that for our case, adapted-BMSC is more susceptible to the selection of its hyperparameters, specifically the epsilon hyperparameter, compared to DBSCAN and Mean Shift when used independently. However, adapted-BMSC demonstrates greater consistency in the resulting decomposition, as evidenced by its better NED values compared to other algorithms where the resulting microservices are either too large or too small.

E. Evaluation and Results for RQ3

1) Evaluation protocol: In order to evaluate the quality of our proposed solution, we compared it with six existing baselines include Bunch [33], CoGCN [40], FoSCI [39], MEM [42], Mono2Micro [38], and HierDecomp [36]. For this purpose and for a comprehensive evaluation, we experimented with all the approaches on four monolithic applications with varying complexity, namely DayTrader, Plants, JPetStore and Acmeair as presented in Table I.

To account for the hyper-parameter sensitivity of each solution, we generated multiple microservices decompositions from each baseline by varying their corresponding hyperparameters.

2) Results: Figure 7 provides a visual representation of the results through boxplots, offering a comprehensive view of the outcomes. Each row corresponds to a distinct project, and each column represents different metrics, considering various methodologies. In particular, let's focus on the first column, which pertains to the structural modularity metric (SM). In this context, our method stands out as a strong contender. It exhibits significant superiority over MEM, FOSCI, Bunch and HierDecomp when applied to the DayTrader project. However, when considering the Plants project, our method lags slightly behind HierDecomp and MEM, with a marginal difference of 0.15 in their mean values, while outperforming all other baseline methods.When we turn our focus to the last two projects, our solution consistently demonstrates superior performance compared to Mono2Micro, CoGCN, MEM, FOSCI, and Bunch when examining the JPetstore project. However, it's worth noting that the hierdecomp approach achieves the highest structural modularity value at 0.6. As for the acmeair project, we attained the lowest mean SM, with CoGCN securing the top rank.

Shifting our focus to the second column, which addresses the Interface Number (IFN) metric, our approach proves its effectiveness. Across the DayTrader, Plants, and JPetStore projects, it consistently maintains the lowest mean value compared to all baseline methods except Hierdecomp. In the Acmeair project, we outperformed Mono2Micro, FOSCI, and Bunch. This further emphasizes the robust performance of our method in efficiently managing interface numbers across a diverse range of projects.

Transitioning to the fourth column, which corresponds to the Non-Extreme Distribution (NED) metric, our approach demonstrates balanced performance. We outperformed hierdecomp's results in all projects except for JPetStore. When compared to Bunch and MEM, our results are notably favorable, closely resembling those obtained with CoGCN, especially in the case of the DayTrader application. Additionally, concerning the Acmeair project, our approach outperforms CoGCN and aligns closely with FOSCI. However, for the JPetStore application, HierDecomp records the best NED values.

It is worth noting that our approach does exhibit a slightly higher Inter-Call Percentage (ICP) value compared to most baseline methods. The exceptions are FOSCI, which records the highest ICP value for the DayTrader project, and HierDecomp, Mono2micro, MEM, as well as an outlier result from FOSCI for the JPetStore application.

Examining the fifth and final column, which pertains to the number of generated microservices, our approach consistently demonstrates robustness in its decomposition results, displaying minimal variability when hyperparameters are adjusted. The only exception to this pattern is a narrow range of variability observed in the HierDecomp approach. Notably, our approach often leads to a higher number of microservices compared to all other approaches, as is particularly evident in the AcmeAir and JPetStore projects.

One notable advantage of our approach is its remarkable stability, especially in terms of the number of identified microservices. This steadiness emphasizes its robustness and underscores its efficacy when contrasted with alternative methodologies.

> The comparison results suggest that our proposed solution yields promising outcomes when compared to the baseline approaches, particularly in terms of SM, IFN, and NED metrics. However, this enhancement is associated with higher values in the ICP metric.

