A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications

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

Ali Ouni
ETS Montreal, University of Quebec
Montreal, QC, Canada
ali.ouni@etsmtl.ca

ABSTRACT

The microservices architectural style offers many advantages such as scalability, reusability and ease of maintainability. As such microservices has become a common architectural choice when developing new applications. Hence, to benefit from these advantages, monolithic applications need to be redesigned in order to migrate to a microservice based architecture. Due to the inherent complexity and high costs related to this process, it is crucial to automate this task. In this paper, we propose a method that can identify potential microservices from a given monolithic application. Our method takes as input the source code of the source application in order to measure the similarities and dependencies between all of the classes in the system using their interactions and the domain terminology employed within the code. These similarity values are then used with a variant of a density-based clustering algorithm to generate a hierarchical structure of the recommended microservices while identifying potential outlier classes. We provide an empirical evaluation of our approach through different experimental settings including a comparison with existing human-designed microservices and a comparison with 5 baselines. The results show that our method succeeds in generating microservices that are overall more cohesive and that have fewer interactions in-between them with up to 0.9 of precision score when compared to human-designed microservices.

CCS CONCEPTS

• **Software and its engineering** → **Software architectures**.

KEYWORDS

Microservices, Clustering, Legacy decomposition, Static Analysis

ACM Reference Format:
Khaled Sellami, Mohamed Aymen Saied, and Ali Ouni. 2022. A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications. In *The International Conference on Evaluation and Assessment in Software Engineering 2022 (EASE 2022), June 13–15, 2022, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3530019.3530040

1 INTRODUCTION

The microservices architecture is a set of finely grained and lightweight components that each define a specific business need and that interact with each other through a RESTful API [2, 19, 20]. It offers multiple advantages such as scalability, independent development, reusability, maintainability and compatibility with cloud technologies [22]. Due to these benefits, many software companies have been migrating their legacy monolithic applications into microservices based software such Netflix, eBay, Amazon, IBM, etc. However, this process has proven to be difficult and costly [11].

As the demand for scalable, available and interoperable software has increased, the monolithic architectural style was not enough to answer the needs of modern applications [5, 22]. Hence, the microservices architectural design has become a commonly preferred solution when developing new software applications [17] .

One of the major challenges encountered in this endeavor is identifying how to split the components of the monolithic application. These components are more often than not highly cohesive and tightly coupled due to the nature of this architectural design. Another challenge lies within the subjectivity of the process which relies heavily on the opinions of the software experts handling this task. For applications with older frameworks or programming languages, developers that are familiar with the code base are hard to retain due to turnover which adds further to the difficulty of this task [3, 9].

For these reasons, prior research has attempted to develop automated tools for extracting a microservices architecture from a monolithic application [1, 7, 14, 15, 17]. Specifically, the main focus of the problem is identifying the different components that can constitute separate microservices. As such, this problem is handled as a clustering problem where the objective is extracting different clusters (i.e., microservices) from the sample elements (the application's components) . The existing approaches can be split into two categories based on whether they use static [1, 15, 17] or dynamic [7, 12, 13] analysis of the legacy applications. Most of these solutions utilize clustering-based algorithms or evolutionary algorithms. However, these tools often require the number of candidate microservices as input which is on its own a difficult task that requires the calibration of expert developers.

In this paper, we propose a novel microservices extraction approach that utilizes a Hierarchical DBSCAN algorithm [18] based on the static analysis of the legacy applications. More specifically, we combine structural and semantic similarity analysis between classes from the source code and then apply the Hierarchical DBSCAN algorithm in order to obtain a set of candidate microservices. The main advantage of using a density-based clustering algorithm is its ability to identify outliers which represent in this case classes that should be removed or refactored in the microservices architecture. Moreover, the Hierarchical DBSCAN not only provides the extracted microservices, but generates as well a hierarchical view of these microservices which would facilitate their customization and understanding their functional role.

In order to evaluate our approach, we compared the extracted microservices with microservices designed by human developers and reviewed how similar they are for 3 projects. The median precision score for each project surpassed 0.65 and had a maximum value above 0.9 in 2 of them. We compared as well the quality of the extracted microservices with those generated by other state-of-the-art approaches using 4 different metrics and 4 projects of varying complexity. Our solution achieved the best overall Structural Modularity scores for 3 projects and the best Inter-Call Percentage and Interface Number scores for all projects.

The main contributions of this paper are as follows:

(1) The formulation of microservices extraction as hierarchical clustering combining structural and semantic analysis. Our solution provides a hierarchical view of the potential microservices and detects possible outliers.
(2) Introducing novel metrics that enable the comparison of different microservices decompositions.
(3) An Empirical evaluation on our approach's ability to extract microservices and a comparison with state-of-the-art solutions.

The paper is organized as follows. We present the work related to this research in section 2. Then, we present our proposed solution in section 3. In section 4, we showcase the empirical evaluation of our approach. Afterwards, we discuss the results this work in section 5. Finally, we conclude the paper in section 6 and we outline our future work.

2 RELATED WORK

Most solutions in the literature that tackle problems similar to microservices extraction can be split into two distinct steps.

The first step revolves around the type of input that is fed to the solution and how it is processed. For example, the method proposed by MSExtractor[17], bunch[15] and [1] take in as input the source code of the monolithic application on which they apply different static analysis techniques. More specifically, both MSExtractor and bunch build call graphs that encode the interactions between the classes in these systems. The method mentioned in [1] converts the source code into a set of Abstract Syntax Trees which are fed to a code embeddings model [4]. The intuition behind the use of static analysis and more specifically the source code structure is that structurally similar classes or functions should be grouped together.

Other solutions such as Mono2Micro[13], FoSCI[12] and CO-GCN[7] are based on the analysis of the use cases and execution traces of the monolithic systems. These solutions aim to group together classes or functions that interact together at runtime for each business need provided with the input. Mono2Micro measures similarity metrics between the classes based on the execution traces. CO-GCN utilizes the execution traces to build matrices as features for its inference architecture. These methods of analysis are not mutually exclusive as they can be combined for better results. For example, MSExtractor analyzes the domain terms included in the classes in addition to the structural analysis with the assumption that each microservices should contain classes with similar domain concepts. Another example is CO-GCN which, in addition to using execution traces, uses the source code of the input application in order to build its model's architecture.

On the other hand, there are solutions that use different inputs such as MEM[14] which analyzes the git commit history of the monolithic applications. This solution constructs a graph from the git history that encodes the similarity between the classes.

