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

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

Khaled Sellami
Laval University
Quebec, QC, Canada
khaled.sellami.1@ulaval.ca

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

***Abstract*—The microservice architectural style has gained widespread popularity among developers due to its ability to provide numerous benefits, such as scalability, reusability and easy maintainability. However, transforming a monolithic application into a microservices-based architecture can be a complex and an expensive process. To address this challenge, we propose a novel method that leverages clustering to identify potential microservices from a monolithic application. Our approach uses a density-based clustering algorithm that considers the static analysis, structural and semantic relationships between the classes to establish a functionally coherent class partitioning. To evaluate our approach, we analyzed its hyperparameter sensitivity and compared it to two other well known clustering algorithms using various metrics on a Java applications. Our approach showed promising results, demonstrating its effectiveness and stability.**

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 [1] 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 [2]–[4].

Many methods have arisen throughout time to solve these issues, such as migrating to new technologies, managing independent services, and deploying more powerful servers. Despite the availability of these solutions, monolithic architectures are still limited by inherent drawbacks such as their large, complex, and often inefficient nature, which may hinder their ability to support advanced and more sophisticated technologies [5]–[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. 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. 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 [11]. The transition from a monolithic design to a more durable 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 [12], 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.

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 and density-based clustering algorithm to divide the classes into multiple potential microservices that may be evaluated further. We conducted an in-depth review utilizing a variety of metrics to measure the efficacy and efficiency of our method.

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.

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

II. Proposed Approach

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

Our technique begins with the extraction of semantic and structural information via static analysis of the source code. Then, it evaluates all potential combinations while selecting only one option from each semantic and structural preprocessing component. We feed these representations to each one of the clustering algorithms, resulting in three distinct decompositions that will be analyzed and compared .

A. Representation of the Monolithic Application

The monolithic application is represented as a set of Object Oriented Programming classes denoted as C_M=(c_1,...,c_Z), where Z represents the total number of classes. In this context, our approach aims to partition the original monolithic application into a set of K microservices M=(m_1,..,m_K). Each microservice, m_i=(c_a,...,c_p), represents a subset of the original classes. We aim to optimize the migration process, where each microservice is expected to be cohesive and loosely coupled, resulting in a more maintainable and scalable architecture.
The initial step, presented in the diagram in Figure 1, focuses on representing the monolithic application and extracting the necessary information to build the microservices. To do this, we begin by creating an encoding scheme for the monolith's classes to capture their structural and semantic links.

1) Structural encoding: Abstract Syntax Trees (ASTs) can be created after the source code has been parsed using a static analysis tool, such as "Understand" [9]. These ASTs are used to extract call relationships between classes in a form of an interaction graph. As described in task 2.1 of the diagram presented in Figure 1 the structural information can be encoded using three different options:

- $Call_{in}$, $Call_{out}$: Each class is represented as the sum of incoming and outgoing calls. Our strategy seeks to group classes that interact frequently within the same cluster in order to reduce coupling while promoting greater cohesion within the resulting microservices.
- Call frequencies: This option tries to build more coherent clusters by encoding classes in greater depth. We analyse the call frequencies between each pair of classes to capture a more nuanced understanding of class connections.
- Codependent calls: We consider call frequencies of classes that interacts with both classes to encode each pair of classes. To aid in understanding this concept, we will go through the example in Figure 2 . We have 4 classes: A, B, C, and D. The objective is to encode the pair of classes A and B. Class A is invoked five times by class B, three times by C, and once by D. In addition, class B is invoked twice by C and once by D. 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 idea behind this technique is that classes that are frequently called together are usually used to handle the same functionality.

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. 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 microservices.

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, method names, and variable names. To preprocess these words, we separate them using CamelCase, filter out stop words and normalise them using stemming.

As seen in task 2.2 in Figure 1, the processed semantic information will be represented in two options:

- Terms frequencies: It 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.
- Term Frequency-Inverse Document Frequency: TF-IDF can improve class clustering in a variety of ways, it considers not only the frequency of a word in a class, but also the inverse document frequency to assess how unique a term is to a class in comparison to the vocabulary. Thus, unique terms that are exclusive to a class will have a larger weight and will be more informative.