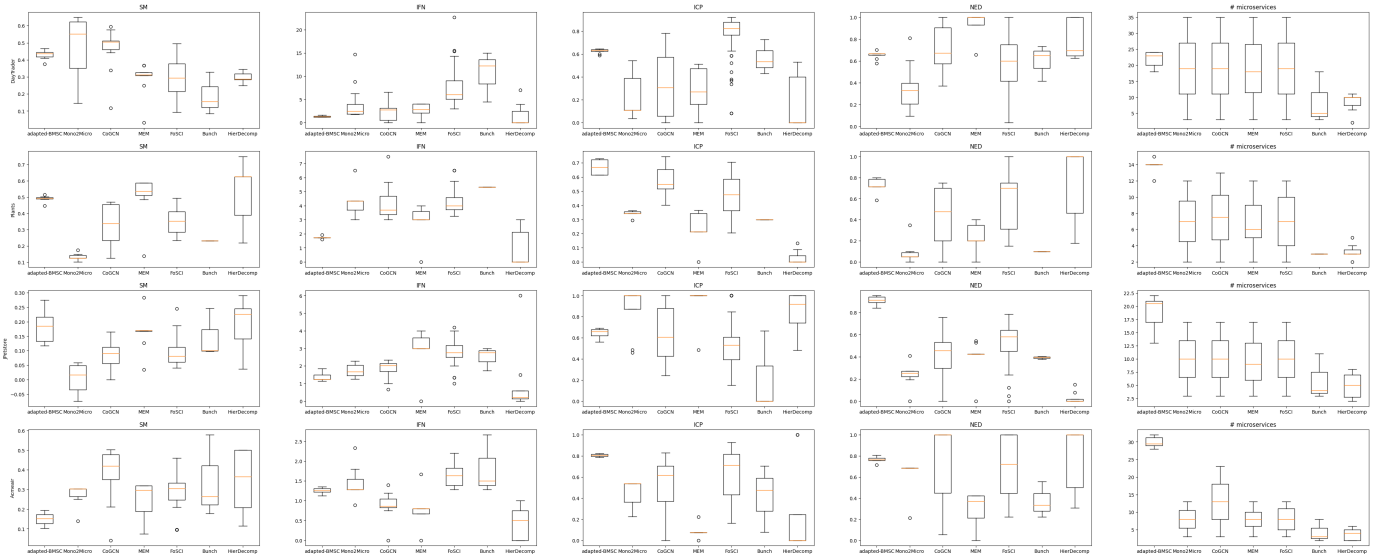

Fig. 7. Boxplot Analysis of Project/Baseline/Metric Combinations.

F. Evaluation and Results for RQ4

1) Evaluation protocol: To address RQ4, we opted to examine three Java-based projects that utilize microservices architecture, each exhibiting varying levels of complexity. These projects are presented in Table IV

Project	Version	SLOC	# of classes	# of microservices
Spring PetClinic [8]	2.3.6	1,889	43	7
Microservices Event Sourcing [9]	2.8.0	4,597	121	12
Kanban Board [10]	0.1.0	4,380	118	21

TABLE IV
- CHARACTERISTICS OF MICROSERVICE-BASED APPLICATIONS

In our research study aimed at evaluating the effectiveness of our proposed approach, it is crucial to define appropriate metrics that can accurately compare two sets of classes representing the extracted microservices with their corresponding ground truth microservices. However, the process of identifying the corresponding ground truth microservice for each extracted microservice is a challenging task. To overcome this challenge, inspired from the work of Sellami and al [36],we have developed a method that utilizes the number of common classes between the extracted microservice and each of the ground truth microservices to determine the corresponding microservice.

To be specific, we introduced a function represented by equation 8 that takes an extracted microservice m_i and a set of ground truth microservices M as input and selects the ground truth microservice with the highest number of common classes with the extracted microservice.

$$Corr(m_i, M) = argmax_{m_j \in M}\left(\frac{|m_i \cap m_j|}{|m_i|}\right) \tag{8}$$

After identifying the ground truth microservices that correspond to each extracted microservice using our suggested method, it is necessary to calculate relevant metrics in order to evaluate our methodology. Among the statistics used is precision, which is calculated using equation 9.

$$Precision = \frac{1}{|M|} \times \sum_{\forall m_i \in M} \frac{|m_i \cap Corr(m_i, M_t)|}{|m_i|} \tag{9}$$

Precision is a measure of how accurate our technique is in identifying the classes that belong to each microservice. It provides the average proportion of correctly recognised classes relative to the total number of identified classes for each extracted microservice. Through the calculation of precision, we can gauge the effectiveness of our approach in accurately identifying the classes that correspond to each microservice.