The second step of each solution takes in as input the data processed in the previous step and applies on it an algorithm in order to generate the decompositions. Most solutions use clustering algorithms like in [1] which uses the vectors obtained from the code embeddings as input to an Affinity propagation [10] clustering algorithm. Mono2Micro[13] uses the similarity metrics it measured with an agglomerative single-linkage clustering algorithm [21]. MEM[14] introduces its own clustering algorithm based on the graph it generated.

Some solutions propose search algorithms in order to achieve their task. MSExtractor[17] uses the non-dominated sorting genetic algorithm (NSGA-II) [6] while FoSCI[12] uses both NSGA-II and hierarchical clustering on the execution traces. Bunch[15] on the other hand uses a different approach which applies a hill-climbing algorithm.

3 PROPOSED APPROACH

In this section, we detail our proposed approach for extracting candidate microservices from a given monolithic application.

3.1 Problem Formulation

The objective of our approach is to identify a set of microservices when a given a legacy monolithic application which is characterized by the set of its classes. We assume that each class can only be contained within one microservice. As such, we formulate our problem as a clustering task where the microservices correspond to clusters and the classes define the samples.

The input of the solution is the legacy application which is defined as a set of classes $C = \{c_1, c_2, c_3, \ldots, c_N\}$ where N is the number of classes in the system. The output of the solution is $M = \{m_1, m_2, m_3, \ldots m_K\}$ where K is the number of the microservices selected by the solution. The output can be represented as a vector $X = [x_1, x_2, x_3, \ldots x_N]$ of size N and where x_i = j means that class c_i is contained in microservice m_j.

We also define outlier classes which are too dissimilar from other classes and hence, do not belong to any specific cluster. consequently, we define $x_i = -1$ which means that class c_i is an outlier.

Figure 1 showcases a simple example of a monolithic application that consists of 5 classes. The second part of the Figure represents a potential decomposition which contains two microservices and an outlier class. In this case, the vector X is represented as $X = [0, 1, 1, 0, -1]$.

In order to apply the clustering algorithms, we need to define a distance function between each class c_i and c_j. In the next subsection, we detail this distance function.

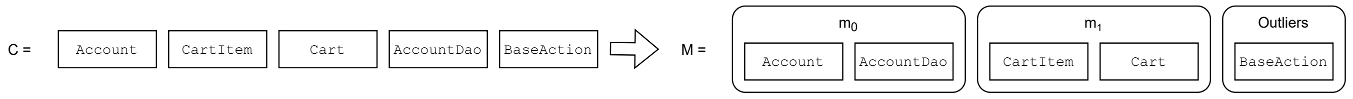

Figure 1: An example showcasing a potential decomposition for an application of 5 classes.

3.2 Representation of the Monolithic Application

As mentioned in the previous section, most of the state-of-the-art solutions [1, 15, 17] rely on the static analysis of the source code of the monolithic application in order to extract meaningful patterns. In our approach, we utilize a static analysis of the source code in order to extract the relations and dependencies between the different classes. We define two distinct types of similarities between classes as follows:

Structural Similarity (Sim_{str}): This metric is based on the shared number of method calls between 2 classes. It encodes the dependency between them and as such evaluates the similarity from a functional point of view [17]. The objective of grouping together classes based on this similarity is to obtain microservices that are cohesive from the implementation perspective. For two given classes c_i and c_j, the structural similarity is defined as follows:

$$Sim_{str}(c_i, c_j) = \begin{cases} \frac{1}{2}\left(\frac{calls(c_i,c_j)}{calls_{in}(c_j)} + \frac{calls(c_j,c_i)}{calls_{in}(c_i)}\right) & if\ calls_{in}(c_i) \neq 0\ and\ calls_{in}(c_j) \neq 0 \\ \frac{calls(c_i,c_j)}{calls_{in}(c_j)} & if\ calls_{in}(c_i) = 0\ and\ calls_{in}(c_j) \neq 0 \\ \frac{calls(c_j,c_i)}{calls_{in}(c_i)} & if\ calls_{in}(c_i) \neq 0\ and\ calls_{in}(c_j) = 0 \\ 0 & otherwise \end{cases} \tag{1}$$

where $calls(c_i, c_j)$ represents the number of times a method of the class c_i has called a method from the class c_j. On the other hand, $call_{in}(c_i)$ is the number of calls incoming to c_i.

The values of $Sim_{str}(c_i, c_j)$ are in the range [0,1] where 1 indicates classes c_i and c_j are very similar and used exclusively together and 0 indicates that they are completely independent.

Semantic Similarity (Sim_{sem}): This metric utilizes natural language processing in order to measure how related are the domain semantics of the two given classes [17]. Given that a microservice should provide a specific function and/or a single domain use case, we need to identify classes that serve similar domain use cases. Assuming the monolithic projects were coded using standard business practices where class, method and variable names reflect business concepts and comments describe their function, analyzing the terminology used within these components can be a powerful tool for extracting the domain significance of each class.

Hence, each class is defined by the set words in its comments, parameter names, field names, method name and variable names. Each word is preprocessed includig splitting using CamelCase, filtering out stop words, and stemming. In the CamelCase splitting phase, we take as input the method, variable and member names and split them into different words assuming that the CamelCase naming convention is used. For example, the word CamelCase would become a list: [camel, case]. Next, we filter out stopwords. Finally, we apply stemming which removes parts of the words in order to remove the impact of conjugation. For example, the words 'stemming' and 'stemmed' would become 'stemm'. The final result is a vector of size n_V where n_V is the number of words in the domain vocabulary extracted from the monolithic application. This vector is measured using a TF-IDF (Term Frequency-Inverse Document Frequency) model [16]. The Semantic Similarity metric is defined as the cosine similarity between two classes:

$$Sim_{sem}(c_i, c_j) = \frac{\vec{c_i} \cdot \vec{c_j}}{\|\vec{c_i}\| \, \|\vec{c_j}\|} \tag{2}$$

where $\vec{c_i}$ and $\vec{c_j}$ are the TF-IDF vectors of class c_i and class c_j and $\|\vec{c_i}\|$ is the Euclidian norm of vector $\vec{c_i}$.

The values of Sim_{sem} range between 0 and 1 where the value 1 signifies that both classes use the exact same vocabulary.

Class Similarity (CS): The previous similarity metrics represent different aspects of the relations between classes. These two metrics do not necessarily correlate and as such using only a single one of them does not guarantee satisfying the other. For these reasons, we use the Class Similarity metric [17] which represents a weighted sum between the Structural similarity and Semantic similarity of two given classes:

$$CS(c_i, c_j) = \alpha \, Sim_{str}(c_i, c_j) + \beta \, Sim_{sem}(c_i, c_j) \tag{3}$$

where $\alpha \in [0, 1]$, $\alpha + \beta = 1$ and $CS \in [0, 1]$

3.3 Clustering Algorithms

Given the Class Similarity metric, we can apply a clustering algorithm to extract the microservices. More specifically, we utilize the DBSCAN algorithm [8].