B. Clustering algorithms

The objective is to extract microservices by encoding classes structurally and semantically using different combinations of options. To achieve this, we experimented with the Boosted Mean Shift Clustering (BMSC) [13] algorithm, along with other well-known clustering algorithms such as Density-Based

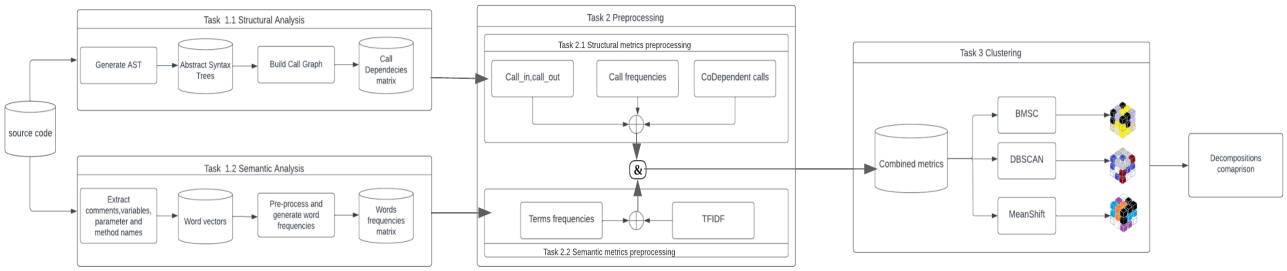

Figure 1: Overview of the Microservices Extraction Process.

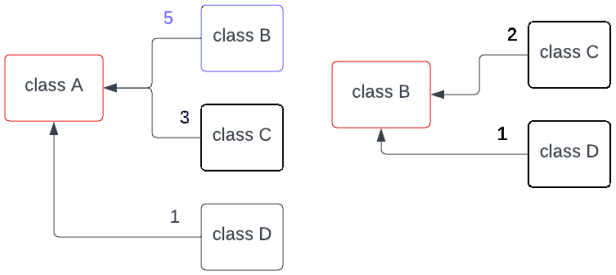

Figure 2: Illustrative example of Codependent calls metric.

Spatial Clustering of Applications with Noise (DBSCAN) [14] and Mean Shift [15].

1) DBSCAN algorithm: DBSCAN is a clustering technique 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 based on the data and the provided hyperparameters, the number of clusters can be arbitrary detected, leading in more accurate clusters [16].

Hyperparameters :

- **Eps** (ϵ) **:** Refers to the radius of the neighbourhood surrounding the cluster's central point.
- **MinPts :** This is the bare minimum of points required to build a cluster.

To build clusters, DBSCAN begins by picking an arbitrary Core point, then it 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 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 that composed it from the database. When no further clusters can be produced with the provided hyparameters, the algorithm stops. The rest of the points are labelled as Noise.

However, this algorithm is highly sensitive to its hyper-parameters, 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 does not need any assumptions about the underlying distribution of the data. It can automatically detect non-linearly formed clusters and compute the number of clusters [15].

It begins by arbitrary identifying a region of interest and calculate its center of density. The mean shift vector is then generated and the center of the area is shifted along the vector until it corresponds with the centre of mass.

Although the Mean Shift method has shown excellent results, the research in [13] demonstrates that BMSC outperformed Mean Shift in a similar clustering problem with more stable clustering.

Given the difficulties involved with clustering algorithms specially when there is no obvious separation between clusters, we decided to investigate alternatives to standard techniques. We picked the BMSC technique since it has showed higher performance in similar clustering tasks.

3) BMSC algorithm: BMSC algorithm is a hybrid clustering technique that combines Mean Shift and DBSCAN. It is a density-based clustering method 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 [13].

The BMSC first applies the Mean Shift algorithm to generate a set of initial centers that will be the input to the DBSCAN algorithm. BMSC selects a sample of the data that captures the skeleton of the clusters in order to properly identify the data's underlying structure.

Algorithm 1 outlines the steps involved in applying the BMSC algorithm. The first step is to divide the data uniformly into cells of a grid. Then, the Mean Shift algorithm is applied independently to the data in each cell. This produces a list of intermediate mode points (iModes). Next, it disperse the data of the cells using a specific mechanism that involves each grid cell interacting with a limited number of cells in its neighborhood. The BMSC paper [13] presents various

Algorithm 1 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)

neighborhood structures, which are depicted in Figure 3. In our work, we adopt the linear (5) neighborhood structure.