To evaluate the effectiveness of our approach in identifying the microservices, we also compute another metric called the Success Rate (SR). The SR is calculated using equation 10 and measures the percentage of successfully retrieved microservices based on the precision metric.

$$SR = \frac{1}{|M|} \times \sum_{\forall m_i \in M} matching(m_i, Corr(m_i, M_t)) \tag{10}$$

The SR provides a complementary perspective to the precision metric in assessing the performance of our approach. It takes into account the overall number of correctly identified microservices and their precision, providing a more comprehensive evaluation of our approach's effectiveness where :

$$matcing(m_1, m_2) = \begin{cases} 1 & if \frac{|m_1 \cap m_2|}{|m_1|} \geq threshold \\ 0 & otherwise \end{cases} \tag{11}$$

Specifically, we consider two sets of classes, m_1 and m_2, and a threshold value, threshold $\in [0, 1]$. Using these inputs, we calculate the success rate (SR) at k, for a given k value ranging from 1 to 10.

In order to evaluate the performance of our microservices decomposition approach, we take the following steps. Firstly,

we collect all Java classes for each test project and combine them to form a Monolithic architecture. This serves as the input to our approach, while the original version of the project is considered as the ground truth decomposition for comparison. Next, we generate multiple microservices decompositions using different hyperparameter values, and for each decomposition, we calculate several metrics such as precision, SR@5, SR@7, and SR@9, along with the corresponding true decomposition. This enables us to analyze the effectiveness of our approach in generating microservices decompositions for a wide range of hyperparameter values.

2) Results: The study's results are shown in Figure 8 through boxplots for each project and metric. Each project's median precision values are within the range of 0.6 to 0.65. More specifically, the Kanban Board demo, Microservices Event Sourcing projects and spring petclinic project achieved precision values greater than 0.56 for all decompositions except for one outlier in the Kanban Board application adn for the spring petclinic with a precision value close to 0.55. The precision variability was reasonable, with a maximum score variation goes for the Microservices Event sourcing application and the smallest variability observed in the spring petclinic project. These results suggest that the method remains stable despite the increase in the number of classes, but projects with a smaller number of classes perform better.

As for the success rate, we observed that as the threshold increases, the median and maximum scores for each project drop. However, the Kanban Board demo and Microservices Event Sourcing projects have less variance in their values. At SR@5, all projects achieved a precision higher than 0.6 for all decompositions, except for an outlier close to 0.5 detected in the Kanban Board project. The spring petclinic project had the highest precision at this rate, exceeding 0.8 as a median. Notably, the scores for SR@7 to the strictest SR@9 were the same for all projects. These results suggest that a high percentage of microservices in all decompositions achieved a median precision score higher than 0.25. Moreover, as the project's size decreased, the variance of the results pattern decreased.

The variability observed in the derived microservices outcomes of the Microservices Event Sourcing project could potentially be attributed to its incorporation of multiple natural languages for domain terms, a distinction not present in the other projects that solely employ English. This linguistic diversity poses a challenge to the re-sampling process, which might lead to misclassification of certain classes with their nearest iModes, thus impacting their accurate assignment to the appropriate microservices. Nonetheless, the outcomes attained by the Microservices Event Sourcing project remain in line with those of its counterparts. It's noteworthy that the Microservices Event Sourcing project exhibited superior performance, surpassing other projects with its highest precision value in the most stringent thresholds.

For a more comprehensive analysis, we delve into the insights presented in Figure 9. As a case in point, we examine the decomposition results of the Kanban Board demo project. This Java-based application, developed using Spring Boot, serves as a practical illustration of how the Eventuate Platform can facilitate the construction of real-time, multi-user collaborative applications. Specifically, the Kanban Board application showcases the collaborative creation and editing of Kanban boards and tasks. Any modifications initiated by one user on a board or task are instantly reflected to other users who are concurrently accessing the same board or task.

