Breaking Down Monoliths: A Graph Based Approach to Microservices Migration

Azaz Ahmed Kiani[1, 2*], Zain ul Islam Adil[1], Yasir Hafeez[3], Javed Iqbal[2], Fahad Burhan Ahmed[3]
[1]National University of Modern Languages (NUML), Rawalpindi, Pakistan.
[2]National University of Science and Technology (NUST), Islamabad, Pakistan.
[3]Pir Mehr Ali Shah Arid Agriculture University, Rawalpindi, Pakistan.
[*]**Correspondence:** Azaz Ahmed Kiani. Azaz.kiani@numl.edu.pk

Citation | Kiani. A. A, Adil. Z. I, Hafeez. Y, Iqbal. J, Ahmed. F. B, "Breaking Down Monoliths: A Graph Based Approach to Microservices Migration", IJIST, Vol. 6 Issue. 3 pp 1076-1087, July 2024
Received | July 10, 2024 **Revised |** July 28, 2024 **Accepted |** July 29, 2024 **Published |** July 30, 2024.

Introduction: The software industry has increasingly transitioned from Monolithic Architecture (MA) to Microservices Architecture (MSA) due to the significant advantages offered by MSA. A crucial first step in this migration process is the identification of suitable microservices.
Novelty Statement: This work aims to introduce an automated method for more effectively identifying potential microservices within monolithic applications.
Materials and Methods: Our approach leverages the source code to construct a frequency-based class dependency graph through graph analysis techniques. A clustering algorithm is then applied to this graph to identify optimal candidate microservices.
Results and Discussion: We evaluate the effectiveness of the proposed approach using several metrics, including the number of microservices, Newman-Girvan Modularity (NGM), and F1-Score. The results demonstrate that the approach accurately identifies candidate microservices, achieving an average F1 score of 0.88 and an average NGM score of 0.526.
Concluding Remarks: The proposed approach proves to be an effective tool for assisting developers in migrating from MA to MSA, facilitating a more streamlined transition process.
Keywords: Microservices; Monolithic, Clustering Algorithm; Microservices Decomposition; Microservices Identification.

Introduction:

As software systems evolve, they often become larger and more complex due to the integration of multiple functionalities. This growth leads to tightly coupled but less coherent components. Monolithic Architecture (MA) is a traditional design approach where all components are integrated into a single, unified system. While MA offers benefits such as ease of development, testing, and deployment in simpler systems, it becomes problematic as applications grow in size and complexity. Monolithic applications can become difficult to scale and maintain, as modifying a single module often requires rebuilding, retesting, and reintegrating the entire application [1][2][3][4][5]. Additionally, monolithic systems face challenges such as poor fault tolerance, limited extensibility, slow development, and inadequate service capacity [3][4][6][7]. Consequently, MA struggles to meet the demands of modern applications, especially those with dynamic workloads [8].

Microservices Architecture (MSA) presents an alternative to these limitations by creating distributed applications as small, independent services. Each microservice operates as a separate process and communicates with others via lightweight mechanisms [5][6][9][10][11][12]. MSA allows services to be developed using different programming languages and data storage technologies, and it emphasizes principles such as single responsibility, high cohesion, low coupling, and minimal disruption to other services. Key features of MSA include scalability, autonomous development, reusability, maintainability, and cloud compatibility [3][4][5][7][8][12].

The software industry has increasingly transitioned from MA to MSA due to the advantages of MSA. Many organizations, including Netflix, eBay, Amazon, IBM, Google, Uber, and Alibaba, have adopted or are in the process of adopting MSA [6][7][8][13]. While building new systems with MSA from scratch can be time-consuming and costly, migrating existing monolithic systems to MSA is often more practical. However, this migration process is challenging [12] since it requires broad and deep analysis of software architecture. Identifying suitable microservices is a critical and complex first step [3][8][12][14][15]. The challenge here is to finding out how to split tightly coupled and highly cohesive components that come with MA. To achieve the high coupling and low cohesion the migration process must address the challenges including managing complexity of inter-service communication, defining clear service boundaries, and ensuring data consistency across the distributed services. In order to effective and smooth transition to MSA the migration process requires an extensive assessment of various architectural dimensions.

Existing strategies for extracting microservices from monolithic systems often require detailed expertise and involve complex heuristics. These methods may also fail to address all aspects of microservice characteristics [11]. Recent systematic literature reviews highlight that current research in microservices decomposition is still developing, with available solutions often focusing on specific scenarios, domains, or programming languages [3].

