Microservices Identification
Methods and Quality Metrics

PhD Dissertation

Omar Al-Debagy

Scientific Supervisor
Dr. Péter Martinek

2021
Declaration

I, undersigned Omar Al-Debagy hereby declare that this Ph.D. dissertation was made by myself, and I only used the sources given at the end. Every part that was quoted word-for-word, or was taken over with the same content, I noted explicitly by giving the reference to the source.

Budapest, Hungary, June 19, 2021

____________________________________
Omar Al-Debagy
Abstract

The dissertation presents different methods for decomposing monolithic applications into microservices ones based on different objectives. These objectives are maintainability, evolution, and scalability. Also, it presents a set of evaluation metrics for measuring the overall design quality of a microservices application.

Three microservices identification methods were presented in this dissertation. Two methods focus on maintainability and scalability, while the other method is focusing on evolution. Evolution can be defined as the continuous process of changing the application throughout its life cycle for various reasons. The first method uses the application programming interface as an input for analysis. The proposed decomposition methodology uses word embedding models to obtain word representations using operation names, as well as, using a hierarchical clustering algorithm to group similar operation names together in order to get suitable microservices. This method is suitable for less complex applications and it is utilized for the purposes of maintainability and scalability.

The second algorithm is a novel decomposition method for refactoring monolithic applications into microservices applications using a neural network model (code2vec) for creating code embeddings from the monolithic application source code. As a result, semantically similar code embeddings are clustered through a hierarchical clustering algorithm to produce microservices candidates to resemble the domain model more efficiently. This method is utilized for the purpose of evolution.

The third method consists of two parts, the first part is representing the source code of the monolithic application as a class dependency graph. This graph represents the structure of the monolithic application and the relationships between the classes of the application. The second part of the method is a graph clustering algorithm to identify the microservices through analyzing the dependencies between the classes of the monolithic application and cluster classes with solid relationships to generate microservice candidates. This method is suitable for complex applications and it is utilized for the purposes of maintainability and scalability.

In this dissertation a set of metrics for microservices architecture applications was introduced. The proposed metrics are the Service Granularity Metric (SGM), the Lack of Cohesion Metric (LCOM), and the Number of Operations (NOO). The proposed metrics measure the granularity, cohesion, and complexity of individual microservices through analyzing the application programming interface. Two points influenced the introduction of these metrics. First point, the lack of empirical research on the evaluation metrics for microservices design. The second point is the need for a new approach to analyze the quality of a microservices design through utilizing the APIs which can provide a different layer of analysis for quality metrics.
Acknowledgment

I would like to thank my parents and girlfriend for their support and encouragement. Also, I am grateful to Dr. Péter Martinek for his guidance throughout my PhD studies.
## Contents

Contents 4

List of Figures 6

List of Tables 7

1 Introduction 9
   1.1 Service Oriented Architecture ............................... 10
   1.2 Microservices Architecture ................................ 10
   1.3 Microservices and SOA ...................................... 11
   1.4 Research Objective ......................................... 12
   1.5 The Decomposition Process .................................. 12
   1.6 Summary of Scientific Contributions ........................ 15
      1.6.1 Journal articles related to the theses .............. 17
      1.6.2 Proceedings articles related to the theses .......... 17

2 Literature Review 19
   2.1 Decomposition Methods in the Literature .................. 20
   2.2 Quality Metrics in the Literature .......................... 23

3 Thesis I 29
   3.1 Introduction ................................................. 30
   3.2 Proposed Microservices Metrics ............................. 31
      3.2.1 The Lack of Cohesion Metric “LCOM” .................. 31
      3.2.2 The Service Granularity Metric “SGM” ................. 32
      3.2.3 Proposed Number of Operations Per Microservice Metric “NOO” 35
   3.3 Results and Discussion ...................................... 36
   3.4 Analysis ..................................................... 40
   3.5 Validation .................................................. 44
   3.6 Conclusion .................................................. 46

4 Thesis II 47
   4.1 Introduction ................................................. 48
   4.2 Methodology ................................................ 48
      4.2.1 Word Embedding Models ................................ 49
      4.2.2 Operation Name Vector ................................. 49
5 Thesis III 61
  5.1 Introduction ............................................. 62
  5.2 Methodology .............................................. 62
    5.2.1 Extracting Code Embeddings ......................... 62
    5.2.2 Code Embeddings Model .............................. 63
    5.2.3 Aggregation Method .................................. 65
    5.2.4 Clustering Method .................................. 65
    5.2.5 Metrics for Evaluating Clustering Method Performance ........................................................................... 66
    5.2.6 Evaluation Metrics .................................... 66
  5.3 Experiments and Results .................................. 68
    5.3.1 Aggregation Method .................................. 68
    5.3.2 Clustering Method Parameters ....................... 69
    5.3.3 Decomposition Results ............................... 69
  5.4 Evaluation Using Chapter 3 Metrics .................... 72
  5.5 Conclusion .................................................. 72

6 Thesis IV 75
  6.1 Introduction ............................................. 76
  6.2 Methodology .............................................. 76
    6.2.1 Constructing the Class Dependency Graph ........... 76
    6.2.2 Identifying Microservices ............................ 77
    6.2.3 Evaluation Metrics .................................. 78
    6.2.4 Clustering Algorithms ............................... 79
    6.2.5 Test Case Applications .............................. 80
  6.3 Results and Analysis ..................................... 81
  6.4 Evaluation Using Chapter 3 Metrics .................... 85
  6.5 Conclusion .................................................. 86

7 Summary of the Results .................................... 87

8 Future Work and Methods Applicability .................. 89
# List of Figures

1.1 Decision Guide for Decomposition Methods ........................................... 14
3.1 Low Metrics Readings .............................................................................. 36
3.2 High Metrics Readings ............................................................................. 37
3.3 Histogram of All NOO Values ................................................................. 39
3.4 Histogram of All SGM Values ................................................................. 40
3.5 All Metric Values for All Microservices .................................................. 41
3.6 Density Distribution of LCOM ................................................................. 43
3.7 Density Distribution of SGM ................................................................. 44
4.1 PayPal’s Silhouette coefficient score of each operation name ................. 53
4.2 Performance of Word2Vec vs fastText ................................................... 54
4.3 The Proposed Decomposition Algorithm ................................................. 58
5.1 AST representation of a factorial function ............................................... 64
5.2 code2vec Model [1] .................................................................................. 65
5.3 Mean Aggregation Method ....................................................................... 65
5.4 High - Level Representation of the Proposed Algorithm ....................... 67
5.5 JPetStore Results .................................................................................... 70
5.6 Metrics Results Comparing the Performance of the Decomposition Methods 71
6.1 An overall View of the Proposed Decomposition Method ....................... 76
6.2 SpringBlog’s Decomposition Result ......................................................... 78
6.3 F1 Results Comparing Clustering Algorithms Results of Different Test Case Application .............................................................. 82
6.4 NGM Results Comparing Clustering Algorithms Results of Different Test Case Application .............................................................. 82
6.5 Averaged Results of F1 and NGM Comparing the Performance of Different Clustering Algorithms .............................................................. 83
## List of Tables

2.1 Literature Review Summary ................................................. 23  
2.2 Presented publications ..................................................... 27  
3.1 Microservices Example ..................................................... 33  
3.2 Descriptive Summary of All the Applications and Calculated Values for All Microservices of All Tested Applications ....................................... 38  
3.3 All the Metrics Values of Overall Metrics ............................ 39  
3.4 Correlation Matrix .......................................................... 41  
3.5 Correlation Matrix of All Metric Values without Single Operation Microservices ..................................................... 42  
3.6 Correlation Matrix for All the Microservices Applications ......... 42  
4.1 Average Silhouette coefficient scores .................................... 53  
4.2 Comparison of the proposed method’s decomposition and the standard decomposition of the two applications ....................................... 55  
4.3 The performance of the proposed decomposition methodology .... 56  
4.4 Comparison of Different Designs for Kanban Boards Application ...... 57  
4.5 Comparison of Different Designs for Money Transfer Application ... 59  
5.1 Dimensions of the Tested Applications .................................... 68  
5.2 Aggregation Methods Accuracy Comparison ........................... 69  
5.3 JPetStore Metrics Scores .................................................... 70  
5.4 Decomposition Results ....................................................... 71  
5.5 Average Results of $CHD$ and $CHM$ Metrics .......................... 72  
5.6 JPetStore Decomposition Results based on Chapter 3 Metrics ....... 72  
6.1 Averaged Results of F1 and NGM Comparing the Performance of Different Clustering Algorithms ........................................... 84  
6.2 Statistical Analysis of the Proposed Decomposition Method Results ... 84  
6.3 Comparison with Other Decomposition Methods from the Literature ... 85  
6.4 AcmeAir Decomposition Results using Chapter 3 Metrics ......... 85  
6.5 FTGO Decomposition Results using Chapter 3 Metrics ............ 85
Chapter 1

Introduction
Monolithic applications are the traditional approach of developing software where the user interface and data access code are implemented together in a single platform. However, this approach can lead to many issues if it was adapted in a cloud environment. For example, tight coupling between components can lead to difficult maintainability [2]. Therefore, many companies, such as Netflix and Amazon, started adopting a new architecture that is more suitable for the cloud environment, and this architecture is known as microservices. Microservices architecture is considered an evolution of Service Oriented Architecture (SOA) [3]; consequently, the next section of the introduction compares the two architectures.

1.1 Service Oriented Architecture

Service-Oriented Architecture and web services were the new technologies used to develop enterprise applications in the last two decades [4]. SOA was introduced to solve the integration issues among the proprietary systems and components. However, these methods and approaches failed to meet the fast-changing business requirements, and many companies adopted SOA prematurely without the need for a move toward a service approach [5]. SOA appeared as a standard for distributed computing and enterprise integration.

SOA benefits the enterprises the most with several benefits includes:

- Agile development that makes the delivery time quicker. Software developers do not need to rewrite code and reintegrate new features. SOA enables them to assemble applications through programmable interfaces.

- Extend the functionality of legacy systems. SOA enables developers to extend the functionality of an old system to newer markets. For example, many financial companies have used SOA to extend old mainframes’ functionality to be used through the web.

- Improve cooperation between the business and IT sectors. Business terminologies can be integrated into the system functionality through using SOA. For example, these terms can be “generate insurance quote,” “calculate capital equipment ROI,” and so on. In this way, business analysts and developers can work together more effectively.