DBSCAN is a density based clustering algorithm where the aim is to group together points that are densely packed in the search space and identify noisy points which do not fit into any clusters using two main concepts which are the neighborhood distance and the minimum number of points in a neighborhood [8].

The advantage of using DBSCAN in this case is that defining the number of target microservices is no longer a requirement since the algorithm identifies on its own the existing clusters. The second advantage is the minimum number of sample hyper-parameter which enables the addition of minimum number of classes per microservice constraint. This constraint helps with preventing the extraction of extremely small microservices which usually do not have a practical significance. However, this parameter does not help with dealing with extremely large microservices. The final advantage of applying this algorithm is the detection of noisy points which we consider as outlier classes. They are classes that are not similar enough to a minimum number of classes in original application and as such cannot be included in any of the extracted microservices.

However, DBSCAN is still constrained by another hyper-parameter which is the neighborhood distance ϵ. The resulted number of extracted microservices, the number of outlier classes and the size of clusters are heavily dependent on this hyper-parameter. For this reason, we used in our solution the variant of the DBSCAN algorithm defined in [18] which was called ϵ-DBSCAN.

The first step in this algorithm is to cluster the input data using DBSCAN with an ϵ value equal to 0. This generates the initial cluster which contains identical points for a given metric. Afterwards we increment ϵ with a certain value that is defined within the input. We apply DBSCAN once more but with the new ϵ value which generates a new layer of clusters that contain within them the previous clusters as well as potentially new points. We continue this step until ϵ is equal to MaxEpsilon which is defined by the user.

Although ϵ-DBSCAN has the MapEpsilon hyper-parameter which influences the final layer of the extracted microservices, the main advantage of this algorithm is the hierarchical nature of its results. The output of ϵ-DBSCAN is a set of ordered layers that correspond each to the microservices extraction at an epsilon step. This output can be used to visualize the evolution of microservices and their classes at each step and allows for the differentiation between very similar classes and those at the edges of the microservices. For a domain expert, this can be a powerful tool to discover outlier classes as well and the optimal point to define the extracted microservices.

Figure 2 summarizes the process executed in order to extract the microservices using our approach. Through this Figure, we can see that as we take the source code of input, two tasks start in parallel. In the first task, we use the Abstract Syntax Trees to build the call graph that encodes the interactions between the classes. Afterwards, we generate the Structural similarity matrix. The second task extracts the comments, method, parameter and variable names within the classes, applies the Natural Language pre-processing like CamelCase splitting, stop-word removal and stemming and generates the TF-IDF vectors. These vectors are used to measure the semantic similarity matrix. By combining the output of both of these tasks, we generate the class similarity matrix and start the extraction task with ϵ-DBSCAN. The output of this phase is a set of decompositions that reflect the output of DBSCAN at each epsilon step. The final layer represents the extracted microservices while the rest can be used for in-depth analysis and manual customization of the microservices.

4 EVALUATION

In this section, we evaluate the effectiveness of our approach in extracting the proper microservices. We first describe the research questions we are seeking to investigate. Thereafter, we showcase and discuss the experimental results.

4.1 Research Questions

We designed our experimental study to answer the following research questions (RQs):

- **Q1:** How well do the extracted microservices compare to those that were manually identified by software engineers.
- **Q2:** What is the impact of various experimental settings on the extracted microservices quality?
- **Q3:** How does our solution perform when compared with state-of-the-art microservices extraction baselines?

Table 1: Metadata of the Microservice-based projects.

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

4.2 Evaluation and results for RQ1

4.2.1 Evaluation protocol. To answer RQ1, we selected 3 Open-Source microservices-based Java projects with different degrees of complexity, namely *Spring PetClinic*[1], *Microservices Event Sourcing*[2] and *Kanban Board demo*[3]. The details of these projects can be viewed in the Table 1.

To evaluate our approach, we need to define measures that can compare two given sets of classes that represent the set of extracted microservices and the ground truth microservices. It is worth noting that the encoding of the clusters for any given decomposition solution can be different. Let's take the application shown in Figure 1 as an example. Let's suppose that the original set of microservices was $X_t = [0, 0, 1, 1, 2]$. If we get a decomposition denoted by $X_1 = [1, 1, 2, 2, 0]$, we will have two vectors that are different but encode the same decomposition. This issue becomes more complicated if the vectors have different microservices (for example $X_2 = [0, 1, 1, 1, 2]$). In such a case, it is even more ambiguous which microservices correspond to each other.

To overcome this challenge, we first need to identify the corresponding ground truth microservice for each extracted microservice. As such, we define the corresponding microservice as the microservice that has the largest number of common classes with the extracted microservice. We introduce the function 4 that given an extracted microservice m_i and a different set of microservices M (set of ground truth microservices), $Corr(m_i, M)$ selects the ground truth microservice with the largest number of common classes with the extracted microservice.

$$Corr(m_i, M) = \underset{m_j \in M}{argmax}\left(\frac{|m_i \cap m_j|}{|m_i|}\right) \tag{4}$$

We then can calculate the precision metric, defined in equation 5, which is the mean of the percentage of the correctly identified classes out of the total number of identified classes for each extracted microservice.

$$precision = \frac{1}{|M|} \times \sum_{\forall m_i \in M} \frac{|m_i \cap Corr(m_i, M_t)|}{|m_i|} \tag{5}$$

where M is the set of the extracted microservices, M_t is the set of the original or ground truth microservices and $Corr(m_i, M_t)$ is the microservice from M_t that corresponds to the extracted m_i .