Objective:

This work aims to introduce an automated method for more effective identification of potential microservices from monolithic applications. The proposed approach employs a modularity-based community detection clustering algorithm along with graph analysis techniques. It starts by creating a class dependency tree based on the frequency of method calls between classes. This frequency-based call dependency network is then analyzed using clustering to identify potential microservices. The approach iterates to refine the clusters until optimal results are achieved. Modularity based clustering approach is used to develop the loosely coupled and functionally cohesive microservices, which is essential for robust and manageable MSA. The objective is to optimize the balance between intra service communications and inter service separations.

Evaluation:

The effectiveness of the proposed method is assessed using metrics such as the number of microservices, Newman-Girvan Modularity (NGM), and F1-Score. Moreover, the proposed algorithm was compared with five different clustering algorithms found in literature (See Section 4: Algorithms). Results demonstrate that the approach accurately identifies well-defined candidate microservices, with an average F1 score of 0.88 and an average NGM score of 0.526. This indicates the approach's effectiveness in facilitating the migration from MA to MSA.

Structure:

The remainder of this paper is organized as follows: Section 2 reviews related literature. Section 3 details the proposed approach. Section 4 presents the results, and Section 5 concludes with recommendations for further research.

Literature Review:

Research into the decomposition of monolithic applications into microservices has led to the development of various techniques. The authors of [14] proposed a method utilizing the SArF software clustering algorithm to extract potential microservices from source code. Their approach, which involved evaluating two distinct programs and reviewing results with two developers, was illustrated using the Software Architecture Finder (SArF) map. Two case studies were conducted to assess the method's effectiveness; one with the purchasing operations division of Fujitsu for an industrial application, and another with the open-source "Spring Boot Pet Clinic" application in both its monolithic and microservices forms. These case studies showcased the method's versatility and effectiveness in various contexts.

A technique introduced by [15] transforms monolithic application code fragments into continuous distributed vectors using a neural network. Microservice candidates are then identified by aggregating related classes based on these vector representations. A study [11] suggested a method for identifying microservices from monolithic object-oriented systems by applying a quality function. This method follows architectural recommendations to enhance outcome relevance and accuracy. [6] Outlined a strategy for converting legacy monolithic applications to microservices architecture (MSA) through program analysis. Their approach analyzed class inheritance, dependency relationships, and function invocation within the legacy application.

DBSCAN hierarchical method was used by [8] based on static analysis of the monolithic application to identify microservices. They validated their approach by comparing results with human-designed microservices. The authors of [16] proposed decomposing monolithic applications into microservices by analyzing Application Programming Interfaces (APIs). Their technique involved grouping similar operation names using hierarchical clustering and word embedding models to generate representations based on operation names. [17] Aimed to enhance application performance by incorporating scalability considerations into their decomposition technique. They analyzed access logs from monolithic applications with an unsupervised machine learning approach and suggested automating resource allocation to microservices in cloud architecture. However, this performance-based approach may be context-specific and dependent on particular testing scenarios.

Researchers in [18] examined version control repositories, converting them into graphs for potential microservice identification using a clustering algorithm. Their method employed three extraction strategies: logical coupling, semantic coupling, and contributor coupling. A limitation of this approach is its focus on classes without considering methods and their input/output arguments. Authors of [19] proposed transforming graph call data into domain entities and computing similarities between these entities. Microservices are then formed by clustering related domain elements using a clustering method.

Proposed Work:

This section outlines the proposed approach which employs modularity-based community detection clustering algorithm to provide a new method to identify the

microservices. The proposed approach optimizes the balance between external separations and internal cohesion thus makes it easier to develop loosely coupled and functionally coherent microservices. Unlike existing methods, the proposed approach innovates by introducing the frequency based weighted graph and iterative modularity optimization concept. It performs well in dynamically refining communities (i.e. microservices), which leads to better cohesion and coupling, and a balance between global and local adjustments for optimum microservices detection performance. The proposed approach consists of two stages.

Phase One involves constructing a frequency-based class dependency network from the source code. **Phase Two** focuses on applying a clustering approach to identify potential microservices. The following details the proposed method.

The approach begins with the source code of the monolithic system to create a weighted class dependency network. This network represents dependencies between classes and the frequency of interactions, serving as the foundation for microservice identification. Algorithm 1 (depicted in Figure 1) illustrates the process of generating the weighted call dependency graph.