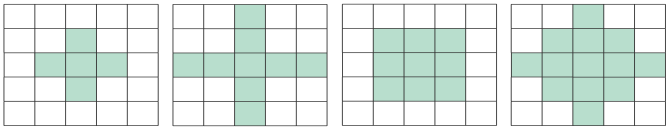

Figure 3: Potential neighbourhood structures.

The subsequent phase involves calculating the distances between all data points in the parent cell and those in its neighboring cells, relative to the iModes using a semantic similarity metric that assesses the confidence level of each relationship. The second stage of the BMSC algorithm utilizes the list of iModes to run DBSCAN. The latter is applied to identify clusters of densely packed iModes, which in turn generates clusters of the original data points.

In our particular scenario, we utilize an aggregation function to transform the iModes into a format similar to that of the legacy application's classes. We represent each group center by summing the structural encodings of its classes, 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 [17] where instead of directly inputting the encoders of iModes into the DBSCAN algorithm, we provide the connections or links between each pair of iModes. To achieve this, we employ the iModes similarity measure that capture the structural and semantic relationships. This approach aims to produce microservices that are consistent from implementation and use cases perspectives. The similarity is calculated as follows :

- **iModes Similarity (MS) :** The 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 : $\alpha, \beta \in$ *[0,1],* $\alpha + \beta = 1$.
Each one of the similarities is computed as follow:

- **Structural similarity** (Sim_{str}) : This allows us to evaluate their similarity from a functional perspective. It's computed 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): The number of calls of m_j by m_i,
- $call_{in}(m_i)$: The number of incoming calls in m_i.
- **Semantic Similarity** (Sim_{sem}) **:** Is represented by the cosine similarity between their respective vectors. This is useful for measuring the similarity between different iModes and identifying possible relationships at the domain level [18].

Finally, we employ DBSCAN algorithm on the iModes similarity metric. The process is iterated until production of the same number of clusters for three consecutive iterations.

III. Evaluation

To help better understand the evaluation, Table I summarizes the characteristics of the monolithic application used to evaluate different aspects of our approach.

TABLE I: CHARACTERISTICS OF MONOLITHIC APPLICATION

Project	Version	SLOC	# of classes
DayTrader	1.4	18,224	118

A. Research Questions

The goal of our experimental investigation is to address these research questions (RQs):
RQ1: What is the most effective and promising option among the various choices in our approach?
RQ2: How does the stability and robustness of BMSC algorithms compare to that of Mean Shift and DBSCAN?

B. Evaluation metrics

We used a set of metrics specified in [19] 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 and the coupling between partitions.
  The higher SM value, the better the decomposition.

- **ICP:** Depicts the percentage of calls that occur between two divisions.
  The lower the ICP value, the better the recommendation.
- **Interface Number (IFN):** It counts the number of interfaces present in a microservice. An interface is defined as a class within a microservice that is invoked by a class within another microservice.
  The lower the IFN value, the better the recommendation.
- **Non-Extreme Distribution (NED):** It assesses the distribution of classes within microservices and aims to ensure that a microservice is non-extreme. According to [19], a microservice is considered non-extreme if it contains a number of classes within the range of [5, 20]. The lower the NED value, the better the recommendation.

C. Evaluation and Results for RQ1

1) Evaluation protocol: The objective is to compare the quality of the findings from different possible combinations of structural and semantic information in order to identify the most effective strategy for each algorithm using DayTrader application. We assigne abbreviations as follows:

- Option 1 : Call$_{in}$,$Call_{out}$ + Terms frequencies.
- Option 2 : Call$_{in}$,$Call_{out}$ + TFIDF
- Option 3 : Call Frequencies + Terms frequencies.
- Option 4 : Call Frequencies + TFIDF.
- Option 5 : Codependant calls + Terms frequencies.
- Option 6 : Codependant calls + TFIDF.

Hyperparameters were fixed according to the literature:

- Bandwidth :Is set using the estimate bandwidth function from scikit-learn, which estimates the value of the bandwidth based on the provided data.
- MinPts : Is set to 5 because a cluster is considered not extreme if its size ranges from 5 to 20 [19] for DBSCAN alone and set to its default value (MinPts = 1) for BMSC algorithm [13].
- Eps : Is set using a k-distance graph technique.