1.2 Microservices Architecture

Nowadays, many companies, such as Netflix, Amazon, and eBay, have migrated their applications and systems to the cloud because the cloud computing model allows these companies to scale their computing resources as per their usage [6]. Martin Fowler defined Microservices Architecture as an approach to developing a small services suite working as a single application. The services are communicating through lightweight mechanisms, such as an HTTP resource API and each service is running independently in its process [7].
1.3. MICROSERVICES AND SOA

Microservices needs a simpler routing mechanism without the need for global governance when compared to SOA [8]. This simple routing mechanism will make services more autonomous and loosely coupled because there is no need to agree on contacts globally. However, services will be responsible for the management of business processes and the interaction with other services.

The main advantages of a microservices architecture are the following:

- Microservices can rely on technology heterogeneity, which means each service in one system can use different technology than the other services to achieve the desired goals and performance [9].

- Microservices is resilient, which means if one component of the system fails, it does not affect the whole system. Newman called this advantage resilience in his book entitled Building Microservices [9].

- The process of scaling can be more accessible compared to monolithic application scaling because only the services that need actual scaling are scaled in the microservices architecture, contrary to a monolithic application requires to be scaled as a whole unit, which may lead to higher hardware usage [9].

- Ease of deployment, because with microservices, each service can be deployed independently without affecting other services’ performance.

- Microservices architecture helps companies align their architecture with its organizational structure, which will minimize the number of people working on a specific codebase. Consequently, microservices enables organizational alignment. Further advantages are composability and optimizing for replaceability [9].

1.3 Microservices and SOA

Microservices and SOA are used to implement applications that require good scalability and maintainability. Microservices focuses on evolutionary design, and many researchers tried to identify the differences between SOA and microservices, but there is no universal agreement between their findings [10].

The main differences between SOA and microservices can be summarized as follows:

- Each service in a microservice architecture is developed independently with its communication protocols. While in SOA, the services communicate through enterprise service bus (ESB). So, if one service fails, it can affect the whole system.

- Microservices uses lightweight mechanisms for messaging protocols such as HTTP and REST, while SOA uses heterogeneous messaging protocols.
• Microservices are made of very specialized services. Each service is doing a specific business process. For SOA, the services can range from small to big enterprise services. In other words, microservices are more granular than SOA.

Nevertheless, many researchers recognize the common characteristics between microservices and SOA, such as loose coupling, location transparency, or statelessness. For example, when Netflix started using microservices architecture, they referred to it as “fine-grained Service Oriented Architecture” [10].

1.4 Research Objective

The objectives of this dissertation are divided into two parts:

• The first part is to identify a methodology to quantify the quality of a microservices application. Moreover, this part was achieved using a set of metrics to quantify the quality of microservices API.

• The second part provides a set of microservices identification algorithms that can refactor a monolithic application into microservices. These algorithms focus on different aspects of an application, such as analyzing the API, static analysis, and software artifacts. Also, these methods focus on different objectives such as maintainability, scalability, or evolution.

Microservices decomposition is the functional decomposition of a monolithic application into services. Microservices have several advantages, including maintainability, reusability, scalability, availability, and automated deployment [6]. Although microservices provide many benefits for the developers, it has some issues, and one of these issues is the decomposition process. In Taibi’s research, the most common issue was the process of decomposing the monolithic system into microservices [11].

The migration issues of microservices found in the literature set a direction for this dissertation’s research process. Therefore, this dissertation aims to find new decomposition methods for refactoring monolithic applications into microservices. Thus, three decomposition methods were found and a metrics framework for evaluating microservices design. Each one of these methods was targeting a different approach in the decomposition process. For example, the first method is analyzing the application’s programming interface to evaluate the decomposition decision. The second method is converting the source code into vector representation based on a neural network model for generating microservices candidates. The third method is analyzing the dependencies between the classes to refactor monolithic applications accordingly.

1.5 The Decomposition Process

Multiple objectives lead developers to decompose a monolithic application into a microservice one. This research provides decomposition methods for three different objectives, and these objectives are:
1.5. THE DECOMPOSITION PROCESS

- **Maintainability**: maintainability can be improved when refactoring a monolithic application into microservices because code understandability can be increased due to the small size of microservices compared to a single monolithic application [12]. Also, the small size leads to reducing the number of bugs in the application [7].

- **Evolution**: evolution is the continuous process of changing the application throughout its life-cycle for various reasons. Again, the small size of microservices can play an essential role in the facilitation of evolution.

- **Scalability**: microservices can enhance the scalability of applications in the cloud because it offers more flexibility on which part of the application needs to be scaled up or down.

Based on the mentioned objectives, different decomposition methods can be provided for developers to understand the decomposition process’s direction. Figure 1.1 shows the flow of how to choose an appropriate decomposition method.

Based on the diagram shown in Figure 1.1, the proposed algorithms in this dissertation can be utilized as follows:

- Thesis II algorithm can be used for the objective of scalability and maintainability with top-down direction and when the application has clearly defined and accessible APIs.

- Thesis III algorithm can be used for the objective of evolution and with bottom-up direction. Also, this method is suitable for re-decomposing already decomposed microservice.

- Thesis IV algorithm can be used for the objective of scalability and maintainability with top-down direction with complex applications that does not have a clearly defined APIs.
Firstly, developers need to set their objective of the decomposition process. For example, why the monolithic application or service needs to be decomposed. Then identifying the direction of the decomposition process, top-down direction, means the decomposition process starts with the analysis of high-level software artifacts. The bottom-up direction starts with low-level artifacts, such as source code [13].
Then, identifying the complexity of the application, if the application is complex or less complex. Ultimately, choosing the decomposition method can be completed by choosing one of the proposed methods based on the previously mentioned inputs.

1.6 Summary of Scientific Contributions

Thesis I

I defined a set of new evaluation metrics to measure the quality of microservices design. These metrics measure the cohesiveness, granularity, and complexity of the services in a microservices application. I have proven the validity of these metrics using Weyuker’s properties.

Microservices are becoming a more popular software architecture among companies and developers. Therefore, there is a need to develop methods for quantifying the quality of microservices design. This research has created a novel set of metrics for microservices architecture applications. The proposed metrics are the Service Granularity Metric (SGM), the Lack of Cohesion Metric (LCOM), and the Number of Operations (NOO). The proposed metrics measure individual microservices’ granularity, cohesion, and complexity by analyzing the application programming interface (API). These metrics can evaluate the overall quality of the design of microservices applications. The proposed metrics were measured on five applications with different sizes and business cases. I have defined limits for the SGM metric which needs to be between 0.2 and 0.6. Besides, the LCOM metric value for a microservice needs to be between 0 and 0.8, with less than ten operations per microservice. These findings can be applied in the decomposition process of monolithic applications as well.

Publications related to this thesis: [JWE20] [SOSE20]

Thesis II

I have created an algorithm for identifying microservices by applying a hierarchical clustering algorithm on API’s operation names. The results proved that this method could decompose a monolithic application efficiently compared to similar methods in the literature. The algorithm is capable to provide a highly scalable and maintainable microservices design.

Many companies are migrating from monolithic architectures to microservice architectures, and they need to decompose their applications to create a microservices application. Therefore, the need comes for an approach that helps software architects in the decomposition process. This research presents a new approach for decomposing monolithic applications to a microservices application through analyzing the application programming interface. The proposed decomposition methodology uses word embedding models to obtain word representations using operation names and a hierarchical clustering algorithm to group similar operation names together to get
suitable microservices. Also, using grid search method to find the optimal parameter values for Affinity Propagation algorithm, which was used for clustering, and using silhouette coefficient score to compare the performance of the clustering parameters. The decomposition approach introduced in this research consists of the OpenAPI specifications as an input, then extracts the operation names from the specifications and converts them into average word embeddings using the fastText model. Lastly, the decomposition approach is grouping these operation names using the Affinity Propagation algorithm. The proposed methodology presented promising results with a precision of 0.84, recall of 0.78, and F-Measure of 0.81.

Publications related to this thesis: [CINTI18] [SOSE20] [PP19]

**Thesis III**

*I have shown and proven that distributed representation of source code can improve monolithic applications’ refactoring process into microservices. The results were proven using cohesion metrics by applying the proposed method on four different use cases.*

The proposed algorithm is a novel decomposition method for refactoring monolithic applications into microservices using a neural network model (code2vec) for creating code embeddings from the monolithic application source code. As a result, semantically similar code embeddings are clustered through a hierarchical clustering algorithm to produce microservices candidates to resemble the domain model more efficiently. The quality characteristics of the results were measured using two metrics for measuring cohesion. These metrics were Cohesion at Message Level (CHM) and Cohesion at Domain Level (CHD). Also, four applications were used as test cases with different sizes ranging from small to big applications. The proposed method showed promising results in terms of cohesion when compared to other decomposition methods. The proposed method resulted in better scores in 4 out of 8 tests compared to other methods. Also, averaged CHD and CHM results were 0.52 and 0.76, respectively, for the proposed method, better results compared to the other methods.

Publications related to this thesis: [SCPE21]

**Thesis IV**

*I have designed a new decomposition method for identifying microservices candidates from monolithic applications using a graph clustering algorithm to cluster classes based on their dependencies. The results were compared with other methods in the literature based on F-Measure and Modularity scores.*

The proposed method consists of two parts; the first part represents the source code of the monolithic application as a class dependency graph. This graph represents the structure of the monolithic application and the relationships between the classes
of the application. The second part of the method is a graph clustering algorithm to identify the microservices by analyzing the dependencies connecting the monolithic application classes and cluster classes with solid relationships to generate microservice candidates. The method was tested with eight different applications, and 11 clustering algorithms were examined to find the most accurate and efficient algorithm. The proposed method produced promising results compared to other research methods with a 0.8 averaged F-Measure (F1) score and 0.44 averaged NGM score. The F1 score shows that the proposed method has good accuracy in detecting microservices candidates. Newman Girvan Modularity metric (NGM) score shows that the generated microservices candidates are correctly structured and that there are well-defined relationships among the clustered classes of the generated microservices.