Initially, the function extract Classes (S) is employed to extract all classes from the source code, as these classes form the core structural components of the monolithic system. The extract Methods (class) function is then used to retrieve the methods defined within each class. Each method is added as a vertex to the graph G. The algorithm iterates over each method to establish dependencies. For each method, the find References (method) function identifies all methods it invokes. This step is crucial as it reveals interactions between various system components.

The algorithm checks if each referenced method already exists as a vertex in the graph. If it does, the algorithm updates the edges between the methods by increasing their weights using the increment Edge Weight (method, reference) function. This increment reflects the frequency of interactions by adjusting the edge weight, indicating how often method calls occur. This frequency information helps determine the strength of dependencies, which is vital for clustering related components during the monolith's decomposition. If no edge exists, the algorithm uses the add Edge (method, reference, weight=1) function to create a new edge with an initial weight of 1. The resulting weighted graph GGG accurately represents the interaction levels between methods, providing a detailed view of the system's behavior and structure. This detailed graph is essential for subsequent grouping and refinement processes aimed at identifying potential microservices.

For example, consider two classes: Class A, with methods 1 and 2, and Class B, with methods 3 and 4. If method 1 calls methods 3 and 4, and method 2 calls method 4, the graph will include vertices for each method. Directed edges will connect method 1 to methods 3 and 4, and method 2 to method 4. The weights on these edges, initially set to 1, are incremented based on additional calls. For instance, if method 1 calls method 3 again, the weight of the edge from method 1 to method 3 increases to 2. This weighted class dependency graph effectively captures method interactions and their strengths, facilitating the identification of clusters that can be consolidated into microservices.

The second step of the proposed approach uses an iterative refinement clustering technique to identify potential microservices from the weighted class dependency graph. This method focuses on maximizing the graph's modularity, a metric that measures the strength of division into clusters, to group methods into potential microservices. The process begins with community initialization, where each method (node) in the graph is assigned to an initial community (see Algorithm 2a in Figure 2). This initial assignment sets the stage for the iterative refinement process (Algorithm 2b, depicted in Figure 3). The algorithm then evaluates modularity, which assesses the effectiveness of the community structure. By iteratively refining the community assignments based on modularity, the approach aims to optimize the clustering of methods into cohesive microservices. To calculate modularity M, use the following formula:

$$M = \frac{1}{2n} \sum x, y \left[Ax\, y - \frac{kx\, ky}{2n} \right] \delta(cx, cy) \quad (1)$$

```
Algorithm 1: Construction of Weighted Class Dependency Graph
Input: S: Source Code
Output: G = (V, E, w)
classes ← extractClasses(S)
foreach class of classes do
    methods ← extractMethods(class)
    G ← addVertices(methods)
end
foreach method ∈ G do
    references ← findReferences(method)
    foreach reference of references do
        if reference ∈ G then
            if G hasEdge(method, reference) then
                G ← incrementEdgeWeight(method, reference)
            else
                G ← addEdge(method, reference, weight=1)
            end
        end
    end
end
end
```

Figure 1. Algorithm (1) for Construction of Weighted Dependency Graph

```
Algorithm 2a: Modularity Calculation and Node Movement
Input:  G=(V,E,w) // Weighted Dependency Graph
Output:  Updated communities
def initCommunities(G):
    return {v: {v} for v in G.nodes()}
def calcModularity(G, coms):
    m = sum(w for _, _, w in G.edges(data='weight'))
    return sum(
        (sum(G[u][v]['weight'] for u in com for v in com if G.has_edge(u, v)) / (2*m)) -
        (sum(G.degree(n, weight='weight') for n in com) / (2*m))**2 for com in coms.values()    )
def moveNode(G, node, coms):
    best_com, best_gain = coms[node], 0
    for n in G.neighbors(node):
        if coms[n] != coms[node]:
            gain = calcModGain(node, coms[n], G, coms)
            if gain > best_gain: best_gain, best_com = gain, coms[n]
    if best_com != coms[node]: coms[node] = best_com
    return coms
def modularityOptimization(G):
    coms, prev_mod = initCommunities(G), -1
    while True:
        improvement = False
        for node in G:
            old_coms = coms.copy()
            coms = moveNode(G, node, coms)
            improvement |= (coms != old_coms)
        if not improvement or calcModularity(G, coms) <= prev_mod: break
        prev_mod = calcModularity(G, coms)
    return coms
```

Figure 2. Algorithm (2a) for Modularity Calculation and Communities (Micro services) Identification.