2) Results: According to the results presented in Table II, option 6 is deemed the most suitable for the Mean Shift algorithm when considering the SM metric and shows a performance that is comparable to the best results obtained when evaluating other metrics. For DBSCAN, presented in Table III, option 6 has shown better results in terms of IFN, ICP, and NED, with a SM value that is close to the maximum. Both DBSCAN and Mean Shift algorithms presented varied results, while BMSC had very similar results for all options and all metrics. Furthermore, BMSC was able to detect a more stable number of microservices compared to the other algorithms, which often formed one large cluster or unique classes that did not meet the research goals. In contrast, the resultant microservices from BMSC were balanced and stable across different approaches, with the largest microservice containing a maximum of 17 classes as presented in Table IV.

TABLE II: EVALUATION RESULTS OF DAYTRADER APPLICATION USING MEAN SHIFT ALGORITHM

Metrics	Option 1	Option 2	Option 3	Option 4	Option 5	Option 6
SM	0.8526	0.7853	0.7944	0.8614	0.8575	0.8742
IFN	1.235	1.8	1.277	1.0454	1.0	1.214
ICP	1.0	0.9	1.0	1.0	1.0	1.0
NED	1.0	0.9	1.0	1.0	1.0	1.0
# microservices	17	10	18	22	21	14
size of the largest micro	98	102	97	92	97	104

TABLE III: EVALUATION RESULTS OF DAYTRADER APPLICATION USING DBSCAN ALGORITHM

Metrics	Option 1	Option 2	Option 3	Option 4	Option 5	Option 6
SM	0.120	0.1085	0.2702	0.2718	0.2487	0.1116
IFN	0.120	0.1085	0.2702	0.2718	0.2487	0.1116
ICP	0.3244	0.1426	0.1591	0.2859	0.3482	0.0079
NED	0.5	0.666	1.0	0.5	0.5	0.333
# microservices	2	3	2	2	2	3
size of the largest micro	108	86	116	113	113	104

Option 6 is the optimal approach for all three clustering algorithms. It uses co-dependent calls metric as structural information and TF-IDF vector as semantic information. Consequently, our work will continue to focus on this strategy.

D. Evaluation and Results for RQ2

1) Evaluation protocol: The purpose is to examine the sensitivity to the hyperparameters of BMSC algorithm compared to that of DBSCAN and Mean Shift individually. For each hyperparameter, we firstly specified the range of potential values. The other hyperparameters were then fixed, and the algorithm was performed for each possible value, recording the extracted microservices. The outcomes were then reviewed using multiple metrics, and the metric values were plotted at each step. We focused on the DayTrader monolithic project for our investigation since it is a well-established benchmark for this topic.

- **Bandwidth :** using the estimate bandwidth function from the scikit-learn we estimate the maximum value of the kernel bandwidth, and then we varied the values of the hyperparameter from 0 to this estimated value.
- **Eps:** We varied the Epsilon values from 0 to 1 with a step equal to 0.05.

2) Results: Figure 4 showcases an evaluation of five different techniques, represented as subfigures. The Y-axes in each subfigure indicate the metric scale, while the X-axis displays the boxplot results for each technique arranged in the following order:

1) BMSC_eps: BMSC results varying its epsilon hyperparameter.
2) DBSCAN_eps: DBSCAN results varying its epsilon hyperparameter.
3) BMSC_band: BMSC results varying its bandwidth hyperparameter.
4) MeanShift_band: Mean shift results varying its bandwidth hyperparameter.

Each subfigure in Figure 4 is dedicated to a specific evaluation metric, allowing for a thorough comparative analysis of various aspects of the techniques to derive insights regarding

TABLE IV: EVALUATION RESULTS OF DAYTRADER APPLICATION USING BMSC ALGORITHM

Metrics	Option 1	Option 2	Option 3	Option 4	Option 5	Option 6
SM	0.3696	0.3435	0.3887	0.4697	0.40545	0.4052
IFN	1.0344	1.250	1.0370	0.9677	1.0769	1.318
ICP	0.6500	0.591	0.618	0.6432	0.6257	0.639
NED	0.7241	0.6666	0.7037	0.7419	0.6538	0.636
# microservices	29	24	27	31	26	22
size of the largest micro	13	13	13	13	16	17

their stability. The subfigures are arranged in the following order: SM, IFN, ICP, and NED, with each focusing on the evaluation of the corresponding metric for each technique. The last subfigure provides the results for the number of generated microservices per technique, represented as "# microservices". By carefully examining these evaluation metrics, we can gain a comprehensive understanding of the performance and stability of the techniques.