We also calculate as well the Success Rate (SR) that measures the percentage of successfully retrieved microservices based on the precision metric:

$$SR = \frac{1}{|M|} \times \sum_{\forall m_i \in M} matching(m_i, Corr(m_i, M_t)) \tag{6}$$

[1]https://github.com/spring-petclinic/spring-petclinic-microservices
[2]https://github.com/chaokunyang/microservices-event-sourcing
[3]https://github.com/eventuate-examples/es-kanban-board

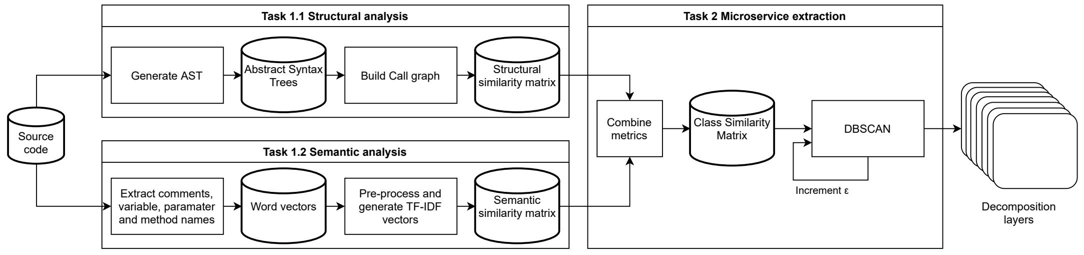
Task 1.1 Structural analysis
Generate AST
Abstract Syntax Trees
Build Call graph
Structural similarity matrix
Source code
Task 1.2 Semantic analysis
Extract comments, variable, paramater and method names
Word vectors
Pre-process and generate TF-IDF vectors
Semantic similarity matrix
Task 2 Microservice extraction
Combine metrics
Class Similarity Matrix
DBSCAN
Increment ε
Decomposition layers

Figure 2: A summary of the steps taken to extract the microservices.

where *matching* is defined as:

$$matching(m_1, m_2) = \begin{cases} 1 & \text{if } \frac{|m_1 \cap m_2|}{|m_1|} \geqslant threshold \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

where m_1 and m_2 are two sets of classes and $threshold \in [0,1]$. So for a given $threshold = \frac{k}{10}$ we calculate the $SR@k$ of the corresponding k value.

For each of these test projects, we regrouped all of their Java classes together in order to simulate a Monolithic architecture. This version served as input for our solution while its original version was considered as the true decomposition with which we compare our results. For each of the mentioned projects, we generated multiple microservices decompositions with varying hyper-parameter values. Afterwards we calculate the metrics precision, SR@5, SR@7 and SR@9 based on these decompositions and the corresponding true decomposition.

4.2.2 Results. The Figure 3 showcases the results in the form of boxplots for each project and each metric. We can observe that the median precision for each project is within the range [0.6,0.7]. More specifically, both Kanban Board demo and Microservices Event Sourcing achieved precision values higher than 0.5 for all decompositions except for a single outlier. On the other hand, there's a large variability in the precision values obtained for Spring Petclinic. This difference shows that as the number of classes increases, the stability of the results improves. As for the success rate, we can observe that as we increase the threshold, the median and minimum scores achieved for each project drop but we can as well see that for Kanban Board demo and Microservices Event Sourcing, the values have less variance. Additionally, there is not a significant difference between the scores for SR@7 and SR@9. These results suggest that for most decompositions, a high percentage of their microservices achieve a precision score higher than 0.5 and that a significant number among them have a high precision score that exceeds 0.9. We can see as well the same pattern of the variance of the results decreasing when the size of the project increases.

The microservices extracted for the project Microservices Event Sourcing seem to have overall lower scores than the other projects. The most likely explanation, however, stems from the fact that this projects does not use a single natural language for its domain terms unlike the other projects which utilize only the English language. This serves as an additional roadblock for this project. Nonetheless, the results achieved are comparable to the others and the precision values are in the same range as the others.

As an example, we focus on a decomposition of the project Spring PetClinic which is an online platform that provides veterinary services that was implemented with a microservices architecture in mind. Given the size of the project, the Figure 4 shows a subset of the microservices obtained from one of the decompositions. In this Figure, the ellipses represent the names of some of the original microservices while the large white rectangles represent the new microservices. Within them, we find the classes which are colored based on their original microservices.

The original microservice mostly implement domain specific concepts like customers, visits and Veterinaries. Other microservices like `ApiGateway`, `DiscoveryServer`, `ConfigServer` and `AdminServer` implement technical requirements for the used platform. More specifically, the `ApiGateway` microservice includes a large range of utility classes as well as data classes that have to mirror their counterparts in the other microservices. For this reason, we can observe some classes like Visits which are included twice.

We can see that the microservice m_2 contains 6 classes that were originally in the Vets service and that represent utility classes related to the `Veterinary` domain concept which shows an example of semantically similar classes being grouped together. A second example for such a case is the microservice m_3 which contains 4 classes, 3 of which were originally within the Customers microservice and that represent the Owner concept. We can see as well that the 4th class in this case represents as well the Owner concept despite being from a different microservice originally. The microservice m_4 shows an example of two classes from different microservices but that mirror the same class being grouped together. In this case, they are the `Visits` domain object class from the visits microservice and the `Visits DTO` from `ApiGateway`. Finally, we can observe that the largest microservice m_1 is grouping multiple classes that represent two major domain concepts: `Pet` and `Visit`. Most of the classes in relation to the visits concept in the `Visits` microservice and the `ApiGateway` microservice were included in m_1. Similarly, most of the concepts in relation to the `Pet` service in the `customers` microservice and the `ApiGateway` microservice were included as well.

Through this example, we can observe the hierarchical structure of the microservices. the microservice m_1 contains in fact 3 potential microservices which are $m_{1.1}$, $m_{1.2}$ and $m_{1.3}$. In this example, our proposed approach extracted initially these 3 microservices with highly similar classes. Since the similarity values in-between these microservices are within the thresholds defined in the input, they were grouped together in a larger microservice that satisfies

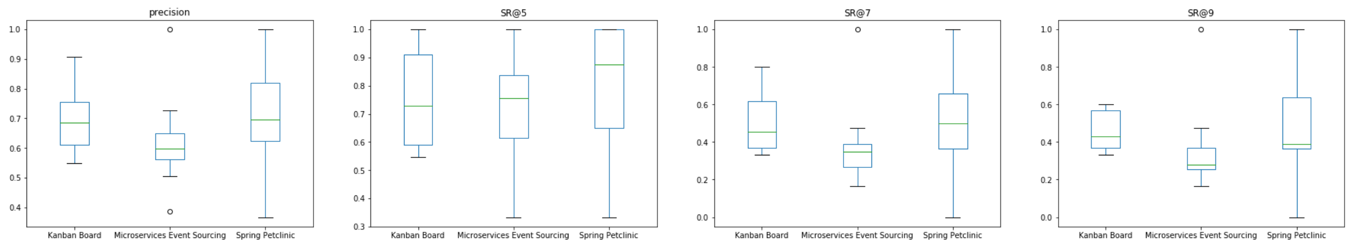

Figure 3: Boxplots showcasing the comparison between the generated decompositions and their corresponding true microservices.

the conditions defined at the start of the process. We can observe another hierarchy within the microservice m_2. In this case, $m_{2.1}$ and $m_{2.2}$ both contain classes that were originally from the Vets service.