The Modularity (M) quantifies the strength of the split of a network into communities by comparing the actual edge density within communities to the expected edge density if edges

were distributed randomly. Where n is the sum of all edge weights in the graph, Ax y is the weight of the edge between nodes x and y, kx is the sum of edge weights connected to node x, and δ (cx, cy) is 1 if nodes x and y are in the same community and 0 otherwise. This formula assesses how well the community structure captures the internal density of edges compared to a random distribution of such edges.

In order to optimize the modularity gain, the algorithm's core includes relocating nodes to nearby communities iteratively. The program analyses the neighboring methods of each method and determines the possible gain in modularity if the method were relocated to the neighbor's community. By analyzing the changes to the internal weights and total weights of the impacted communities, this modularity increase is ascertained. A node is moved to a new community if its new location results in a better modularity. Until no more nodes can be moved to better the situation, this procedure recursively applies to every node, indicating that a locally optimal community structure has been achieved. The algorithm manages singleton nodes and unconnected components within each community to further refine the communities (Algorithm 2b).

If a disconnected subgraph within a community contains more than one node, it is identified and treated as a separate community. Singleton nodes; those with weak connections within their current community are reassigned to the nearest community based on edge weights. This refinement ensures that the identified communities are connected and cohesive. The process of moving nodes and adjusting communities continues iteratively until modularity stabilizes, indicating that no further significant improvements can be made. Stabilization signifies that the community structure optimally reflects the internal organization of the graph in its cluster partitions.

```
Algorithm 2b: Refinement of Communities
Input:  G=(V,E,w) // Weighted Dependency Graph, communities
Output: Refined communities
def refineCommunities(G, coms):
    new_coms = {}
    for com in set(coms.values()):
        for sub in findDisconnectedComponents(com, G):
            if len(sub) > 1: new_coms[hash(frozenset(sub))] = sub
            else: new_coms.setdefault(findClosestCommunity(sub.pop(), coms, G), set()).add(sub.pop())
    return new_coms
def iterRefinementClustering(G):
    coms = modularityOptimization(G)
    prev_mod = calcModularity(G, coms)
    while True:
        coms = refineCommunities(G, coms)
        curr_mod = calcModularity(G, coms)
        if curr_mod <= prev_mod: break
        prev_mod = curr_mod
    return coms
G = constructWeightedClassDependencyGraph(source_code)
MS = iterRefinementClustering(G)
return MS
```

Figure 3. Algorithm (2b) for Refinement of Identified Communities (Microservices).

The proposed method employs a two-step approach to handle unconnected components and singleton nodes, combining refinement with modularity optimization. Unlike existing algorithms that focus solely on modularity optimization, this approach addresses issues of single-node isolation and subgraph disconnections, ensuring that communities are both cohesive and well-connected. This feature enhances the precision and significance of potential microservices identification. By continually maximizing modularity and refining community

structures, the algorithm ensures that the resulting microservices are well-defined, capturing the most coherent and strongly interacting groups of methods. This approach effectively decomposes the monolithic system into smaller, more manageable microservices, improving modularity and maintainability. It is particularly suited for real-world software engineering applications, where maintaining a consistent microservices architecture is crucial.

Evaluation:

Three metrics; the number of Micro Services (MS), the NGM, and the F1-Measure metric were used to assess the proposed approach. The details of these metrics are provided in the subsections that follow.

F1-Measure:

To get the harmonic mean between the two communities that the community identifying algorithms produced, we adapted the method from [12][20]. A clustering technique generates community set X, while community set Y is the reference community set. The reference community y ∈ Y that x fit in then labels each community y x ∈ X.

Furthermore, this stage will generate pairings of (x, y) by matching community x with y that have the greatest number of matching labels. Then evaluating these community sets' quality using recall and accuracy. By computing the ratio of the intersection of sets x and y to the size of set x, where P ranges from 0 to 1, precision P expresses the percentage of correctly identified nodes in set x (equation 2).

$$P = \frac{x \cap y}{|x|} \in [0,1] \quad (2)$$

Below is a quantification of recall: It is the percentage of y's covered nodes divided by x, meaning the percentage of nodes in set y that are covered by set x is quantified by recall R. Where R is between 0 and 1.