Publications related to this thesis: [IJCA21]

1.6.1 Journal articles related to the theses


1.6.2 Proceedings articles related to the theses


Chapter 2

Literature Review: Selected Researches from the Literature
This chapter covers the research in the literature related to microservices identification methods and design metrics. The chapter is divided into two sections, the first one is about the decomposition methods, and the second section is about available metrics for microservices in the literature.

2.1 Decomposition Methods in the Literature

There are several research papers done on the topic of microservices decomposition. These methods can be grouped into three categories: static analysis, dynamic analysis, and software artifacts. The static analysis method uses source code as an input and analyzes the application to provide microservice candidates. The dynamic analysis uses the application’s performance as an input to analyze the software components and decompose them accordingly. Software artifacts use software engineering artifacts such as APIs, DFDs, Class Diagrams, or others to analyze and provide microservice candidates for the developers.

Several methods were presented for extracting microservices from a monolithic application. For instance, Gysel et al. [14] developed Service Cutter, a service decomposition framework in which domain models and use cases were used to extract coupling information. This information was represented as weighted graphs to allocate closely related services. Also, they deployed Newman and Girvan [15], and Epidemic Label Propagation [16] clustering algorithms on the extracted coupling information. One issue with these clustering algorithms is that they require the number of clusters to be assigned to be functional, which can be a weak point for this framework because it is hard to define the number of services for large applications. For the evaluation of the method, they developed their classification metrics to calculate the candidate services’ quality. Service Cutter framework was tested with two sample applications, and the suggested services ranged from good to bad according to the classification method. Overall results were good for one test application and acceptable for the other one [14]. They defined 16 coupling criteria based on the literature and the industry. Converting the coupling information into weighted graphs and then clustering them using Epidemic Label Propagation clustering algorithms to generate microservice candidates.

Baresi et al. [17] proposed a solution based on the semantic similarity between the operations available in OpenAPI specifications. They used Schema.org\(^1\) specifications as a vocabulary reference to be mapped with the specifications of the available OpenAPI\(^2\). Also, a fitness function was applied, which was based on DISCO [18], which is a tool that calculates the distributional similarity between two words in a large corpus of the text. The idea of Baresi et al. [17] was to couple standardized OpenAPI specifications with similar semantic characterizations. Baresi et al. [17], evaluated their method using two microservices applications: the first one was a Money Transfer application, which consists of four microservices, and the second application was the Kanban Board which consists of three microservices. The resulted microservices

\(^1\)https://schema.org/
\(^2\)https://www.openapis.org/
candidates of the first application were 80% accurate, which means 8 of the ten operations were adequately decomposed. In the second application, the accuracy rate was 77%, 10 of the 13 operations were decomposed correctly. Thus, they claimed that their method was 80% accurate if the expected decompositions would have been available in advance. Their paper showed that their method produced better results than the Service Cutter [14] method.

Mazlami et al. [19] proposed a decomposition method for monolithic applications by analyzing the version control repository of the application and converting it into graphs for detecting microservices candidates using a graph clustering algorithm. Their method consisted of three different extraction strategies of various coupling strategies. One limitation of the method is the use of classes without considering methods and their input and output parameters.

Abdullah et al. [20] created a decomposition method that considers the scalability and performance of the application and improve its performance after decomposition. Their method used an unsupervised machine learning approach analyzing access logs of monolithic applications. Then, they proposed a method to automatically assign the type of virtual machines and their resources to the microservices instances on a cloud architecture. Their method of decomposing a monolithic application based on the application’s performance can be misleading because it depends on how the application can be used or how the users are using it. Therefore, the methods based on performance analysis need to have very detailed testing scenarios to work efficiently.

Kamimura et al. [21] created a method for extracting microservices candidates from source code using a clustering algorithm. They tested their method on two different applications, and two developers reviewed their results. Also, they visualized the provided microservices for ease of understanding with the Software Architecture Finder "SArF" map for visualization.

Li et al. [22] proposed a dataflow-driven approach for decomposing monolith applications into microservices candidates. They highlighted how the decomposition process is different between service-oriented architecture (SOA) and Microservices. First, services in SOA are coarse-grained, while microservices are fine-grained. Second, the process is bottom-up in SOA and top-down first, then bottom-up in microservices. Their method consists of 4 steps, first analyzing the requirements of the monolithic application. Second, constructing data flow diagrams (DFD). Third, compress DFDs into decomposable DFDs. Fourth, propose microservice candidates through decomposable DFDs. Furthermore, they used cohesion and coupling metrics to evaluate their results compared to Service Cutter and API analysis. The issue with their approach is the need of a very detailed DFD to make the process of identifying microservices more accurate. Besides, they used a relatively small application for the evaluation.

Taibi and Systa [2] proposed a decomposition method using a data-driven approach based on process mining by utilizing log files as a data source. Their decomposition method consisted of 6 steps. The first step is the execution analysis path. Including the second step is the frequency analysis of the execution path. Removing circular dependencies is the third step. The fourth step is identifying decomposition options. The fifth step is ranking the decomposition options based on metrics. Fi-
Finally, the sixth step is selecting the decomposition option. They used coupling and the number of classes as metrics for step five. Their evaluation method depending on the coupling metric, lacks because their method needs other metrics such as cohesion.

Saidani et al. [23] introduced a new decomposition method called MSExtractor. They used the source code of monolithic applications to extract classes and group classes to create microservices candidates. For the evaluation of their method, they used cohesion and coupling metrics. Furthermore, they utilized a nondominated sorting genetic algorithm to identify microservices from the source code. Their method is based on extracting classes from the source code and cluster these classes using the NSGA-II algorithm.

Jin et al. [24] [25] proposed a functional oriented decomposition method for microservices applications that monitor the dynamic application behavior and clusters execution logs or traces. They proposed some evaluation metrics for cohesion and coupling. The logs are generated using specific test cases, but sometimes these test cases cannot cover all the business functionalities, which may lead to ignoring some essential classes and functionalities.

Nunes et al. [26] used call graphs to represent the source code of the monolithic application to be utilized in the process of identifying microservices. Their method consists of four steps, including collecting data, generate clusters, visualization, and modeling. The first step is generating call graphs from the source code. The second step is clustering the generated call graphs using hierarchical clustering algorithms. The third step is visualizing the cluster algorithms to have a high-level view of the identified microservices. The fourth step is changing the visualized clusters to meet the requirements of the development team. Their method has some limitations, such as the method is only limited to a specific development framework, and the method does not work with all the versions of Java programming language.

Santos and Silva [27] proposed a decomposition method that collects graph calls of the monolithic application and converts them into domain entities. Then a similarity function measures the similarity between two entities, and a clustering algorithm groups similar entities together to create microservices candidates. Also, they proposed a complexity metric to verify the validity of the suggested microservices candidates.

Selmadji et al. [28] created a decomposition method that uses the source code of the monolithic application. Their method uses a quality function to evaluate the structural and functional validity of microservices and their data autonomy. Their method can face issues when there is a microservice containing only one class.

Also, graph clustering algorithms were used in refactoring other software architectures. For example, Chiricota et al. [29] created a method for clustering software modules based on the relations between these modules. Also, they created a new metric that measures the edge density of graphs, so they have a metric-based graph clustering algorithm. Their method has one limitation which is the large number of isolated vertices that are produced. Shtern and Tzerpos [30] created a review article on the most important software clustering methodologies which included a section about software decomposition through clustering algorithms. Other clustering algorithms were utilized by other researchers in several refactoring algorithms [31–34].
Table 2.1 summarizes the methods mentioned in the literature review section above and included the applied types of inputs and the type of decomposition they used.

<table>
<thead>
<tr>
<th>Research</th>
<th>Input</th>
<th>Decomposition Method</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abdullah et al. [20]</td>
<td>Log Files</td>
<td>Performance Based</td>
<td>2019</td>
</tr>
<tr>
<td>Mazlami et al. [19]</td>
<td>Commits</td>
<td>Version Control Analysis with Graphs</td>
<td>2017</td>
</tr>
<tr>
<td>Kamimura et al. [21]</td>
<td>Source Code</td>
<td>SArF software clustering algorithm</td>
<td>2018</td>
</tr>
<tr>
<td>Li et al. [22]</td>
<td>Data Flow Diagrams</td>
<td>Analyzing DFD</td>
<td>2019</td>
</tr>
<tr>
<td>Taibi and Systa [2]</td>
<td>Log Files</td>
<td>Analysis of the execution paths</td>
<td>2019</td>
</tr>
<tr>
<td>Saidani et al. [23]</td>
<td>Source Code</td>
<td>Nondominated sorting genetic algorithm</td>
<td>2019</td>
</tr>
<tr>
<td>Jin et al. [24]</td>
<td>Log Files</td>
<td>Clustering execution traces</td>
<td>2018</td>
</tr>
<tr>
<td>Nunes et al. [26]</td>
<td>Source Code</td>
<td>Clustering call graphs</td>
<td>2019</td>
</tr>
<tr>
<td>Santos and Silva [27]</td>
<td>Method Calls</td>
<td>Clustering call graphs</td>
<td>2020</td>
</tr>
</tbody>
</table>

2.2 Quality Metrics in the Literature

Elhag et al. [35] introduced a set of metrics for evaluating the design of service-oriented design. They introduced two types of metrics, the first type is called basic metrics, and the second type is called derived metrics. The basic metrics include the number of services, number of operations, provider, consumer, and importance of provider. Derived metrics include coupling metric, cohesion metric, and complexity metric. The results of their paper show the significance of these metrics and how these metrics can be calculated. These metrics were evaluated theoretically, but the paper lacks empirical evaluation.

In Perepletchikov et al. [36] research, the authors introduced the effects of cohesion metrics on maintainability prediction. The metrics presented were Service Interface Data Cohesion, which measures the cohesion of service by finding which operations are using the same input parameters. A service is considered highly cohesive when all the operations in service share the same input parameters. The second metric is Service Interface Usage Cohesion, which works by finding the number of clients using specific service operations. The service is considered highly cohesive if all the service operations get invoked by all the clients or service consumers. The third metric of Perepletchikov et al. paper [36] is Service Sequential Usage Cohesion which is similar
 CHAPTER 2. LITERATURE REVIEW

to the previously mentioned metrics. However, it considers the dependency between
the service operations. For example, the service is considered cohesive if one opera-
tion’s output is the input of another operation. The fourth metric is Strict Service
Implementation Cohesion, which measures a service’s cohesiveness based on the ex-
posure of its operations through the interface. Thus, it is presenting the relatedness
between implementation elements. The fifth metric is Loose Service Implementation
Cohesion, which is similar to Service Sequential Usage Cohesion, except it also con-
siders the indirect connection between elements in the measurement. The last metric
is Total Interface Cohesion of a Service, the normalized sum of all the previously
mentioned metrics, representing a service’s total cohesiveness. They used a property-
based software engineering measurement framework proposed by Briand et al. [37]
to validate their metrics, which consist of several mathematical properties to define
complexity, cohesion, coupling, and size of the software. These metrics are similar
to the cohesion metrics proposed by this dissertation because both are derived from
metrics used for object-oriented programming. Perepletchikov et al. [36] research was
not validated empirically, which can be considered a limitation.