The results obtained show that the extracted microservices achieve a median precision score around 0.65 with a maximum value surpassing 0.9 for some projects. Although the extracted microservices are not identical to the original microservices, we observed that they include nonetheless most of the classes from the human-built microservices. Additionally, the hierarchical structure of the results helps the developer with identifying alternative decompositions.

4.3 Evaluation and results for RQ2

4.3.1 Evaluation protocol. In the previous question, we observed that for certain projects, the used hyper-parameters can have a large impact on the quality of the result. For example, for Spring Petclinic, there was a large variance in the evaluation metrics. This research question focuses on analyzing the impact of the hyper-parameters of our proposed method on the quality of the final result. In order to address this question, we experiment with varying the values of the hyper-parameters MaxEpsilon, α and MinSamples.

- **MaxEpsilon** refers to the maximum value of the ϵ hyper-parameter that is fed to the DBSCAN algorithm during the ϵ-DBSCAN algorithm. Its values range from 0 to 1 in this case which correspond to the minimum and maximum possible distance between two classes for the metric CS defined in equation 3.
- **alpha** α, on the other hand, refers to the weight associated with the similarity value Sim_{str} when calculating CS in the formula 3. α values range from 0 to 1 where 0 indicates complete reliance on the Sim_{str} metric which means the extraction is exclusively based on the structural similarity of the classes. On the other hand, $\alpha = 1$ corresponds to an extraction process based on the semantic domain similarity between the classes so $CS = Sim_{sem}$.
- **MinSamples** is the minimum number of possible classes that can exist within a microservice. It is a hyper-parameter of the DBSCAN algorithm and in consequence the ϵ-DBSCAN algorithm.

For each hyper-parameter, we used the following process: First, we define the range of possible values and the iteration step. Then, we fix the rest of the hyper-parameters. Afterwards, we run our approach for each possible value and we record the extracted microservices. Next, we evaluate the obtained results using different metrics. Finally, we plot the metric values at each step. The following analysis is focused on the Monolithic project JPetStore as it serves as a standard benchmark for this problem and an ideal example to study. Details regarding this project can be viewed in the Table 2.

In order to analyze different aspects of the obtained decompositions without the need for the ground truth microservices, we use the following metric:

- **Structural Modularity (SM)**: [12] is a metric that combines the measures for the cohesion in each microservice and the coupling between different microservices in order to evaluate the structural quality of the obtained microservices. It is defined as follows:
$$SM = \frac{1}{K}\sum_{i=1}^{K}\frac{\mu_i}{m_i^2} - \frac{1}{(K(K-1))/2}\sum_{i\neq j}^{K}\frac{\sigma_{i,j}}{2\,m_i\,m_j} \tag{8}$$
Where K is the number of the extracted microservices, μ_i is the number of unique calls between the classes in microservice i, m_i is number of classes in microservice i and $\sigma_{i,j}$ is the number of unique calls between classes of microservice i and classes of microservice j.
Higher values of SM reflect higher cohesiveness and lower coupling so better structural quality.
- **Interface Number (IFN)**:[17] is a metric for measuring the number of interfaces within a list of extracted microservices. It is defined as:
$$IFN = \frac{1}{K}\sum_{i=1}^{K}|I_i| \tag{9}$$
Where K is the number of the extracted microservices, I_i is the set of interface classes within microservice i. An interface class in this case is a class that has been called by a class in an external microservice.
Lower values of IFN indicate a better result.
- **Non-Extreme Distribution (NED)**:[13] One of the most notable problems encountered in microservice extraction is the Boulder and Grain problem where solutions tend to output microservices that are extremely large (Boulders) or extremely small (Grain). This metric enables the detection and evaluation of such cases. It is defined as:
$$NED = 1 - \frac{|\{m_i\ ;\ 5 < |m_i| < 20, i \in [1, K]\}|}{K} \tag{10}$$

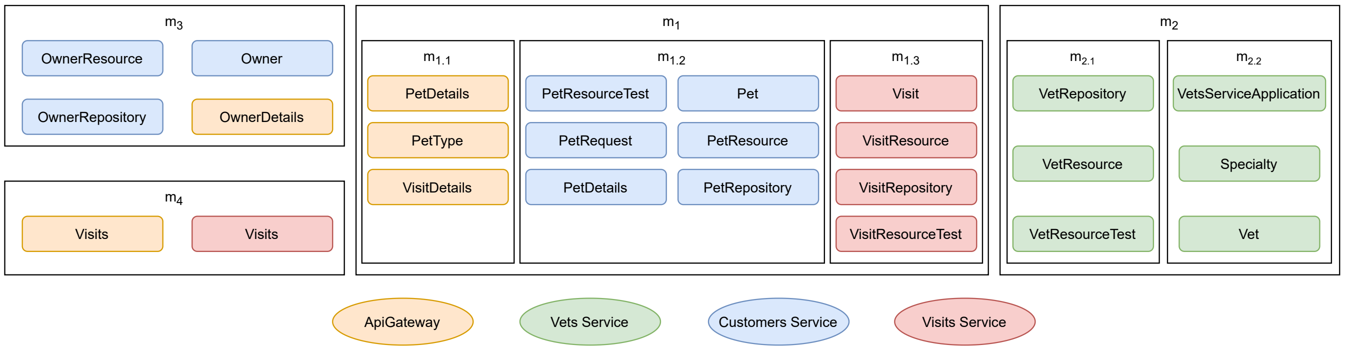

Figure 4: A subset of the microservices obtained from a decomposition of the project Spring PetClinic.

Where K is the number of the extracted microservices and $|m_i|$ is the size of microservice m_i.
Lower values of NED indicate lower number of extreme microservices which corresponds to better results.

- **Inter Call Percentage (ICP)**:[13] is measured in the literature as the percentage of runtime calls between two microservices. In this case, we are measuring ICP as the percentage of static calls between two microservices. The objective of this metric is still the same which is evaluating the dependencies between the microservices or in other terms the coupling. It is defined as follows:

$$ICP = \frac{\sum_{i=1, j=1, i \neq j}^{K} icp(M_i, M_j)}{\sum_{i=1, j=1}^{K} icp(M_i, M_j)} \tag{11}$$

Where K is the number of microservices, M_i is the set of classes in microservice i and $icp(M_i, M_j)$ is defined as:

$$icp(M_i, M_j) = \sum_{c_k \in M_i} \sum_{c_l \in M_j} (log(calls(c_k, c_l)) + 1) \tag{12}$$

where $calls(c_k, c_l)$ is the number of calls from class c_k to class c_l.
Lower numbers of ICP indicate less interactions between the different microservices which represents a better result.