$$R = \frac{x \cap y}{|y|} \in [0,1] \quad (3)$$

The accuracy and recall of each pair can be calculated to show the over- and under-estimations of the clustering technique in use. F-Measure can be used to calculate a harmonic mean between precision and recall. F1- Measure is an important metric to measure the accuracy by combining precision and recall. In our study, it is important to measure how well the proposed approach performs. In order to demonstrate the effectiveness of the proposed approach, this metric is crucial for ensuring that our proposed approach not only identifies relevant microservices but also avoids incorporating irrelevant ones. F1 Measure is computed as follows:

$$F1 = 2\frac{\mathbf{precision} * \mathbf{recall}}{\mathbf{precision} + \mathbf{recall}} \quad (4)$$

Newman Girvan Modularity (NGM):

To evaluate the overall quality of the clusters or communities, the modularity metric is utilized. High modularity indicates that nodes within a community are strongly interconnected. The Next Generation Modularity (NGM) metric explores the community structure of the entire network, serving as a key measure for assessing cluster quality. NGM is widely recognized as a standard method for evaluating community detection techniques. It is employed in this study to evaluate the efficiency of the proposed approach. A robust and suitable MSA demands the well-defined microservices that have robust internal relationships and few external dependencies, all of which are indicated by high modularity in the identified microservices. It is an ideal option for evaluating the decomposition of components in our proposed approach because of its widespread use and sensitivity to community structure.

The metric is based on the principle that a random graph is unlikely to exhibit the core characteristics of a well-defined cluster. Therefore, clusters are identified by comparing the actual density of vertices within a community to the expected density if the network's nodes were

connected randomly, disregarding the community's structure. NGM calculate the modularity as follows:

$$Q(s) = \frac{1}{m} \sum c \in S \ (ms - \frac{(2ms+ls)2}{4m}) \quad (5)$$

Number of Microservices:

Total number of microservices that make up an application is represented by these metrics. It is necessary to compare this measure with the ideal amount of microservices for an application. The total number of microservices is the definition of this metric:

$$MS = \sum_{n=1} m \quad (6)$$

Algorithms:

We compared the proposed technique with several algorithms that have been proposed in the literature to determine the correctness and efficiency of the proposed algorithm. These algorithms are the following: the Leiden algorithm (LA), the Louvain algorithm (LV), the Markov clustering algorithm (MC), the Speaker Listener Label Propagation Algorithm (SLPA), and the Rb pots algorithm (RB). Python was used to implement these algorithms.

Test Case Applications:

We took into consideration many small to medium sized applications in order to evaluate the suggested approach and compare it with other competing algorithms. Table 1 displays the applications' specifics:

Table 1. Details of Test Case Application used in Study

Test Case Applications		
Sr. No.	**Application Name**	**Description**
1	Acme Air	Functionalities like booking and searching flights are provided by a monolithic Java program used as an example airline.
2	Spring blog	A Spring Boot Framework-created microservices application. It's just a basic blog website.
3	J Pet Store application	Java was used to develop a reference monolithic application. There is a microservices equivalent to this application that may be utilized to compare the findings of this study.

Results:

Proposed algorithm was evaluated using three distinct monolithic applications. Its performance was compared against five alternative clustering algorithms. Additionally, three assessment measures were utilized to benchmark the clustering algorithms and evaluate the effectiveness of the proposed approach.

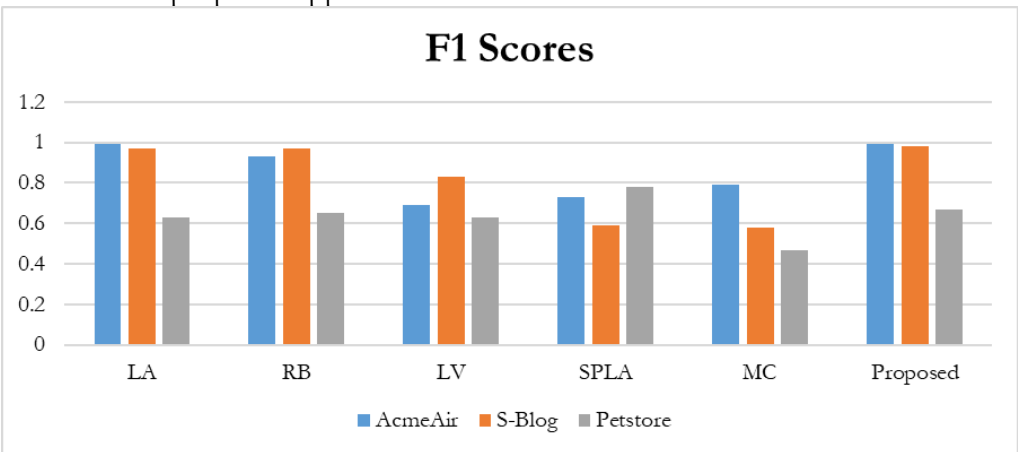

Figure 4. F1 Results comparing proposed and other clustering algorithms on test case applications.