Upon analyzing Figure 4, it becomes clear that BMSC exhibits greater sensitivity compared to DBSCAN in relation to the epsilon hyperparameter (BMSC_eps vs. DBSCAN_eps), as well as greater sensitivity compared to Mean Shift in relation to the bandwidth hyperparameter (BMSC_band vs. Mean shift_band) across all evaluated metrics.

In the analysis of Figure 4, several noteworthy observations can be made. Firstly, despite DBSCAN outperforming BMSC in terms of the structural modularity (SM) and interface number (IFN) metrics, it results in a significantly high number of microservices. With an average of 115 microservices for an application containing only 118 classes, this outcome does not align with our migration goals. This discrepancy suggests that DBSCAN may be suffering from the "boulders and grains" problem, generating microservices that are either too small or too large. Such an outcome fails to address the limitations of the monolithic application and does not contribute to the desired loosely coupled microservices architecture.

On the other hand, Mean Shift exhibits better performance in terms of the number of generated microservices. Its mean number of microservices is comparable to that of BMSC, indicating a more balanced decomposition approach with fewer than 20 microservices on average. This suggests that Mean Shift provides a more suitable solution for achieving the desired granularity in the migration process of monolithic applications.

Furthermore, the analysis reveals that BMSC demonstrates greater sensitivity when varying the epsilon hyperparameter compared to the bandwidth hyperparameter (BMSC_band vs. BMSC_eps), as evident from the boxplots in the final subfigure. This sensitivity is also apparent in the boxplot variation of the non-extreme distribution (NED) subfigure, where the variation in epsilon results in up to a 60% change. This discrepancy can be attributed to the fact that varying the bandwidth can generate different modes that are connected using DBSCAN, whereas varying the epsilon hyperparameter directly affects the final number of microservices, as indicated by the comparison of variations in the number of microservices.

In contrast to the findings in [13], our analysis suggests that for our case, BMSC is more susceptible to the selection of its hyperparameters, specifically the epsilon parameter, compared to DBSCAN and Mean Shift when used independently. However, BMSC demonstrates greater consistency in the resulting decompositions across hyperparameter variations.

IV. Related Work

The first component of a decomposition approach is concerned with the type of input and how it is handled. The methods suggested by MSExtractor [20], Bunch [21], and [22] , for example, take as input the source code of a monolithic system and apply various static analysis techniques to it. The approach called HierDecomp [17], employes in addition the semantic similarity generated from the code text analysis. Other approaches, such as Mono2Micro [19], FoSCI [23], and COGCN [24], are based on the study of monolithic system use cases and execution traces. Sellami and al [25] combine both static and dynamic analysis in order to cover the individual disadvantages of each of the analysis approaches. There are, on the otehr hand, systems that employ different inputs, such as MEM [26], which analyses the git commit history of monolithic programs.

Most methods utilize clustering algorithms, such as [22] which feeds vectors derived from code embedding into an Affinity propagation clustering process [27]. The similarity metrics computed by an agglomerative single-linkage clustering method [28] are used by Mono2Micro [19]. Based on the graph it developed, MEM [26] provides its own clustering mechanism. Based on the similarity metrics, HierDecomp [17] and HyDecomp [25] employ a DBSCAN [16] density based clustering algorithm which ends by having a hierarchical microservices decomposition recommendation. Some methods suggest search algorithms to accomplish their goal. MSExtractor [29] uses the non-dominated sorting genetic algorithm (NSGA-II) [30] whereas FoSCI [23] employs both NSGA-II and hierarchical clustering. A community discovery method is used by Service Cutter to provide a decomposition.