The application decomposition process through measuring cohesion between the
services interface is presented in Athanasopoulos et al. [38]. The first metric is Mes-
sage Level Cohesion, which measures the similarities between the messages used by
operations, so two operations are related if the input and output messages of these
operations are similar. The other metric is Conversation Level Cohesion, which mea-
sures the cohesion between operations on the premise that the output of one operation
is similar to the other operation’s input. Thus, two operations are similar if their input
is similar to the output of another operation. The next metric is Domain Level Cohe-
sion, which measures cohesion between operations if they have similar functionalities.
In other words, two operations are considered related to each other if the names of
these operations share the same domain-level terms. In their paper, Athanasopoulos
et al. [37] found a decomposition method for decomposing application based on the
cohesion level of the services interface and their method of measuring the cohesion
between these services are interesting to the work of this research. Because they
used a similar methodology to the proposed method of this research, this method
analyzes the interface of services to find the quality of application design. However,
Athanasopoulos et al.’s metrics are used for web services, while this research’s met-
rics are proposed for microservices architecture. Furthermore, this research provides
other metrics for a microservices architecture, such as granularity and complexity,
that Athanasopoulos et al. did not provide.

In research done by Heinrich and Zimmermann, [39], they propose four different
granularity metrics to decide if services are fine-grained or coarse-grained. Their
research focused on a service-oriented architecture. The first metric is the Width
metric, which works based on the number of direct and indirect functions a service
provides. This metric’s value is between 1 and 0, where a value close to 0 refers to
coarse-grained service granularity, while a value close to 1 refers to a fine-grained
service granularity. Secondly, the Depth metric is instinctively interpretable and uses
the decomposition layer where the services can be found, and this feature makes these
metrics different from the Width metric. Its value is between 0 and 1, where 0 refers
2.2 QUALITY METRICS IN THE LITERATURE

to coarse-grained services, and one refers to fine-grained services. The third metric is a combination of the previous two metrics, Width, and Depth. This combination improves the measurement of these metrics because it combines the advantages of the two. The last metric is Size, where each function is calculated independently using its lines of code. Again, similar to previous research, this research focuses on one aspect of service-oriented applications rather than more than one. Also, they did not provide proper validation for their metrics.

Research done by Alahmari et al. [40] presents a framework to evaluate service granularity in a service-oriented architecture. This framework consists of several metrics to measure the granularity of services. The metrics for data granularity score consists of assigning a weight for different data types in parameters; for example, simple data type parameter has 1 as a weight, user-defined data type has 5, and complex data types have 10 as a weight. The value of this metric is between 0 and 1. The closer to 1 it is, the more coarse-grained the services are. Another metric from this framework is the Functional Granularity Score, which works by assigning different weights for different operations types. For example, assigning 1 as a weight if the service operation has the only CRUD “create, read, update, and delete” operations, assigning 5 if the operation performs business logic, and assigning 10 as a weight if the operation performs business logic and has CRUD operations. Finally, Service Operations Granularity “SOG” metric is the product of multiplying ODG “Operational Data Granularity” and OFG “Operational Functional Granularity” metrics of each operation, which is the metric that gives the measurement of the granularity between services. To get the whole application’s granularity, they calculated the average of SOG scores of the services. Chapter 3 implemented a similar version of Alahmari et al. metrics with some modifications to be suitable for microservices architecture design. Their case study was relatively small, consisting of one application with 14 operations.

An important measurement for assessing the complexity of any software is the number of methods that it has. Weighted Methods per Class “WMC” is a metric used for object-oriented programming to find a number of methods used by a specific class. This metric hypothesizes that a class with a higher number of methods is more prone to errors compared to classes with fewer methods [41]. In their research, they used the complexity of the methods, which they used the number of methods as the complexity metric for classes [42]. Therefore, the number of operations per microservice has been considered as a complexity metric for the microservices application in chapter 3.

The Lack of Cohesion metric measures a class’s cohesiveness in an object-oriented environment by finding the similarity between methods and fields within a class [43]. The Lack of cohesion method by Chidamber and Kemerer is defined as the number of methods in a class that does not have any common fields between minus the number of methods in the class with at least one common field [44]. Henderson-Sellers introduced a revised version of the Lack of Cohesion was introduced by Henderson-Sellers [45] to normalize it. A revised version of Henderson-Sellers’s Lack of cohesion metric is used in chapter 3 to be suitable for microservices design.

In a research [46] done by Coscia et al. used metrics to calculate the quality of software design using WSDL documents of Web Services. Furthermore, they presented a
statistical correlation analysis showing the correlation between object-oriented met-
rics and WSDL metrics. Also, they used 11 different metrics for calculating service
implementations. For example, they used WMC and LCOM, which are similar to
some presented in this research.

Taibi and Systa [2] proposed a decomposition method using a data-driven ap-
proach based on process mining through utilizing log files as a data source. Their
dercomposition method consisted of 6 steps. They used coupling and the number of
classes as metrics for evaluating one of the steps. They proposed a coupling metric to
validate the proposed microservices by their decomposition method. Moreover, evalu-
ating microservices with more metrics, such as cohesion and granularity, can provide
more information regarding the application, which can be considered a limitation to
the research of Taibi and Systa [2].

Jin et al. [24] proposed new cohesion and coupling metrics to evaluate the design of
microservices. These metrics evaluate the design of microservices through analyzing
the interface and classes of the source code. They introduced two metrics for calculat-
ing cohesion at the domain level and message level. Also, three coupling metrics were
defined by identifying the interaction between the microservices. Their approach for
calculating the cohesion metric is different from the lack of cohesion metric proposed
in this dissertation because they used source code as the base for their metric, while
the proposed LCOM uses APIs.

Santos and Silva [27] presented a complexity metric for microservices applications.
This metric was used to evaluate the proposed microservices while migrating from
monolithic applications. They calculated the complexity of each functionality in the
application. The proposed metrics measure the complexity through the cost of the
decomposition process. So, two metrics were proposed for complexity: the complexity
of functionality, and second, the complexity of a decomposition. To evaluate their
metrics, they used correlation with other metrics, similar to the approach that I used
in chapter 3.

A literature review [47] was done by Bogner et al., which investigates different
metrics for measuring maintainability used in service-oriented architecture and their
applicability in a microservices architecture. They characterized the researches that
they found into 4 different categories, which were size, complexity, coupling, and
cohesion. Their research found only one metric related to size, which is the Weighted
Service Interface Count “WSIC,” which represents the number of available operations
through the service API. Then they discussed the applicability of these metrics on
microservices applications. However, they did not apply any of the obtained metrics
on any specific microservices application.

Comparing the presented metrics of chapter 3 with the metrics that are available
in the literature, it is apparent that there is a small number of researchers working
on metrics that are dealing with microservices architecture. Furthermore, the
scope of the proposed metrics is different from the available metrics in the literature.
For example, this research provided a metrics framework focusing on granularity, co-
hesion, and complexity, while other papers focus on each metric alone or different
combinations of other metrics.

The mentioned papers in the literature review fall into three groups based on the
approaches they used to measure their metrics. The first group is the researches that used service interfaces to obtain their metrics measurements. This group focuses on measuring the metrics without accessing the source code, and this group includes [36], [38], [40], and [46]. A second group is the researches that used source code as their medium to obtain the metrics measurements, and this group includes these researches [2, 24, 27]. The third group consists of only single research [39], which utilized a graph definition to measure their defined metrics. Lastly, there are two other researches [47], [35], which did not fall into any of the mentioned groups, because they used theoretical approaches of evaluation instead of empirical methods. The approach that my research used falls under the first group because it uses API to measure the proposed cohesion, complexity, and granularity metrics for microservices applications.

Table 2.2 presents a comparison of the metrics presented in the literature review section. This table consists of the papers’ authors and the metrics that these papers focused on, such as cohesion, complexity, coupling, or granularity. Also, the architecture listed metrics targets like Service Oriented Architecture (SOA), microservices, or others. Additionally, the approach that the research used to obtain the metrics.

<table>
<thead>
<tr>
<th>Papers</th>
<th>Focus</th>
<th>Architecture</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elhag et al.</td>
<td>coupling, cohesion and complexity</td>
<td>SOA</td>
<td>Theoretical</td>
</tr>
<tr>
<td>Perepletchikov et al.</td>
<td>cohesion</td>
<td>SOA</td>
<td>Service Interface</td>
</tr>
<tr>
<td>Athanasopoulos et al.</td>
<td>cohesion</td>
<td>Web Services</td>
<td>Service Interface</td>
</tr>
<tr>
<td>Heinrich and Zimmermann</td>
<td>granularity</td>
<td>SOA</td>
<td>Graphs</td>
</tr>
<tr>
<td>Alahmari et al.</td>
<td>granularity</td>
<td>SOA</td>
<td>Service Interface</td>
</tr>
<tr>
<td>Ordiales Coscia et al.</td>
<td>cohesion and complexity</td>
<td>Web Services</td>
<td>Service Interface</td>
</tr>
<tr>
<td>Bogner et al.</td>
<td>size, complexity, coupling, and cohesion</td>
<td>Microservices</td>
<td>Theoretical</td>
</tr>
<tr>
<td>Taibi and Systa</td>
<td>coupling</td>
<td>Microservices</td>
<td>Source Code</td>
</tr>
<tr>
<td>Jin et al.</td>
<td>cohesion and coupling</td>
<td>Microservices</td>
<td>Source Code</td>
</tr>
<tr>
<td>Santos and Silva</td>
<td>complexity</td>
<td>Microservices</td>
<td>Source Code</td>
</tr>
</tbody>
</table>
Chapter 3

3.1 Introduction

Nowadays, many companies are migrating to microservices architecture because of the advantages that this architecture is providing, such as ease of maintenance, flexibility in implementing different technologies, more scalability, and better IT governance [6]. Nevertheless, there is still no straightforward approach to evaluating the design of microservices applications to create exceptional designs [48]. Although other software architectures and approaches have different methods and metrics to evaluate their design, microservices architecture has a shortage in design metrics; therefore, this chapter presents three software metrics to evaluate the design of microservices applications.