The architecture of the Kanban Board application relies on Eventuate's Event Sourcing-based programming model, which is optimally suited for such use cases. The application persists business objects such as Boards and Tasks, as a sequence of events that alter their state. Upon a user's action to create or update a board or task, the application records an event in the event store. Subsequently, this event is conveyed to subscribers with an interest in the event. Within the Kanban application, an event subscriber transforms each event into WebSocket messages, facilitating real-time updates in each user's browser interface.

Given the scale of the project, Figure 9 presents a subset of the microservices that resulted from one of the decomposition processes. In this illustrative representation, ellipses denote the original microservices' names, while large white rectangles symbolize the new microservices that have been generated. These rectangles encompass the classes, which are color-coded based on their originating microservices.

Importantly, it's pertinent to highlight that in this specific decomposition, the exact count of microservices matches that of the original application. However, it's essential to recognize that obtaining an identical distribution of classes is not necessarily guaranteed. A closer examination of the results reveals intriguing nuances. For instance, within Microservice 2, we observe an aggregation of 7 classes that were initially divided4 residing in the task-query-side microservice and 3 in the common-task microservice. Our approach amalgamated these classes due to their shared implementation of task specifications and services.

"Let's explore the original Test-utils microservices in more detail. In our approach, its classes underwent partitioning into two distinct microservices, namely Microservice 2 and Microservice 3. Additionally, Microservice 4 emerged exclusively to encapsulate the board-command-side classes. Notably, the concept of Board-query-side also underwent partitioning, resulting in Microservices 5 and 2.

However, an interesting observation lies in the amalgamation of the BoardQueryController class with the test utils concept in our approach. If we focus more on this classification of classes, we find that the BoardQueryController class shares a significant number of words in its vocabulary with the remaining classes associated with the same microservice in our approach. Such words include "native," "code," "id," "notification," "class," "hash," "equals," "wait," "clone," and more. This intricate differentiation highlights the nuanced decisions our approach makes to optimize microservice composition while maintaining functionality and cohesion."

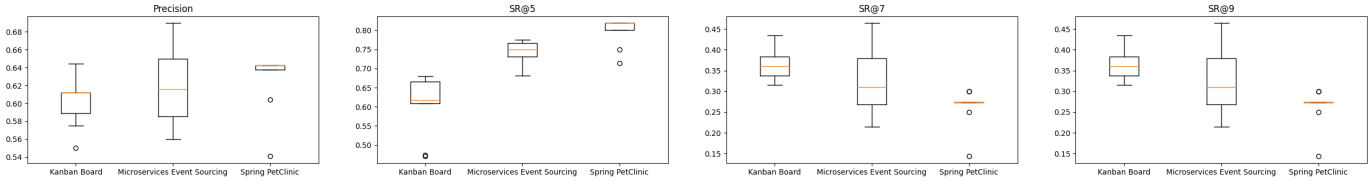

Fig. 8. A comparative analysis of generated vs true decompositions through boxplots.