4.3.2 Results. The following subsection showcases the results when individually varying each hyper-parameter:

MaxEpsilon: We varied the MaxEpsilon values from 0 to 1 with a step equal to 0.05 . We fixed alpha to 0.5 and MinSamples to 2. The Figure 5 shows the results obtained for the project JPetStore. We can observe that for MaxEpsilon values below 0.5, all of the classes have been classified as outliers. On the other hand, we can see that as we increase MaxEpsilon, the SM value, which is plotted in blue, starts off by slightly increasing and then rapidly decreasing starting from the value 0.8 where we can observe a large spike in the values of IFN and ICP. The plots for these couple of metrics, which correspond respectively to the green and orange plots, share the same shape where they start out with low values, rapidly increase between 0.7 and 0.8 and finally decrease back to almost 0. As for NED, which is represented in red, starts with high values for the range of MaxEpsilon in [0.5, 0.6]. Then, it drops to 0.5 and starts increasing slowly back to 1 when MaxEpsilon ranges between 0.65 and 0.85. Finally, the NED values stay constant for the rest of the range of MaxEpsilon values.

Observing these metrics together, we can infer the explanation behind these plot shapes and select the ideal MaxEpsilon value for this project. MaxEpsilon is the hyper-parameter that controls how similar are the classes that we are grouping together to form the microservices. In other terms, ranging its values from 0 to 1 is akin to loosening the condition for grouping the classes.

So, not having any microservices for MaxEpsilon values between 0 and 0.5 shows that the condition is too strict in that case and the algorithm is unable to find any classes that satisfy it. As we continue to increase MaxEpsilon, the algorithm starts gradually grouping together some classes that satisfy the corresponding condition which explains the observation in the range [0.5, 0.6] where NED and SM are high due to having small but coherent microservices. For the values 0.65 and 0.7, we can observe higher SM values, lower NED values and low ICP and IFN values due to having larger but more balanced microservice and that still contain coherent classes. As for the values 0.75 and 0.8, the sharp increase in NED, ICP and IFN values and the sharp decrease of SM values suggest that the condition is now loose enough that outlier classes are added to the microservices which results in increased inter-microservices interactions and less coherent microservices. As for the rest of the values, the high NED value and the low values for the rest of the metrics suggest that the condition is too loose and the microservices have been grouped together which is to be expected for extreme MaxEpsilon values.

From this Figure, it is clear that the best MaxEpsilon values for the project JPetStore would be 0.65 or 0.7 where the trade-off between the metrics would be at its best. For the rest of the projects, we can observe a similar pattern as the one we described.

alpha α: For this experiment, we fixed MaxEpsilon to 0.7 based in the previous results and MinSamples to 2. We varied alpha from 0 to 1. Figure 6 shows the results obtained for the project JPetStore. We can observe that overall, SM and IFN values seem to be increasing when we increase alpha. On the other hand, ICP shows only a couple of spikes for alpha equal to 0.6 and 0.65. As for NED, its values stay equal to 1 for all alpha values except for the range [0.45, 0.55] where it drops to 0.25 and 0.45.

In this case, varying alpha from 0 to 1 is equivalent to relying less and less on the structural similarity and increasing the impact of

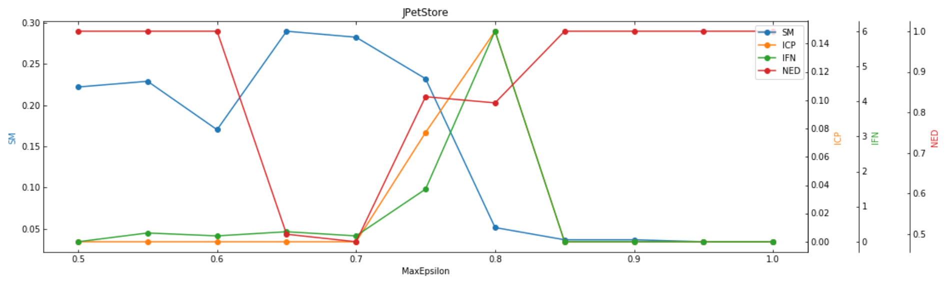

Figure 5: Evaluation metrics for different MaxEpsilon values when extracting microservices from the project JPetStore

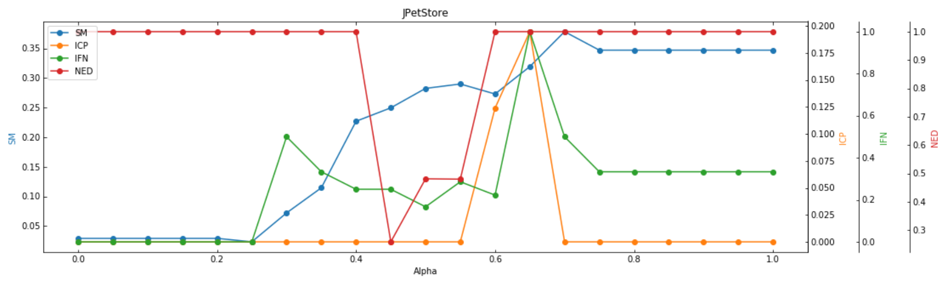

Figure 6: Evaluation metrics for different alpha values when extracting microservices from the project JPetStore

the semantic similarity between the classes. This can explain why ICP and IFN values are higher for high alpha values as relying more on the structural similarity will lead to the algorithm prioritizing grouping together classes that are more similar structurally and hence that communicate with each other which in turn lowers these metrics.

From this Figure, we can conclude that the best values for alpha should be in the range [0.45, 0.55] where we balance the impact of both similarity values. This was the case as well for the other projects where this range represented the base trade-off point between the different metrics.

MinSamples: In this case, we fixed MaxEpsilon to 0.7 and alpha to 0.5 and we varied the MinSample hyper-parameter in the range [1, 4]. The Figure 7 shows the results for the project JPetStore.

We see that ICP, IFN and NED decrease as we increase MinSample and that SM peaks at the value 2 and decreases afterwards. We observed similar results for the rest of the projects which suggests that the best value for this hyper-parameter is 2.

The optimal values for MaxEpsilon are in the range [0.65, 0.75]. For the hyper-parameter α, they are in the range [0.45, 0.55]. Finally, for MinSample, the optimal value is 2.