Software metrics are used to measure software quality, which is an essential method to make the quality of software quantifiable [49], [50]. Applications are becoming more complex and sophisticated. Besides, they need to be updated regularly due to the demands of customers. Hence, the need for software metrics is essential to estimate the cost of these developments and the later maintenance. There are different types of software metrics in the literature for different types of architectures and environments, such as cohesion, coupling, complexity, granularity, etc.

In his book, Newman mentioned that good service design should focus on high cohesion and loose coupling. A good service needs to have several functions related to each other and communicate with other functions as loosely as possible, based on Newman’s description of a good design [9]. Another aspect that can affect the performance of microservices is the size of the application or the size of these services, referred to as granularity [51]. Therefore, cohesion and granularity metrics are considered in this chapter.

Cohesion is “the degree of conceptual consistency within an object” [52]. In object-oriented programming, cohesion refers to the degree to which data and methods in a class are related. High cohesion tends to be more favored than software with low cohesion because high cohesion software refers to qualities such as reliability, understandability, robustness, and reusability. Thus, microservices applications need to be cohesive in order to provide good performance to the users.

One of the major objectives of software architecture and design is defining the responsibilities of each software component, which was introduced by Robert Martin [53] as Single Responsibility Principle (SRP) and defined as follows "A class should have only one reason to change.” The SRP principles can be applied for microservices architectures as well to create small and cohesive services. This will reduce the complexity and the size of the services. Therefore, it will produce better cohesion between the microservices in the designed application. Another principle is Interface Segregation Principle which defined as "No client should be forced to depend on methods it doesn’t use” [53]. In other words, a microservice’s API should expose methods that are related to each other.

Service granularity is the range of functionalities provided by a service [54]. Different services have different levels of granularity. Business logic and requirements can affect the granularity of service operations in a service-oriented architecture. Fine-grained services can increase the reusability of business logic significantly. Also,
3.2.3 PROPOSED MICROSERVICES METRICS

fine-grained services can improve cohesion, coupling, and a better understanding of the design. Nevertheless, fine-grained services lead to increased traffic on the network and the processing of identifying and working with errors more challenging. However, these issues can be solved with a faster and more stable network [55].

As a result of the lack of empirical research on the evaluation metrics for microservices design and the need for an approach to identify a microservices application’s goodness. Also, the topic of microservices is still young, especially evaluation metrics for microservices [47]. Therefore, these points motivated this research work. This research presented a set of evaluation metrics to measure the quality of microservices design and validate these metrics’ measurement against several microservices applications design. The proposed metrics provide guidelines for developers in the process of decomposing monolithic applications into microservices applications.

3.2 Proposed Microservices Metrics

This section presents the metrics for measuring the cohesiveness, granularity, and complexity of the services in a microservices application by obtaining operations of the services from the application programming interface (API) and extracting several information from these operations such as input parameters, output responses, and type of operations. Using the extracted information from the API, several metrics were created to measure the quality of a microservices application, and these metrics were Lack of Cohesion Metric (LCOM), Service Granularity Metric (SGM), and Number of Operations (NOO).

3.2.1 The Lack of Cohesion Metric “LCOM”

LCOM measures the cohesiveness or, in other words, the similarity between the operations in specific service and if these operations are related to each other. In this research, the LCOM metric is based on Henderson-Sellers’s lack of cohesion metric for object-oriented programming. It consists of finding how many times a specific parameter has been used in a specific microservice, divided by the product of the number of operations multiplied by the number of unique parameters, see Equation 3.1.

\[
LCOM = 1 - \frac{\sum_{i=1}^{n} OP_i}{M \times F}
\]

where \(OP\) is the occurrence of a specific microservice parameter, \(M\) is the number of operations in a specific microservice, and \(F\) is the number of unique parameters in a microservice.

Furthermore, finding the average of LCOM to get the lack of cohesion for the whole microservices application, and the value of LCOM must be between zero and unity. The closer to unity, the more the applications lack cohesion, which leads to more complexity and, in return, may lead to more errors in the application. Therefore, zero is considered the perfect score for this metric which means the microservice has
operations that are entirely related to each other and fully cohesive. Equation 3.2 gives the LCOM average for the entire microservices application.

\[ ALCOM = \frac{\sum_{i=1}^{n} LCOM_i}{NS} \]  

where \( NS \) is the number of microservices in the application, this metric can show the application’s overall cohesion, and \( n \) is the number of microservices in the whole application.

An API for a small microservices application (Table 3.1.) has been used to show these metrics’ mechanisms in detail. The application consists of three microservices and 13 operations. The name of the application is the Kanban Board\(^1\), a sample application written in Java; it allows users to create Kanban boards and tasks. For further testing, more applications were used in the next section of this chapter.

For example, LCOM for Boards microservice was 0.11, which means this microservice has a cohesive relationship between its operations. It was calculated as follows:

\[ LCOM = 1 - \frac{24}{3 \times 9} \approx 0.11 \]

where 24 refers to the total number of all parameters, 3 is the number of operations inside the microservice, and 9 is the total number of unique parameters without repetition. The LCOM value for the Authentication microservice is zero and 0.64 for the Tasks microservice. In order to get the overall LCOM value for the whole application, we need to calculate the average of LCOM, which was calculated like this:

\[ ALCOM = \frac{0.11 + 0 + 0.64}{3} \approx 0.25 \]

The ALCOM value of 0.25 signifies that the application has a good cohesion among its microservices and high cohesion between its operations, which means easier maintainability and increased reusability. If the value was close to 1 it means that the application has a poor cohesive design.

### 3.2.2 The Service Granularity Metric “SGM”

SGM metric consists of two different measurement metrics to measure the service granularity of a microservices application. These two metrics are Data Granularity of a Service (DGS) and Functional Granularity of a Service (FGS). This metric is based on Alahmari et al. metrics [40] with some modifications to be suitable for microservices architecture design.

First, DGS considers the size of the input and output data of a specific microservice and calculates DGS for each operation in the microservice. The measurement of DGS checks if the operations are using excessive data. Fine-grained and coarse-grained parameters define the whole idea of the DGS metric. DGS is defined as follows:

\[^1\text{https://github.com/eventuate-examples/es-kanban-board}\]
### Table 3.1: Microservices Example

<table>
<thead>
<tr>
<th>Microservice Name</th>
<th>Method Name</th>
<th>Description</th>
<th>Input parameters</th>
<th>Output Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>getBoardUsingGET</td>
<td>Request Board using GET method</td>
<td>id</td>
<td>createdBy, createdDate, id, title, updatedBy, updatedDate, description</td>
</tr>
<tr>
<td>Boards</td>
<td>saveBoardUsingPOST</td>
<td>Save Board to the database using POST method.</td>
<td>title, creation, update, description</td>
<td>createdBy, createdDate, id, title, updatedBy, updatedDate, description</td>
</tr>
<tr>
<td></td>
<td>listAllBoardsUsingGET</td>
<td>Request all boards using GET method</td>
<td>id, createdBy, createdDate, id, title, updatedBy, updatedDate, description</td>
<td></td>
</tr>
<tr>
<td></td>
<td>startTaskUsingPUT</td>
<td>Make the task active using PUT method</td>
<td>id, boardId</td>
<td>createdBy, createdDate, id, title, updatedBy, updatedDate, description</td>
</tr>
<tr>
<td></td>
<td>scheduleTaskUsingPUT</td>
<td>Schedule a task using PUT method</td>
<td>id, boardId</td>
<td>createdBy, createdDate, id, title, updatedBy, updatedDate, description</td>
</tr>
<tr>
<td></td>
<td>getHistoryUsingGET</td>
<td>Retrieve all previous tasks using GET method</td>
<td>id</td>
<td>eventType, boardId, createdBy, createdDate, description, deleted, status, title, updatedBy, updatedDate</td>
</tr>
<tr>
<td></td>
<td>completeTaskUsingPUT</td>
<td>Assign a task completed using PUT method</td>
<td>id, boardId</td>
<td>createdBy, createdDate, description, deleted, status, title, updatedBy, updatedDate</td>
</tr>
<tr>
<td></td>
<td>backlogTaskUsingPUT</td>
<td>Backlog task using PUT method</td>
<td>id</td>
<td>createdBy, createdDate, description, deleted, status, title, updatedBy, updatedDate</td>
</tr>
<tr>
<td></td>
<td>deleteTaskUsingDELETE</td>
<td>Delete a task using DELETE method</td>
<td>id</td>
<td>createdBy, createdDate, description, deleted, status, title, updatedBy, updatedDate</td>
</tr>
<tr>
<td></td>
<td>updateTaskUsingPUT</td>
<td>Update a task using PUT method</td>
<td>Id, TaskDescription, title</td>
<td>createdBy, createdDate, description, deleted, status, title, updatedBy, updatedDate</td>
</tr>
<tr>
<td></td>
<td>saveTaskUsingPOST</td>
<td>Save a task using POST method</td>
<td>boardId, creation, update, deleted, status, title, TaskDescription</td>
<td>createdBy, createdDate, description, deleted, status, title, updatedBy, updatedDate</td>
</tr>
<tr>
<td></td>
<td>listAllTasksUsingGET</td>
<td>Request all tasks using GET method</td>
<td>boardId</td>
<td>createdBy, createdDate, id, title, updatedBy, updatedDate, deleted, status, description</td>
</tr>
<tr>
<td>Authentication</td>
<td>doAuthUsingPOST</td>
<td>Login using POST method</td>
<td>email</td>
<td>token</td>
</tr>
</tbody>
</table>
\[
DGS = \frac{\sum_{i=1}^{n} IPR_i}{\sum_{i=1}^{n} FP_i} + \frac{\sum_{i=1}^{n} OPR_i}{\sum_{i=1}^{n} CP_i}
\]

(3.3)

where Input Parameters (IPR) represents the number of input parameters in an operation, FP is the total number of input parameters in a microservice, Output Parameters (OPR) is the number of output parameters in an operation, and CP is the total number of output parameters in a microservice. If the value of DGS close to 1 it indicates coarse-grained data in the microservice. While the value of DGS is close to 0 indicates fine-granular data.