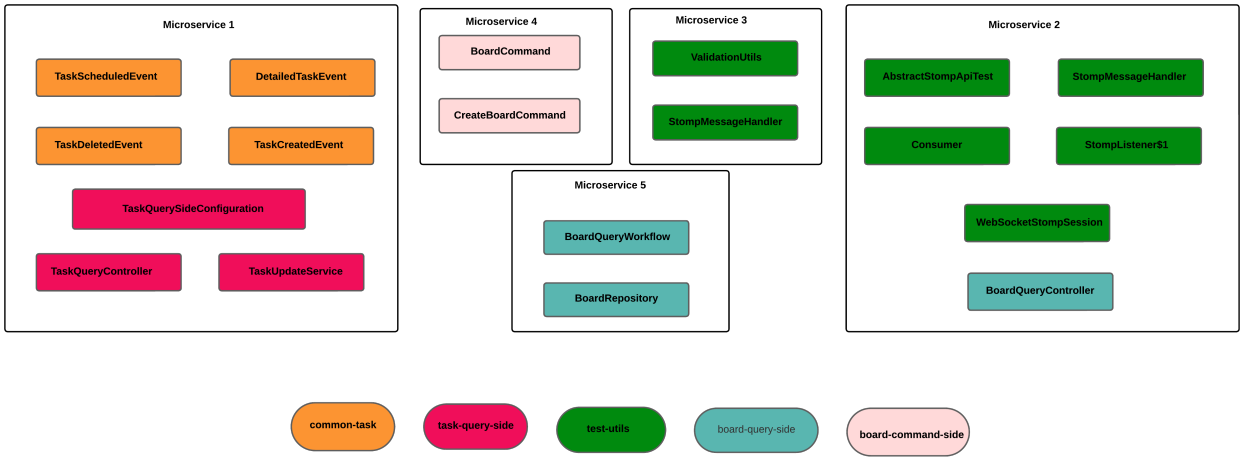

Fig. 9. A subset of the microservices obtained from a decomposition of the project Kanban Board Demo.

The results obtained show that the extracted microservices achieve a median precision score around 0.6. Interestingly, it was observed that the extracted microservices, while not entirely identical to the original microservices, still included most of the classes from the human-built microservices. These results suggest that utilizing machine learning-based approaches for microservice extraction could be a promising direction for software development.

V. Threats to Validity

The validity of our findings may be limited in terms of external and internal validity. External validity is limited due to the small sample size of seven diverse applications used in our review. Additionally, the approach has only been tested on Java-based applications at a class level, which may limit generalization to other programming languages and levels of coverage. Regarding internal validity, we need to consider the possibility of coding errors or bugs in our model's implementation, experimental infrastructure and data collection. To address this concern, we took extensive measures, such as conducting code reviews and rigorous testing throughout the development process. Additionally, we ran multiple iterations of experiments to ensure the consistency and reliability of our results.

VI. Conclusion and Future Work

In this study, our primary focus was to address the challenge of decomposing a monolithic application into a recommended set of new microservices through a clustering task. The proposed method involved utilizing static analysis tools on the monolithic application's source code to extract class dependencies and encode both structural and semantic information. Subsequently, we employed the adapted-Boosted Mean Shift Clustering algorithm to extract microservices from this encoding, leveraging its advantages, including the ability to infer the number of microservices and robustness to outlier classes. In addition to conducting a sensitivity analysis on hyperparameters, we evaluated the performance of our approach by comparing it with six baseline methods and assessing the quality of the extracted microservices. Our approach yielded encouraging results across most of the comparison metrics, demonstrating its effectiveness in addressing the decomposition challenge.

To take the granularity level to an advanced level, the approach could be further developed to use methods or functions of the monolith as a basis for decomposition rather than classes. Furthermore, since static analysis does not provide all of the information necessary for a clear understanding of functionalities and interactions during application execution, a hybrid solution incorporating dynamic analysis of the source code could be developed.

REFERENCES

[1] F. Tapia, M. . Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, "From monolithic systems to microservices: A comparative study of performance," *Applied Sciences*, vol. 10, no. 17, 2020.

[2] O. Benomar, H. Abdeen, H. Sahraoui, P. Poulin, and M. A. Saied, "Detection of software evolution phases based on development activities," in *2015 IEEE 23rd International Conference on Program Comprehension*, pp. 15–24, IEEE, 2015.

[3] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A kubernetes controller for managing the availability of elastic microservice based stateful applications," *Journal of Systems and Software*, vol. 175, p. 110924, 2021.

[4] V. Velepucha and P. Flores, "Monoliths to microservices - migration problems and challenges: A sms," in *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, pp. 135–142, 2021.

[5] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with kubernetes: Experiments and lessons learned," in *2018 IEEE 11th international conference on cloud computing (CLOUD)*, pp. 970–973, IEEE, 2018.

[6] M. A. Saied, H. Sahraoui, E. Batot, M. Famelis, and P.-O. Talbot, "Towards the automated recovery of complex temporal api-usage patterns," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1435–1442, 2018.

[7] S. Huppe, M. A. Saied, and H. Sahraoui, "Mining complex temporal api usage patterns: an evolutionary approach," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 274–276, IEEE, 2017.

[8] M. A. Saied, O. Benomar, and H. Sahraoui, "Visualization based api usage patterns refining," in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pp. 155–159, IEEE, 2015.