Figure 4 displays the F1-measure results, which offer a balanced view of the decomposition approach's accuracy by combining precision and recall. With an average F1 score of 0.86, the LA algorithm demonstrated superior performance compared to the other algorithms. Conversely, the MC algorithm performed the worst, with an average F1 score of 0.61. The proposed method achieved an average F1 score of 0.88, indicating a significant improvement over the other algorithms.

Figure 5 presents the output of the clustering algorithms based on the NGM metric. For the Acme Air application, most methods achieved the highest possible score for this criterion, while LV reached a maximum score of 0.67. Consistent with the F1 results, the MC algorithm performed the poorest. The proposed method also scored 0.67 for the Acme Air application, reflecting robust and clear connections among classes compared to previous algorithms. Table 2 provides additional details on the average outcomes of the various clustering techniques. The RB and LA algorithms show similar results, ranking highest overall with average F1 values of 0.86 and 0.85, respectively, and NGM scores of 0.49 and 0.48. However, LA slightly outperforms RB with its higher F1 score. NGM scores range from 0.12 to 0.52 across different algorithms, with GN and MC showing notably lower scores of 0.33 and 0.12, respectively.

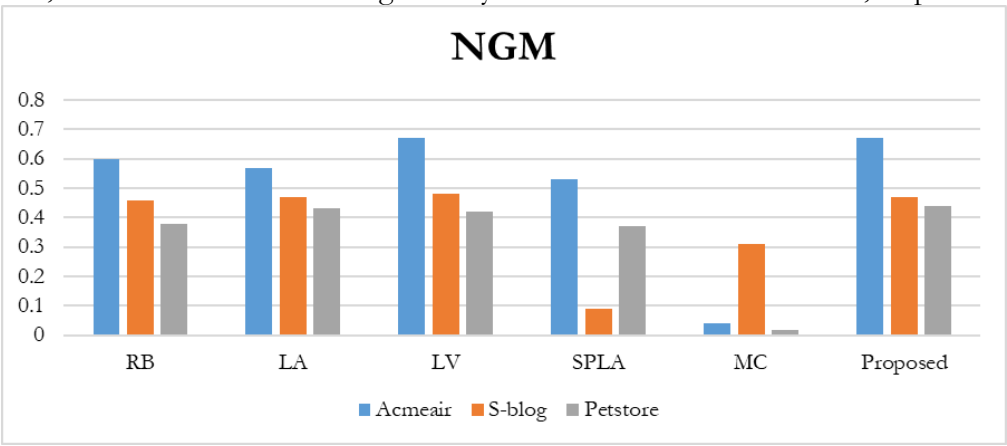

Figure 5. NGM Results comparing proposed and approach and other clustering algorithms

The proposed approach outperforms the other algorithms by a large margin, as seen by the scores for all three criteria (averaged F1-score of 0.88 versus 0.86 and average NGM score of 0.526 versus 0.520). As a result, the suggested solution outperforms the other strategies in terms of accuracy.

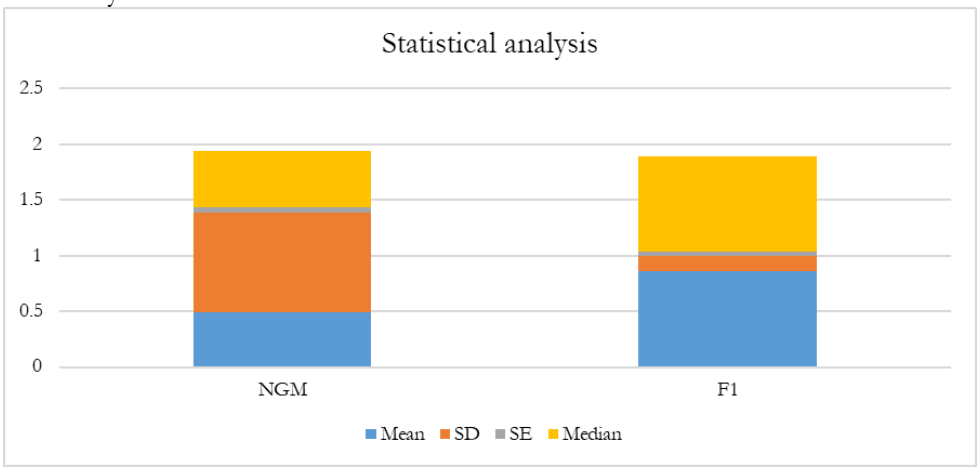

Figure 6. Statistical analysis of the proposed approach.