However, many existing approaches encounter the challenge known as the "boulders and grains" problem, which arises when microservice decompositions lean towards being excessively large or overly small. Both situations introduce drawbacks in terms of system architecture and management.
When a microservice decomposition becomes too large, it can lead to heightened complexity and diminished modularity. Large microservices that encompass numerous classes or functionalities become cumbersome to maintain, understand, and update. Furthermore, even minor changes to a component within a large microservice may necessitate redeploying the entire service, impeding agility and scalability.
Conversely, when a microservice decomposition is excessively small, it can result in an abundance of services and unnecessary network communication overhead. Microservices consisting of only a few classes can lead to an excessively fragmented architecture, resulting in increased latencies and

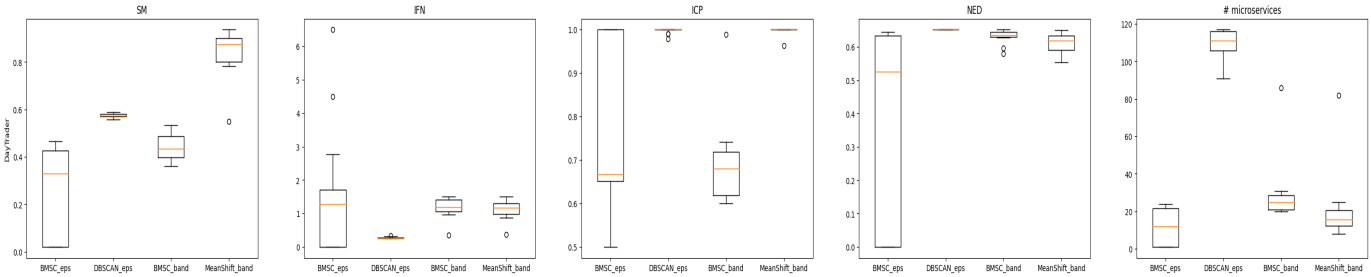

Figure 4: Evaluation metrics for different hyperparameters values when extracting microservices from the project DayTrader.

added complexity in managing the interactions among numerous small services.

Assessing whether a microservice decomposition is too large or too small requires careful evaluation. Qualitative factors such as complexity, cohesion, and adherence to the Single Responsibility Principle offer valuable insights into the size of a microservice. Additionally, quantitative metrics can be employed to measure microservice size, such as counting the number of classes or lines of code it encompasses. For example, "Mono2micro" paper [19] suggests a guideline for microservice size, recommending that an optimal microservice consists of 5 to 20 classes. This quantitative threshold aims to strike a balance, ensuring that microservices remain manageable and cohesive without succumbing to excessive granularity or complexity.

V. Conclusion and Future Work

In conclusion, this paper has presented a comparative study of different strategies for decomposing monolithic applications into microservices. Our proposed approach, utilizing the BMSC algorithm, effectively groups semantically and structurally similar classes to extract potential microservices. Notably, our approach demonstrates promising results by solely utilizing select characteristics of the monolithic application's source code as input, distinguishing it from approaches that require additional data sources.

Through extensive evaluation using various performance metrics, we have compared our approach with two well-established algorithms in the field. The experimental results highlight the superior cohesion within microservices, reduced interactions between microservices, and overall improved stability achieved by our method. However, it should be noted that the sensitivity to the epsilon hyperparameter remains a limitation, posing challenges in its selection.

Looking ahead, our future work will focus on developing more refined metrics to evaluate the extracted microservices and conducting comparative analyses against existing decomposition techniques. We also aim to explore different similarity metrics and investigate alternative types of interactions between classes beyond direct method calls. To further enhance the granularity of our approach, we intend to extend it to consider methods or functions of the monolithic application as a basis for decomposition, going beyond class-level granularity. Additionally, we recognize that static analysis alone may not provide a comprehensive understanding of application functionalities and interactions during runtime. Therefore, we propose exploring hybrid solutions that incorporate dynamic analysis of the source code to enrich the decomposition process.

By addressing these avenues for future research, we aim to advance the field of microservice decomposition and contribute to the development of effective and scalable approaches for migrating monolithic applications to microservices architectures.

References

[1] "sample.daytrader7," 2023-04-24. [retrieved: May, 2023].

[2] 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, 2020.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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," vol. 1073, pp. 190–214, Springer Verlag, 2019.

[13] 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.

[14] 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, 2019.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] K. Sellami, M. A. Saied, A. Ouni, and R. Abdalkareem, "Combining static and dynamic analysis to decompose monolithic application into microservices," 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.

[26] 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.

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

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

[29] 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.

[30] 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.