[9] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Microservice based architecture: Towards high-availability for stateful applications with kubernetes," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pp. 176–185, IEEE, 2019.

[10] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 1st ed., February 2015.

[11] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices migration in industry: Intentions, strategies, and challenges," in *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, pp. 481–490, Institute of Electrical and Electronics Engineers Inc., 9 2019.

[12] A. Shatnawi, H. Shatnawi, M. A. Saied, Z. A. Shara, H. Sahraoui, and A. Seriai, "Identifying software components from object-oriented apis based on dynamic analysis," in *Proceedings of the 26th Conference on Program Comprehension*, pp. 189–199, 2018.

[13] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20357–20374, 2022.

[14] P. Rosati, F. Fowley, C. Pahl, D. Taibi, and T. Lynn, "Right scaling for right pricing: A case study on total cost of ownership measurement for cloud migration," in *Communications in Computer and Information Science*, vol. 1073, pp. 190–214, Springer Verlag, 2019.

[15] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment* (J.-M. Bruel, M. Mazzara, and B. Meyer, eds.), vol. 11350, pp. 128–141, Springer International Publishing, 2019.

[16] M. A. Saied, H. Abdeen, O. Benomar, and H. Sahraoui, "Could we infer unordered api usage patterns only using the library source code?," in *2015 IEEE 23rd International Conference on Program Comprehension*, pp. 71–81, IEEE, 2015.

[17] S. Mujahid, D. E. Costa, R. Abdalkareem, E. Shihab, M. A. Saied, and B. Adams, "Toward using package centrality trend to identify packages in decline," *IEEE Transactions on Engineering Management*, 2021.

[18] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an availability manager for microservice applications," *arXiv preprint arXiv:1901.04946*, 2019.

[19] K. Sellami, A. Ouni, M. A. Saied, S. Bouktif, and M. W. Mkaouer, "Improving microservices extraction using evolutionary search," *Information and Software Technology*, p. 106996, 2022.

[20] N. Almarimi, A. Ouni, S. Bouktif, M. W. Mkaouer, R. G. Kula, and M. A. Saied, "Web service api recommendation for automated mashup creation using multi-objective evolutionary search," *Applied Soft Computing*, vol. 85, p. 105830, 2019.

[21] K. Sellami, M. A. Saied, and A. Ouni, "A hierarchical dbscan method for extracting microservices from monolithic applications," in *The International Conference on Evaluation and Assessment in Software Engineering 2022*, pp. 201–210, 2022.

[22] I. Saidani, A. Ouni, M. W. Mkaouer, and A. Saied, "Towards automated microservices extraction using muti-objective evolutionary search," in *International Conference on Service-Oriented Computing*, pp. 58–63, Springer, 2019.

[23] M. A. Saied, E. Raelijohn, E. Batot, M. Famelis, and H. Sahraoui, "Towards assisting developers in api usage by automated recovery of complex temporal patterns," *Information and Software Technology*, vol. 119, p. 106213, 2020.

[24] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Improving reusability of software libraries through usage pattern mining," *Journal of Systems and Software*, vol. 145, pp. 164–179, 2018.

[25] M. Gallais-Jimenez, H. A. Nguyen, M. A. Saied, T. N. Nguyen, and H. Sahraoui, "Api misuse detection an immune system inspired approach," *arXiv preprint arXiv:2012.14078*, 2020.

[26] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Automated inference of software library usage patterns," *arXiv preprint arXiv:1612.01626*, 2016.

[27] M. A. Saied and H. Sahraoui, "A cooperative approach for combining client-based and library-based api usage pattern mining," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–10, IEEE, 2016.

[28] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui, "Mining multi-level api usage patterns," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pp. 23–32, IEEE, 2015.

[29] M. A. Saied, H. Sahraoui, and B. Dufour, "An observational study on api usage constraints and their documentation," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 33–42, IEEE, 2015.

[30] M. Mouine and M. A. Saied, "Event-driven approach for monitoring and orchestration of cloud and edge-enabled iot systems," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pp. 273–282, IEEE, 2022.