The DGS metric was calculated for the Kanban Board application using the same data available in Table 3.1. For example, the DGS value for getBoardUsingGET operations was 0.57, which indicates that the microservice has a good granularity level because it is not too fine-grained or too coarse-grained, a microservice has a good granularity level if it maximizes system modularity while minimizing the complexity [56].

To get the service granularity metric of the microservice, we need to calculate the DGS metric for every operation in the microservices application. The DGS value for the getBoardUsingGET operation is calculated as follows:

\[
DGS = \frac{1}{5} + \frac{7}{19} \approx 0.57
\]

The Functional Granularity of a Service (FGS) metric measures the functional granularity of operations in a microservice. Each operation has a different level of capability or a different level of logic. The FGS assigns different weights to each CRUD function (create, read, update, and delete). These weights depend on the level of data manipulation that the operation accomplishes; for instance, a create operation has a higher weight than the other operations because it creates new records in the database. Therefore, create operations have a weight of 4, update operations have a weight of 3, delete operations have a weight of 2 and read operations have a weight of 1. In order to measure the functionality granularity of each operation in a microservice, the FGS metric is defined as follows:

\[
FGS = \frac{OT}{\sum_{i=1}^{n} O_i}
\]

(3.4)

where \( OT \) is the weight for a specific operation in a microservice, and \( O \) is the summation of all the weights in a specific microservice. For example, calculating FGS for the getBoardUsingGET operation, a read operation results in a 0.17 score for the FGS metric. This value is the result of dividing the weight of this read operation, which is 1 by the total weights of all the operations in the microservice, which is equal to 6, which is the addition of three different operations; two of them are read operations, and one is a create operation. This FGS score was calculated as follows:

\[
FGS = \frac{1}{6} \approx 0.17
\]
Finally, Service Granularity Metric (SGM) measures the overall granularity of operation based on DGS and FGS metrics for every operation in the microservices application. SGM was defined as it is presented in Equation 3.5.

\[ SGM = \sum_{i=1}^{n} DGS_i \times FGS_i \]  

(3.5)

The SGM metric for Boards microservice of Kanban Board was calculated as follows:

\[ SGM = (0.54 \times 0.17) + (0.5 \times 0.67) + (0.54 \times 0.17) \approx 0.49 \]

where the value of the SGM metric for Boards microservice indicates a good score because the microservice is not too fine-grained or too coarse-grained, which means this microservice has a good design.

In order to get the granularity for the whole microservices application, an average of SGM of all the microservices is considered where SGM is greater than zero, and the number of microservices is greater than zero as well, so the Average Service Granularity Metric is defined as follows, where NS is the number of microservices:

\[ ASGM = \frac{\sum_{i=1}^{n} SGM_i}{NS} \]  

(3.6)

where the value of ASGM for the Kanban Boards application, which is presented in Table 3.1, can be calculated as follows:

\[ ASGM = (0.25 + 0.2 + 0.49)/3 \approx 0.31 \]

which shows the overall granularity score for the Kanban Boards application, which is 0.31, so in this case, it means this application has a good granularity score.

### 3.2.3 Proposed Number of Operations Per Microservice Metric “NOO”

The number of operations per service is the number of member operations related to one microservice (see Eq. 3.7). Similar to the WMC metric [42], which considers the number of member methods related to a specific class as a complexity metric, a higher number of methods leads to higher complexity. In microservices, the number of operations related to a specific microservice is the complexity indicator for the microservices application.

\[ NOO = \sum_{i=1}^{n} M_i \]  

(3.7)

where \( M \) is the number of operations per service. The higher the number of this metric, the more error the application may produce.

In Table 3.1, considering Boards microservice as an example, the NOO value for this microservice is going to be 3 because it has 3 operations, which are getBoardUsingGET, saveBoardUsingPOST, and listAllBoardsUsingGET. On the other hand,
the Tasks’ microservice NOO value is 9 because it has 9 operations, which means that Tasks microservice is more complex than Boards microservice, which can lead Tasks microservice to produce more errors than the board’s microservice.

Figure 3.1 shows an example of how low LCOM looks like in terms of API design and classes composition. This decomposition results in a very high cohesion and low complexity of 3 and high granularity of 0.67, which these metrics combination are not really desirable. These examples are fabricated using API endpoints from Kanban Boards Application.

<table>
<thead>
<tr>
<th>Microservice Candidate API</th>
<th>Classes Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>backlogTaskUsingPUT</td>
<td>moveToBacklogTaskCommand.java</td>
</tr>
<tr>
<td>completeTaskUsingPUT</td>
<td>TaskBacklogEvent.java</td>
</tr>
<tr>
<td>scheduleTaskUsingPUT</td>
<td>BacklogResponse.java</td>
</tr>
<tr>
<td></td>
<td>AuditEntry.java</td>
</tr>
<tr>
<td></td>
<td>Task.java</td>
</tr>
<tr>
<td></td>
<td>CompleteTaskCommand.java</td>
</tr>
<tr>
<td></td>
<td>ScheduleTaskCommand.java</td>
</tr>
</tbody>
</table>

LCOM = 0.17  NOO = 3  SGM = 0.67

In Figure 3.2, a composition of classes that lead to high LCOM metric reading, which means a low cohesion design. Also, a complexity of 7 is observable in the decomposition and the granularity metric is low with 0.24, these readings are also undesirable in terms of optimal design.

### 3.3 Results and Discussion

This section of the chapter presents the acquired results from calculating the metrics scores on five different applications. Two of these applications are considered small, one of them is medium-sized, and the other two are large industrial applications. First, the Kanban Board application\(^2\), the application consisted of three microservices and 13 operations. The application is a sample application written in Java; it allows users to create Kanban boards and tasks. Second, The Money Application\(^3\), which is a simple money transfer application, gives the ability to its users to create and view banking accounts and transfer money between them. Third, Galileo is “an analytics platform for APIs, Microservices, and Serverless Software”\(^4\). Fourth, PayPal\(^5\) is a

\(^2\)https://github.com/eventuate-examples/es-kanban-board
\(^3\)https://github.com/cer/event-sourcing-examples
\(^4\)https://www.programmableweb.com/api/galileo-rest-api-v200
\(^5\)https://developer.paypal.com/docs/api/overview/
platform that enables users to facilitate payments between customers using online transfer.

Fifth, Amazon Web Services\(^6\) is a cloud services provider platform that offers multiple services for its users, such as content delivery networks, database storage, computational power, and others. The data for the experiments were collected from the APIs of the previously mentioned applications. These data were the operations of each microservice, the type of these operations were POST, PUT, DELETE, or GET, the number of parameters of these operations, and the number of microservices for each application.

A descriptive statistical table of the tested applications’ metrics are presented in

---

\(^6\)https://docs.aws.amazon.com/AWSEC2/latest/APIReference/OperationList-query-ec2.html
Table 3.2, which shows some interesting points regarding the metrics. For example, LCOM values are affected by the number of operations and the number of microservices in the application, so this metric is affected by the size of the application or the complexity. For instance, maximum values for LCOM metric in the small and medium-sized applications are ranged from 0.48 to 0.68, while the maximum values of this metric in the big applications ranged from 0.88 to 0.89. This can show some relation between NOO and LCOM metric, which means big applications tend to be less cohesive than smaller applications. On the contrary, for the SGM metric, the maximum values for this metric range between 0.25 to 0.76 without any clear pattern. This can mean that there is an effect of the number of parameters that the microservice operations have and the type of these operations on the value of SGM.

Furthermore, some observations regarding the descriptive summary of all applications in Table 3.2 were the followings:

- The highest value of metric NOO is 24, which means that the largest number of operations per microservice is 24.

- LCOM with the lowest value of 0, which may refer to a microservice with only one operation. Therefore, it has a high cohesion because it has a single operation in the microservice.

These metrics’ final results are ANOO, ALCOM, and ASGM, showing the overall value of these metrics for the whole application. Table 3.3 presents these metrics for all the tested applications, where I noticed that ALCOM ranged from 0.25 to 0.53. For ASGM, the range of these metrics starts from 0.35 and ends by 0.58. From Table 3.3, we can see that the application’s size affects the cohesion between the microservices of the application. The higher the number of microservices, the less cohesive the application will be.
3.3. RESULTS AND DISCUSSION

Table 3.3: All the Metrics Values of Overall Metrics

<table>
<thead>
<tr>
<th></th>
<th>LCOM</th>
<th>ASGM</th>
<th>ANOO</th>
<th># of microservice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kanban</td>
<td>0.25</td>
<td>0.43</td>
<td>4.33</td>
<td>3</td>
</tr>
<tr>
<td>Money</td>
<td>0.3</td>
<td>0.58</td>
<td>2.75</td>
<td>4</td>
</tr>
<tr>
<td>Galileo</td>
<td>0.38</td>
<td>0.39</td>
<td>4.5</td>
<td>8</td>
</tr>
<tr>
<td>PayPal</td>
<td>0.57</td>
<td>0.35</td>
<td>7.13</td>
<td>15</td>
</tr>
<tr>
<td>AWS</td>
<td>0.53</td>
<td>0.39</td>
<td>6.35</td>
<td>52</td>
</tr>
</tbody>
</table>

Figure 3.3 shows that most microservices have a small number of operations ranges between 1-10. Most microservices seem to be simple in terms of the number of operations, which means they will provide specific abstraction and functionality. The largest microservice has 24 operations, which is a part of the AWS application, and it is the largest application in terms of size that have been used in this research. It has a 0.84 LCOM value and 0.08 SGM value, which means too fine-grained and not cohesive design.

A microservice with a high LCOM value tends to have more diverse functionality than microservices with a low LCOM. A microservice with a high LCOM value mainly refers to a microservice trying to accomplish many different objectives. Therefore, they are prone to be less predictable than microservices with lower LCOM values.
These microservices can produce more errors and can be more difficult in testing, so it is better to divide them into multiple, more specific microservices. According to Chidamber and Kemerer [57], “Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.” Developers can utilize the LCOM metric as a reasonably simple way to check whether the cohesion principle is followed in designing an application and recommend changes, if needed, at an earlier stage in the design phase.