Figure 6 shows statistical summary of the proposed migration approach. The mean F1 score 0.86, with Standard Deviation (SD) 0.14 indicates the high accuracy. With SD 0.9 and mean NGM score of 0.526 implies good clustering quality. The accuracy of mean estimates is

shown by F1 and NGM standard errors (SE) which are 0.04 and 0.045 respectively. These results show that the proposed approach identifies the more structured microservices from MA with robustness and reliability.

Table 2. Average F1 and NGM results of comparison.

Algorithm	F1	NGM	No. of MS
LA	0.86	0.526	4.2
RB	0.85	0.480	4.1
LV	0.71	0.520	4.8
SPLA	0.70	0.330	6.6
MC	0.61	0.120	7.75
Proposed	0.88	0.526	4.3

Discussions:

The performance evaluation of the proposed technique, using three monolithic applications and three evaluation measures, reveals significant results compared to five alternative clustering algorithms. The proposed algorithm achieved the highest average F1 score of 0.88 and excelled in the NGM metric, attaining an average score of 0.526 for the Acme Air application, slightly surpassing the LV algorithm's score of 0.52. The proposed approach consistently delivered top scores across all metrics, including an average NGM score of 0.526, compared to average F1 scores of 0.86 and 0.85 for the RB and LA algorithms, respectively. Detailed data further support the superiority of the proposed algorithm, demonstrating its effectiveness in accuracy and efficiency for application decomposition. The proposed approach has been tested on three different monolithic applications, proving its robustness and adaptability. It outperformed existing methods and proved its effectiveness in handling the different software architecture scenarios due to its high accuracy and successful decomposition.

Conclusion and Future Work:

A crucial step in migrating from Monolithic Applications (MA) to Microservices Architecture (MSA) is the identification of potential Microservices. This paper introduces a novel method for discovering potential microservices. The proposed approach consists of two key steps: Frequency-Based Class Dependency Graph: The first step involves creating a frequency-based class dependency graph using graph analysis techniques on the source code. Clustering for Microservices Identification: The second step utilizes a proposed clustering algorithm to identify the most promising microservice candidates.

To evaluate the proposed algorithm, we tested it on three distinct small- to medium-sized applications, some of which were also used in previous research. The proposed algorithm was compared with five other clustering algorithms, demonstrating superior results with an average F1 score of 0.88 and an average NGM score of 0.526. The high F1 score indicates strong accuracy in identifying the anticipated microservices, while the NGM score confirms that the generated microservice candidates are not random but exhibit well-defined linkages among the clustered classes. Future work will involve testing the proposed algorithm on larger applications and incorporating additional algorithms to create a comprehensive and robust comparison.

Acknowledgement: The authors acknowledged that this study has not been published before, nor it is under consideration.

Author's Contribution:

- **Azaz Ahmed Kiani and Zain ul Islam Adil:** Proposed the idea, conceptualize the study and prepared the draft.
- **Javed Iqbal:** Supervise the study and validate the outcomes.
- **Yasir Hafeez and Fahad Burhan Ahmed:** Performed the data analysis, experimentation, algorithm implementations and scientific discussions.

Conflict of Interest: The Authors declare that they have no conflict of interest.

References:

[1] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From Monolith to Microservices: A Classification of Refactoring Approaches," Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 11350 LNCS, pp. 128–141, Jul. 2018, doi: 10.1007/978-3-030-06019-0_10.

[2] J. Kazanavicius and D. Mazeika, "Migrating Legacy Software to Microservices Architecture," 2019 Open Conf. Electr. Electron. Inf. Sci. eStream 2019 - Proc., Apr. 2019, doi: 10.1109/ESTREAM.2019.8732170.

[3] Y. Abgaz et al., "Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review," IEEE Trans. Softw. Eng., vol. 49, no. 8, pp. 4213–4242, Aug. 2023, doi: 10.1109/TSE.2023.3287297.

[4] D. Kuryazov, D. Jabborov, and B. Khujamuratov, "Towards Decomposing Monolithic Applications into Microservices," 14th IEEE Int. Conf. Appl. Inf. Commun. Technol. AICT 2020 - Proc., Oct. 2020, doi: 10.1109/AICT50176.2020.9368571.