[31] K. Sellami, M. A. Saied, A. Ouni, and R. Abdalkareem, "Combining static and dynamic analysis to decompose monolithic application into microservices," in *International Conference on Service-Oriented Computing*, pp. 203–218, Springer, 2022.

[32] K. Sellami, A. Ouni, M. A. Saied, S. Bouktif, and M. W. Mkaouer, "Improving microservices extraction using evolutionary search," *Information and Software Technology*, vol. 151, p. 106996, 2022.

[33] B. S. Mitchell and S. Mancoridis, "On the evaluation of the bunch search-based software modularization algorithm," *Soft Computing*, vol. 12, no. 1, pp. 77–93, 2008-01-01.

[34] O. Al-Debagy and P. Martinek, "A microservice decomposition method through using distributed representation of source code," *Scalable Computing*, vol. 22, pp. 39–52, 2021-02.

[35] O. Al-Debagy and P. Martinek, "A microservice decomposition method through using distributed representation of source code," *Scalable Computing*, vol. 22, pp. 39–52, 2021.

[36] K. Sellami, M. A. Saied, and A. Ouni, "A hierarchical dbscan method for extracting microservices from monolithic applications," in *The International Conference on Evaluation and Assessment in Software Engineering 2022*, pp. 201–210, 2022.

[37] M. Brito, J. Cunha, and J. Saraiva, "Identification of microservices from monolithic applications through topic modelling," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pp. 1409–1418, ACM, 2021.

[38] A. K. Kalia, X. Jin, K. Rahul, S. Saurabh, V. Maja, and B. Debasish, "Mono2micro: A practical and effective tool for decomposing monolithic java applications to microservices," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[39] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 987–1007, 2021.

[40] U. Desai, S. Bandyopadhyay, and S. Tamilselvam, "Graph neural network to dilute outliers for refactoring monolith application."

[41] K. Sellami, M. A. Saied, A. Ouni, and R. Abdalkareem, "Combining static anddynamic analysis todecompose monolithic application intomicroservices," in *Service-Oriented Computing* (J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, and A. Ruiz-Cortés, eds.), (Cham), pp. 203–218, Springer Nature Switzerland, 2022.

[42] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *2017 IEEE International Conference on Web Services (ICWS)*, pp. 524–531, 2017.

[43] M. Gysel, L. Klbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing* (M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, eds.), vol. 9846, pp. 185–200, Springer International Publishing, 2016.

[44] M. Dehghani, S. Kolahdouz-Rahimi, M. Tisi, and D. Tamzalit, "Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning," *Softw Syst Model*, vol. 21, pp. 1115–1133, 2022.

[45] H. Brunelire, J. Cabot, G. Dup, and F. Madiot, "MoDisco: A model driven reverse engineering framework," *Information and Software Technology*, pp. 1012–1032, 2014.

[46] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *Science*, vol. 315, no. 5814, pp. 972–976, 2007-02-16.

[47] R. Sibson, "Slink: An optimally efficient algorithm for the single-link cluster method," *Comput. J.*, vol. 16, pp. 30–34, 1973.

[48] D. Deng, "Dbscan clustering algorithm based on density," *2020 7th International Forum on Electrical Engineering and Automation (IFEEA)*, pp. 949–953, 2020.

[49] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Trans. Evol. Comput.*, vol. 6, pp. 182–197, 2002.

[50] Y. Ren, U. Kamath, C. Domeniconi, and G. Zhang, "Boosted mean shift clustering," in *Machine Learning and Knowledge Discovery in Databases* (T. Calders, F. Esposito, E. Hllermeier, and R. Meo, eds.), vol. 8725, pp. 646–661, Springer Berlin Heidelberg, 2014.

[51] H. V. Singh, A. Girdhar, and S. Dahiya, "A literature survey based on DBSCAN algorithms," in *2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS)*, pp. 751–758, 2022.

[52] K. G. Derpanis, "Mean shift clustering."

[53] A. Mishra and S. K. Vishwakarma, "Analysis of tf-idf model and its variant for document retrieval," *2015 International Conference on Computational Intelligence and Communication Networks (CICN)*, pp. 772–776, 2015.

[54] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.