4.4 Evaluation and results for RQ3

4.4.1 Evaluation protocol. For this research question, we compared our solution with 5 baselines that tackle similar problems but with different methods which are Bunch[15], CoGCN[7], FoSCI[12], MEM[14] and Mono2Micro[13]. For each of these baselines as well as our solution, we measured the quality of the obtained microservices using the 4 metrics SM, IFN, ICP and NED.

Project	Version	SLOC	# of classes
JPetStore	1.0	3,341	73
DayTrader	1.4	18.224	118
Plants	1.0	7,347	40
AcmeAir	1.2	8,899	86

Table 2: Metadata of the Monolithic projects.

In order to compare these State-of-the-art solutions, 4 Open-source monolithic Java projects were used as subjects for testing and evaluating the performance of our approach which are JPetStore[4], DayTrader[5], Plants[6] and AcmeAir[7]. The Table 2 lists these projects as well as their metadata.

However, since each solution has at least one hyper-parameter that can significantly impact the quality of the final result, we extracted different microservices decompositions from each solution based on different values of their corresponding hyper-parameters. For example, Mono2Micro takes as input the number of target microservices, so we generated a different decomposition for each value of this hyper-parameter in [3,5,7,9,11] for the project JPetStore. As for our solution, given the results we observed in the previous section, we decided to fix alpha to 0.5 and MinSample to 2 since these hyper-parameters show less variability between projects when fixed in this range but we varied the values of MaxEpsilon between 0.5 and 0.9 with a 0.05 step.

[4]https://github.com/KimJongSung/jPetStore
[5]https://github.com/WASdev/sample.daytrader7
[6]https://github.com/WASdev/sample.mono-to-ms.pbw-monolith
[7]https://github.com/acmeair/acmeair

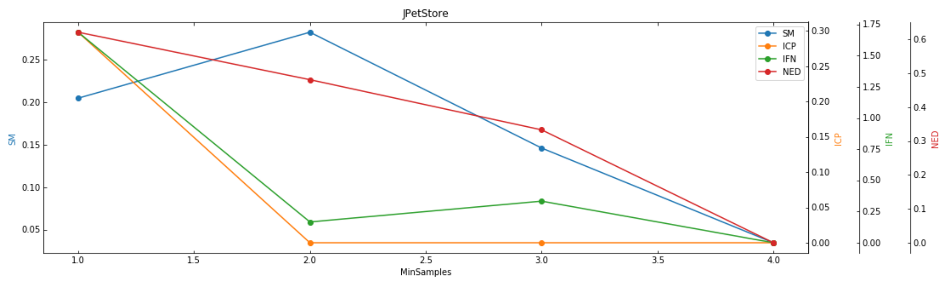

Figure 7: Evaluation metrics for different MinSamples values when extracting microservices from the project JPetStore

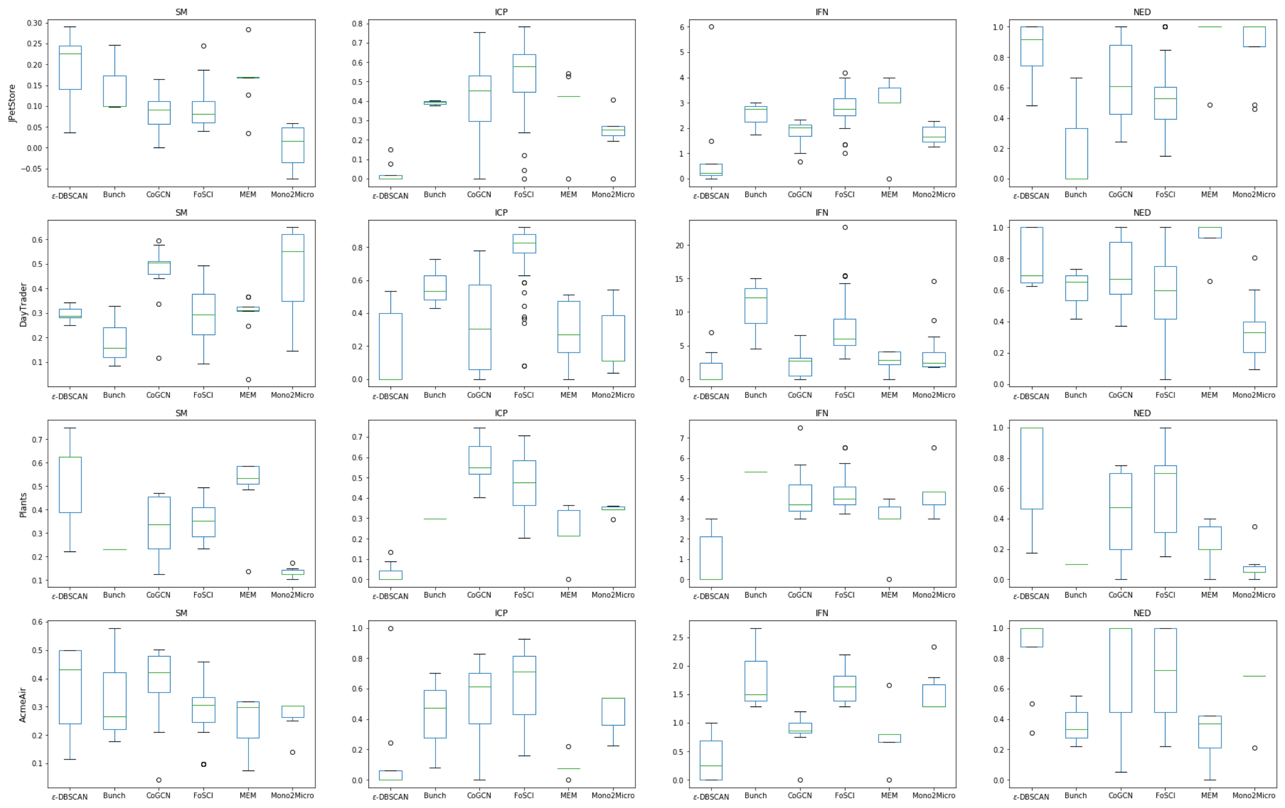

Figure 8: Boxplot representation for the results of each project/baseline/metric combination

4.4.2 Results. The Figure 8 shows the boxplot results for each project and metric combination.

Looking at the metric SM in the first column, we can see that our method outperforms the other baselines for all projects expect DayTrader where we can see that our results always have the highest mean and the highest maximum with one exception being Bunch having a higher maximum for the project AcmeAir. As for DayTrader, our solution outperforms Bunch and has comparable median values to Fosci and MEM.