[5] L. De Lauretis, "From monolithic architecture to microservices architecture," Proc. - 2019 IEEE 30th Int. Symp. Softw. Reliab. Eng. Work. ISSREW 2019, pp. 93–96, Oct. 2019, doi: 10.1109/ISSREW.2019.00050.

[6] Z. Ren et al., "Migrating web applications from monolithic structure to microservices architecture," ACM Int. Conf. Proceeding Ser., Sep. 2018, doi: 10.1145/3275219.3275230.

[7] A. F. A. A. Freire, A. F. Sampaio, L. H. L. Carvalho, O. Medeiros, and N. C. Mendonça, "Migrating production monolithic systems to microservices using aspect oriented programming," Softw. Pract. Exp., vol. 51, no. 6, pp. 1280–1307, Jun. 2021, doi: 10.1002/SPE.2956.

[8] K. Sellami, M. A. Saied, and A. Ouni, "A Hierarchical-DBSCAN Method for Extracting Microservices from Monolithic Applications," pp. 11–2022, Jun. 2022, doi: 10.1145/3530019.3530040.

[9] H. Ren, Q. H., Li, S. L., Qiao, "Method of Refactoring a Monolith into Micro-services," J. Softw., vol. 13, no. 12, pp. 646–653, 2018.

[10] J. Kazanavičius and D. Mažeika, "The Evaluation of Microservice Communication While Decomposing Monoliths," Comput. INFORMATICS, vol. 42, no. 1, pp. 1-36–1–36, May 2023, doi: 10.31577/CAI_2023_1_1.

[11] A. Selmadji, A. D. Seriai, H. L. Bouziane, R. Oumarou Mahamane, P. Zaragoza, and C. Dony, "From monolithic architecture style to microservice one based on a semi-automatic approach," Proc. - IEEE 17th Int. Conf. Softw. Archit. ICSA 2020, pp. 157–168, Mar. 2020, doi: 10.1109/ICSA47634.2020.00023.

[12] O. Al-Debagy and P. Martinek, "Dependencies-based microservices decomposition method," Int. J. Comput. Appl., vol. 44, no. 9, pp. 814–821, 2022, doi: 10.1080/1206212X.2021.1915444.

[13] "An Innovative Methodology for Transitioning from Monolith to Microservices." Accessed: Jul. 31, 2024. [Online]. Available: http://www.icicel.org/ell/contents/2023/4/el-17-04-10.pdf

[14] M. Kamimura, K. Yano, T. Hatano, and A. Matsuo, "Extracting Candidates of Microservices from Monolithic Application Code," Proc. - Asia-Pacific Softw. Eng. Conf. APSEC, vol. 2018-December, pp. 571–580, Jul. 2018, doi: 10.1109/APSEC.2018.00072.

[15] O. Al-Debagy and P. Martinek, "A Microservice Decomposition Method Through Using Distributed Representation of Source Code," Scalable Comput. Pract. Exp., vol. 22, no. 1, pp. 39–52, Feb. 2021, doi: 10.12694/SCPE.V22I1.1836.

[16] O. Al-Debagy and P. Martinek, "A new decomposition method for designing

microservices," Period. Polytech. Electr. Eng. Comput. Sci., vol. 63, no. 4, pp. 274–281, 2019, doi: 10.3311/PPEE.13925.

[17] M. Abdullah, W. Iqbal, and A. Erradi, "Unsupervised learning approach for web application auto-decomposition into microservices," J. Syst. Softw., vol. 151, pp. 243–257, May 2019, doi: 10.1016/J.JSS.2019.02.031.

[18] G. Mazlami, J. Cito, and P. Leitner, "Extraction of Microservices from Monolithic Software Architectures," Proc. - 2017 IEEE 24th Int. Conf. Web Serv. ICWS 2017, pp. 524–531, Sep. 2017, doi: 10.1109/ICWS.2017.61.

[19] N. Santos and A. Rito Silva, "A complexity metric for microservices architecture migration," Proc. - IEEE 17th Int. Conf. Softw. Archit. ICSA 2020, pp. 169–178, Mar. 2020, doi: 10.1109/ICSA47634.2020.00024.

[20] G. Rossetti, L. Pappalardo, and S. Rinzivillo, "A Novel Approach to Evaluate Community Detection Algorithms on Ground Truth," Stud. Comput. Intell., vol. 644, pp. 133–144, 2016, doi: 10.1007/978-3-319-30569-1_10.