In Figure 3.4, a histogram shows the SGM metric values of all the tested microservices applications. Most of the microservices have an SGM value between 0.2 to 0.29. This is because most of the tested microservices have been designed according to well-known industrial applications such as Amazon and PayPal. Also, microservices with a single operation have a 0.25 SGM value, so if these microservices are not included in the results, they will give different results.

![Histogram of All SGM Values](image)

**Figure 3.4: Histogram of All SGM Values**

### 3.4 Analysis

After showing all the tested applications’ acquired results and how the metrics are affected by different characteristics of these applications, the results of these tests need to be analyzed. First, a scatter plot presenting the values of metrics LCOM and SGM for all the microservices found in the five applications to identify the correlation between these metrics and the number of operations per microservice. In Figure 3.5,
it is clear that there is a correlation between the metric NOO and the other two metrics LCOM and SGM. In Table 3.4, a correlation matrix for all the microservices has been calculated using the correlation coefficient, and it shows that there is a positive relation between NOO and LCOM, which means that the higher the number of operations in a microservice, the higher the value of LCOM metric. Hence, the more complex the application is, the less cohesive it will be. Therefore, there is a negative relation between cohesiveness and complexity. On the other hand, the relation between NOO and SGM is negative, which means the higher the number of operations in a microservice is, the lower the SGM value. Lastly, the correlation between LCOM and SGM is negative, which means the higher value of LCOM, the lower the SGM value for a microservice. Therefore, granularity and cohesiveness are related to each other; see Table 3.4.

In another experiment, removing the microservices with a single operation from the measured cases shows a different correlation between the metrics. For example, the negative relation between SGM and LCOM was more apparent than the results that include all the tested microservices. Table 3.5 shows the correlation between all
Table 3.5: Correlation Matrix of All Metric Values without Single Operation Microservices

<table>
<thead>
<tr>
<th></th>
<th>NOO</th>
<th>LCOM</th>
<th>SGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOO</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCOM</td>
<td>0.73</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SGM</td>
<td>-0.81</td>
<td>-0.75</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.6: Correlation Matrix for All the Microservices Applications

<table>
<thead>
<tr>
<th></th>
<th>ANOO</th>
<th>ALCOM</th>
<th>ASGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANOO</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALCOM</td>
<td>0.89</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ASGM</td>
<td>-0.69</td>
<td>-0.30</td>
<td>1</td>
</tr>
</tbody>
</table>

the microservices without including the microservices’ results with a single operation because these types of microservices can be considered outliers.

Additionally, another correlation matrix among the values of this chapter’s overall microservices applications is presented in Table 3.6. It shows that the correlation between the overall metrics is similar to the correlation between the individual microservices’ metrics in each application.

In order to find the optimal scale for these metrics, density distribution was used with the values of all metrics reading from every 82 microservices of 5 different applications. For example, LCOM metrics range between 0 to 0.8, 17 microservices were with metrics values between 0.5 to 0.59, 18 microservices were calculated with 0.6 to 0.69, and 10 microservices with values between 0 to 0.09, as it is shown in Figure 3.6. Based on that, a microservice with 0.8 and higher LCOM consisting of more than 10 operations needs to be divided into more microservices because this type of microservices probably tends to be more error-prone.
The optimal scale for the SGM metric was also defined. A density distribution was applied using all the readings from 82 different microservices. 28 microservices were between 0.2 to 0.29 for the SGM metric, 11 for both values between 0.4 to 0.49 and 0.5 to 0.59, and values between 0.3 to 0.39 were found in 10 microservices, as it is shown in Figure 3.7. Therefore, the optimal range for this metric ranges from 0.2 to 0.6 for the SGM metric.
3.5 Validation

This section includes the validation process of these metrics using Weyuker’s properties [58]. These properties are widely used and well-known in terms of validating software metrics [44], [59–62]. Many researchers use these properties to validate object-oriented software metrics. There is a version of the same properties in this research but applied to microservices instead of classes. These properties are:

- Property 1: there are microservices $P$ and $Q$, also metric $m$ for which $m(P) \neq m(Q)$. This means it is not possible for every microservice to have the same value for a metric.

- Property 2: there should be a finite number of microservices having an identical metric score. If $c$ is equal to zero or a positive number, so there are finite microservices $P$ for which $m(P) = c$.

- Property 3: there can be two different microservices with the same metric value such as $m(P) = m(Q)$.
3.5. VALIDATION

- Property 4: if there are two microservices with the same functionality, that does not mean they have the same metric values, for which $m(P) \neq m(Q)$.

- Property 5: for any microservice $P$ and $Q$, we must have $m(P) \leq m(P; Q)$ and $m(Q) \leq m(Q; P)$, this means that the combination of two microservices cannot have less metric value than one microservice.

- Property 6: for microservices $P$, $Q$, and $R$, if $m(P) = m(Q)$ that does not mean $m(P; R) = m(Q; R)$. This means that the interaction between microservices can be different depending on the functionality of these microservices.

- Property 7: “A measure is sensitive to the permutation of classes. This property requires that permutation of elements within the item being measured can change the metric value.” [58]

- Property 8: renaming a microservice will not affect the metric values.

- Property 9: if two microservices are combined there is a possibility that their metric value will increase, for which, $m(P) + m(Q) < m(P; Q)$.

The 1st, 2nd, 3rd, 4th, 6th, and 8th properties are satisfied by all the metrics measures because these properties are general in nature; therefore, these properties are satisfied by many metrics. For example, property 1 states that not all microservices metrics can have the same value, which is already true by the results that were obtained. For property 2, there should be a fixed number of microservices in an application, which is the general case. For property 3, it is reasonable to have two different microservices with the same value metric, because large systems have a lot of microservices deployed. So, it is reasonable to assume that there might be two microservices with similar operations and parameters. For property 4, it is possible to have two microservices with same functionality but different metric values.

For property 6, if two microservices have the same metric value, that does not mean that these two microservices will have the same metric value if there will be a microservice combined with them.

For the ninth property, using the operations in Table 3.1, for example, authentication microservice have only one operation with 1 NOO, 0 LCOM, and 0.25 SGM, after adding another operation to the microservice the metric values become 2 NOO, 0.44 LCOM, and 0.6 SGM. Therefore, the ninth property is satisfied because combining more operations will increase the value of the metrics.

For the fifth property, we can use the previous example of the ninth property. Adding a different operation to the authentication microservice gives different metrics values, which are 2 NOO, 0.5 LCOM, and 0.44 SGM. These values are higher than the values of the microservice without combining any other operation to the microservice, and these values are different from the combination of other operations with the authentication microservice. Therefore, the fifth property is satisfied by all the metrics too.
Property 7 is not satisfied with the proposed metrics. Because this property is related to traditional software design methods, so it is not applicable in a microservices architecture. This exception has been found in other research papers [44], [59–62].

3.6 Conclusion

This research has created and developed a new set of software metrics for microservices architecture. It proposes software metrics to measure the granularity, cohesion, and complexity of individual microservices based on the application programming interface and evaluate different granularity levels, cohesion, and complexity using a quantified scale. Using this quantification allows software developers to evaluate the overall quality metrics in the microservices application. Also, the relations between the proposed metrics have been investigated, and it was shown that there was a clear negative relationship between cohesion and complexity. Also, there is another relation between LCOM and SGM. It appears to be a negative relation. The granularity metric depends on the number of operations and the number of attributes in these operations, and the type of these operations. The complexity of the microservices applications in this research depends on the number of operations per microservice. Finally, the proposed metrics were validated using Weyuker’s properties, which are well-known properties for validating software metrics in object-oriented programs, but in this research, these properties were modified to apply to microservices applications. The proposed metrics satisfied all the properties except the 7th property, which could not be applied in a microservices environment.

In conclusion, it was defined that the value of the LCOM metric for a microservice needs to be between 0 and 0.8 with less than 10 operations per microservice. If the value of LCOM is higher than 0.8 and the NOO value is higher than 10, this microservice needs to be decomposed into multiple microservices. Also, the SGM metric value needs to be between 0.2 and 0.6 because we do not need too fine-grained or too coarse-grained microservices that leads to exhausting the resources of the system. Too fine-grained microservices can be identified when having a high NOO value, usually higher than 10, and having a low value of SGM, usually lower than 0.2.

Publications related to this thesis: [JWE20] [SOSE20]
Chapter 4

*Thesis II*: A New Decomposition Method for Designing Microservices
CHAPTER 4. THESIS II

4.1 Introduction

Microservices’ decomposition is the process of converting an existing monolithic application to microservices application architecture. This is done through extracting services from the monolithic application into candidate microservices [17].

I have created a method for decomposing monolithic applications into microservices by identifying functions related to each other based on their semantic similarity. Furthermore, clustering of these similar functions is also performed to create actual microservice candidates. In the proposed decomposition method, the monolithic application is considered in the first step of the decomposition process because microservices will use the monolithic application’s functions by grouping similar functions together.

Finding proper microservices can lead to easier maintainability and scalability of a software [63]. Therefore, microservices decomposition is an essential phase in migrating from monolithic architecture to a microservice architecture. The decomposition method or extraction method is required to identify microservice candidates from the monolithic application in this process. The decomposition itself is an essential process for the whole experience of migrating to a microservice architecture because creating inappropriate microservices can lead to performance issues in the application and problems with governance policies at the organization. Thus, the properties of microservices like granularity, loose coupling, and high cohesion must be maintained during the decomposition [17].

This research aimed to provide a new methodology, using word embedding and hierarchical clustering method, to identify microservice candidates in a monolithic application by analyzing the application programming interface (API). OpenAPI specification was used, a format description for REST APIs that is usable for automatic processing. OpenAPI specification contains endpoints, operation names, operation parameters, inputs, and outputs of operations. Hence, I used the OpenAPI specification to obtain operation names widely used by developers and are fully machine-readable.

4.2 Methodology

The decomposition approach introduced in this research consists of the OpenAPI specifications as an input, then extracts the operation names from the specifications and converts them into average word embedding using a given model. fastText [64] and Word2Vec [65] models were utilized to obtain word vectors from the operation names. To obtain word representation, a vector is created from input tokens by searching a word embedding model [66]. Algorithm 1 presents a general overview of the proposed decomposition method.