As for the metrics ICP and IFN which are respectively represented in the second and third columns, we can observe that our solution consistently achieves better results than the other baselines with only one exception where for the project JPetStore we can see one outlier that has higher IFN score than the others. We can see as well that for most cases, the variance of these metrics for our solution is small which suggests that our method often results in decoupled microservices.

Finally, for the metric NED, our solution seems to outperform MEM and Mono2Micro for the project JPetStore. In the case of DayTrader, it outperforms MEM again and achieves comparable results to most of the rest with the exception of Mono2Micro. For Plant, we can observe a large variance in the results obtained which suggests that choosing the right MaxEpsilon value for this project can have a large impact on the final decomposition. In the previous research question, we recommend the optimal values for this parameter.

However, the role of the human experts remains crucial to validate and calibrate the extracted microservices, thus the hierarchical structure of the extracted microservices will be very helpful in this process. Overall, for this metric, we can see that Bunch achieves better NED scores than the rest.

The comparison results show that our solution achieves noticeably better results than the the baseline approaches for the metrics SM, IFN and ICP, but at the cost of higher NED values.

5 THREATS TO VALIDITY

The major threat to the validity of our experiments is related the external validity. Our empirical evaluation was based on a total of 7 open source projects with different architectures and sources. To better generalize the results of our approach, a larger number of projects should have been considered. To mitigate this issue, we selected projects with varying scales and from diverse origins. Future replications of our approach on other monolithic systems are needed. The second threat to validity lies within the qualitative evaluation. An alternative solution to evaluate the quality of the extracted microservices would be to involve experienced software engineers who are familiar with microservices-based software systems and the microservices extraction task to inspect the results and give their feedback. We are thus planning to further evaluate our approach with developers in an industrial setting as part of our future work.

As for internal threats to validity, it concerns factors that could influence our observations. The comparison of our solution with the different baselines was based on the results that were generated using specific hyper-parameters which could be an important internal threat to validity. In order to mitigate this issue, we based the results on multiple runs while varying hyper-parameter values for every solution. In a situation without any constraints, a better alternative would be to optimize the hyper-parameters for each solution and experimental setup. The used performance metrics represent a threat to validity. To mitigate this issue, we employed four different metrics that reflect different aspects such as the cohesion within the microservices, the interactions between the microservices and their granularity. Other metrics can also be considered such as the number of microservices per decomposition, number of classes per microservice and the domain modularity.

6 CONCLUSION AND FUTURE WORK

In this paper, we proposed a hierarchical clustering based approach to decompose a given monolithic application into a set of microservices. The presented approach groups together semantically and structurally similar classes using a modified density-based clustering algorithm and provides as a result a hierarchical structure of the potential microservices as well as outlier classes. We evaluated our approach using different performance metrics and compared it to multiple baselines. The experimental results show that our method achieved better cohesion within the microservices and less interactions between the microservices.

In the future, we plan on developing more fine-grained metrics for evaluating the extracted microservices and comparing them with existing decompositions. We plan as well on investigating the impact of separating utility classes from domain classes and reviewing methods on how to automatically identify them. In addition, we are interested in experimenting with different similarity metrics as well as different types of interactions between classes other than direct method calls.

REFERENCES

[1] Omar Al-Debagy and Peter Martinek. 2021. A Microservice Decomposition Method Through Using Distributed Representation of Source Code. *Scalable Computing* 22 (02 2021), 39–52.

[2] Nuri Almarimi, Ali Ouni, Salah Bouktif, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Mohamed Aymen Saied. 2019. Web service API recommendation for automated mashup creation using multi-objective evolutionary search. *Applied Soft Computing* 85 (2019), 105830.

[3] Nuri Almarimi, Ali Ouni, Moataz Chouchen, Islem Saidani, and Mohamed Wiem Mkaouer. 2020. On the detection of community smells using genetic programming-based ensemble classifier chain. In *15th International Conference on Global Software Engineering*. 43–54.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, Article 40 (jan 2019), 29 pages.

[5] Omar Benomar, Hani Abdeen, Houari Sahraoui, Pierre Poulin, and Mohamed Aymen Saied. 2015. Detection of software evolution phases based on development activities. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 15–24.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197. https://doi.org/10.1109/4235.996017

[7] Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth Tamilselvam. 2021. Graph Neural Network to Dilute Outliers for Refactoring Monolith Application. arXiv:2102.03827 [cs.SE]

[8] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.

[9] Fabio Ferreira, Luciana Lourdes Silva, and Marco Tulio Valente. 2020. Turnover in Open-Source Projects: The Case of Core Developers. In *34th Brazilian Symposium on Software Engineering*. 447–456.

[10] Brendan J. Frey and Delbert Dueck. 2007. Clustering by Passing Messages Between Data Points. *Science* 315, 5814 (2007), 972–976.

[11] Alexis Henry and Youssef Ridene. 2020. *Migrating to Microservices*. 45–72.

[12] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. 2021. Service Candidate Identification from Monolithic Systems Based on Execution Traces. *IEEE Transactions on Software Engineering* 47, 5 (2021), 987–1007.

[13] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. 2021. Mono2Micro: a practical and effective tool for decomposing monolithic Java applications to microservices. *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Aug 2021).

[14] Genc Mazlami, Jürgen Cito, and Philipp Leitner. 2017. Extraction of Microservices from Monolithic Software Architectures. In *2017 IEEE International Conference on Web Services (ICWS)*. 524–531. https://doi.org/10.1109/ICWS.2017.61

[15] B.S. Mitchell and S. Mancoridis. 2006. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering* 32, 3 (2006), 193–208.

[16] Juan Enrique Ramos. 2003. Using TF-IDF to Determine Word Relevance in Document Queries.

[17] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Aymen Saied. 2019. Towards automated microservices extraction using muti-objective evolutionary search. In *International Conference on Service-Oriented Computing (ICSOC)*.

[18] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. 2018. Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software* 145 (2018), 164–179. https://doi.org/10.1016/j.jss.2018.08.032

[19] Mohamed Aymen Saied, Erick Raelijohn, Edouard Batot, Michalis Famelis, and Houari Sahraoui. 2020. Towards assisting developers in API usage by automated recovery of complex temporal patterns. *Information and Software Technology* 119 (2020), 106213.

[20] Mohamed Aymen Saied and Houari Sahraoui. 2016. A cooperative approach for combining client-based and library-based api usage pattern mining. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–10.

[21] Robin Sibson. 1973. SLINK: An Optimally Efficient Algorithm for the Single-Link Cluster Method. *Comput. J.* 16 (1973), 30–34.

[22] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. 2021. A Kubernetes controller for managing the availability of elastic microservice based stateful applications. *Journal of Systems and Software* 175 (2021), 110924.