In this research, Word2Vec [66] was trained on Google News corpus, and fastText was trained on different datasets. Nevertheless, before converting the operation names into vectors, removing the stop words was an initial step because keeping stop words

---

1https://www.openapis.org/
4.2. METHODOLOGY

**Algorithm 1:** The proposed decomposition algorithm

<table>
<thead>
<tr>
<th>Data:</th>
<th>OpenAPI Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result:</td>
<td>microservices’ candidates</td>
</tr>
</tbody>
</table>

1. sentences $\leftarrow \emptyset$
2. **foreach** operationName **do**
   3. sentences $\leftarrow$ ConvertToLowerCase(operationName);
   4. sentences $\leftarrow$ RemoveStopWords(sentences);
   5. sentences $\leftarrow$ ShorttextToVector(operationName);
3. **end**
4. microserviceCandidates $\leftarrow$ AffinityPropagation(sentences);
5. **return** microserviceCandidates

lead to inaccurate results. Also, a list of specific words was created to be removed from the operation names because they can change the meaning of the sentence or the operation name in this context. For example, the word “post,” “get,” “update,” and others, which can be found in many operation names.

### 4.2.1 Word Embedding Models

The Word2Vec embedding model combines two different algorithms, the continuous bag of words (CBOW) and skip-gram. The skip-gram model is an efficient method for learning high-quality distributed vector representations that capture many precise syntactic and semantic word relationships. The Skip-gram model’s goal is to find word embedding that is good for predicting the semantically closest words in a large corpus of text [66].

fastText [64] is a library for word embedding and sentence classification created by Facebook Research Group. It is also an evolution of the Word2Vec model.

In this research, two pre-trained models were used for word representations with two different text corpora. First one is the Word2Vec model with Google News corpus, the second model is the fastText with one million-word vectors trained on Wikipedia 2017. The purpose of using fastText and Word2Vec models is to create a sentence level embedding or operation’s name level embedding as used in this research.

### 4.2.2 Operation Name Vector

Usually, operation names have more than one word in their names. Therefore, in order to convert the operation name to a word representation vector, I utilized a method of getting the sum of each word vector in the operation name and dividing it by the number of words in the same operation name. In other words, getting the average of word vectors in an operation name as it was proposed by Le and Mikolov [67]. Thus, this method returns the average of all word representations for each operation name. The output needs to be fed by a clustering algorithm such as Affinity Propagation after that.
4.2.3 Clustering Method

The clustering is applied after converting the operation names into vectors based on the word embedding of the pre-trained models and removing stop words. The clustering method was utilized to group similar operation names together to create a microservice candidate consisting of these similar operation names. The Affinity Propagation [68] algorithm was used because it defines the number of clusters without the need to specify it beforehand. This clustering algorithm will find the number of microservices that are going to be created from the API of a monolithic application.

Affinity Propagation works by passing messages between data points; also, it finds exemplars, which are unique data points representing the clusters, and each cluster has one exemplar [68]. The purpose of these messages is to find the willingness of the data points to be exemplars. These exchanged messages between the data points are divided into two types. The first type is “responsibility” messages, which are messages sent from data points to candidate exemplars to show if the data points are suitable for being a member of the candidate exemplar’s cluster. The similarity $s(i, k)$ implies that how well a data point with index $k$ is capable to be an exemplar for the data point $i$. The “responsibility” denoted by $r(i, k)$ in 4.1 indicates if point $k$ is suited to be an exemplar for point $i$. Responsibilities are exchanged from point $i$ to exemplar to be $k$:

$$r(i, k) \leftarrow s(i, k) - \max_{k' \neq k} \{a(i, k') + s(i, k')\} \quad (4.1)$$

The second type is “availability” messages, which are messages sent from candidate exemplars to other data points, demonstrating if the candidate exemplar is suitable to be an exemplar. The “availability” denoted by $a(i, k)$ indicates if point $i$ can choose point $k$ as an exemplar. Availabilities are exchanged between exemplar candidate $k$ and data point $i$ starting from $k$:

$$a(i, k) \leftarrow \min \left\{0, r(k, k) + \sum_{i' \notin \{i, k\}} \max \{0, r(i', k)\} \right\} \quad (4.2)$$

Self-availability is updated in a different way, which can be seen in 4.3:

$$a(k, k) \leftarrow \sum_{i' \neq k} \max \{0, r(i', k)\} \quad (4.3)$$

Then the algorithm finds pairwise similarities between the data points, and it will identify the clusters by maximizing the total similarity between the exemplars and their data points.

Mézard [69] explained the importance and efficiency of message passing algorithms even on complicated problems. Hence, Affinity Propagation was utilized in this decomposition method for clustering similar operation names to create candidates for microservices.

In Affinity Propagation algorithm there are three parameters which are related to the performance of the algorithm:
4.2. METHODOLOGY

1. The first parameter is damping, which damps the exchange of messages between responsibility and availability to prevent numerical oscillations while updating the values of responsibilities and availabilities [70].

2. The second one is the maximum number of iterations.

3. The third one is the number of iterations with no change in the number of estimated clusters that stops the convergence.

Algorithm 2 summarizes all the steps in Affinity Propagation algorithm.

**Algorithm 2:** Affinity Propagation algorithm

- **Data:** \( \{s(i,j)\}_{i,j \in \{1,\ldots,N\}} \) data similarities and preferences
- **Result:** cluster assignments \( \hat{c} \)

1. Availability ← 0
2. repeat
3. \( r(i,k) \leftarrow s(i,k) - \max_{k' \neq k} \{a(i,k') + s(i,k')\} \)
4. \( a(i,k) \leftarrow \min \left\{ 0, r(k,k) + \sum_{i' \notin \{i,k\}} \max \{0, r(i',k)\} \right\} \)
5. if \( k \neq i \) then
6. \( a(k,k) \leftarrow \sum_{i' \neq k} \max \{0, r(i',k)\} \)
7. end
8. until convergence;
9. return \( \hat{c} = \arg\max_k [a(i,k) + r(i,k)] \)

### 4.2.4 Evaluation Metrics

Silhouette coefficient [71] is a method for validating data consistency within clusters. It measures the similarity between an object and its cluster compared to other clusters. Silhouette coefficient score ranges from -1 to 1; this means an object is matched correctly to its cluster when it has a value of 1 for its silhouette coefficient \( s(i) \).

This method was used to evaluate the clustering algorithm’s performance while using different values for the algorithm’s parameters.

\[
s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \tag{4.4}
\]

where \( a(i) \) represents the average dissimilarity of object \( i \), to all other objects of the same cluster. \( b(i) \) represents the smallest average distance between object \( i \) and any other object in other clusters.

Also, grid search was used to try different combinations of the three parameters of the clustering algorithm to find the optimal values for these parameters. Precision, recall, and F-measure were the metrics used to evaluate the proposed decomposition method’s performance.

The precision for the clustering method used in this research is the averaged precision of each object or, in this case, of each operation name. The precision \( P(O_O) \)
of a given object \( O \), which will find the precision of the object \( O \) in a computed cluster compared with the ideal cluster of the same object \( C(O) \):

\[
P(O) = \begin{cases} 
\frac{|S_\tau(O) \cap C(O)| - 1}{|C(O)| - 1} & |C(O)| > 1 \\
1 & |C(O)| = 1
\end{cases}
\]  \hspace{1cm} (4.5)

where \( S_\tau(O) \) denotes the object related to the same ideal cluster that the selected object \( O \) belongs to. After calculating the precision of every object, the precision of the clustering \( P \) is the average of the precisions of every object.

The recall \( R(O) \) of a given object, \( O \), which will find the recall of the object \( O \) in a computed cluster compared with the ideal cluster of the same object \( C(O) \):

\[
R(O) = \begin{cases} 
\frac{|S_\tau(O) \cap C(O)| - 1}{|C(O)| - 1} & |C(O)| > 1 \\
1 & |C(O)| = 1
\end{cases}
\]  \hspace{1cm} (4.6)

where \( S_\tau(O) \) denotes the object related to the same ideal cluster that the selected object \( O \) belongs to. After calculating the recall of every object, the recall of the clustering \( R \) is the average of the recalls of all the objects.

Furthermore, F-Measure \( F1 \) was used to get the harmonic mean of recall and precision, where 1 represent the best value and 0 represent the worst result. F-Measure can be calculated as follows:

\[
F1 = 2 * \frac{P * R}{P + R}
\]  \hspace{1cm} (4.7)

4.3 Results and Discussion

For the implementation of the algorithm in this research, Python\(^2\) programming language was utilized with specific libraries for text analysis and clustering such as Gensim [72], NLTK [73], and Sklearn [74].

To find the optimal values for the Affinity Propagation algorithm parameters like damping, the maximum number of iterations, and convergence iterations, a grid search approach was used. The test cases were the APIs of four different applications, Amazon Web Services, PayPal, Kanban Board, and Money Transfer app. Furthermore, the Silhouette coefficient (SC) was utilized to evaluate the clustering algorithm’s performance with different parameter values. The parameter values were a range of numbers. First, damping (DA) value started from 0.5 until 0.9 with 0.1 step size. Second, maximum iteration (MI) ranges from 100 to 1000 with 100 step size. Finally, convergence iteration (CI) ranges from 10 to 100 with 10 step size. Eventually, grid search results showed that the parameters’ optimal values were 0.6 for damping, 300 for maximum iteration, and 50 for convergence iteration. Table 4.1 shows the highest values of the average Silhouette coefficient score, and the parameter values were used to achieve these results.

\( ^2 \)http://python.org
Table 4.1: Average Silhouette coefficient scores

<table>
<thead>
<tr>
<th>App</th>
<th>DA</th>
<th>MI</th>
<th>CI</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS</td>
<td>0.6</td>
<td>300</td>
<td>50</td>
<td>0.50</td>
</tr>
<tr>
<td>Money</td>
<td>0.6</td>
<td>300</td>
<td>50</td>
<td>0.40</td>
</tr>
<tr>
<td>PayPal</td>
<td>0.6</td>
<td>300</td>
<td>50</td>
<td>0.43</td>
</tr>
<tr>
<td>Kanban</td>
<td>0.6</td>
<td>300</td>
<td>50</td>
<td>0.55</td>
</tr>
</tbody>
</table>

Figure 4.1 illustrates the Silhouette coefficient score of each operation name and its cluster for PayPal’s API using the Affinity propagation algorithm with the optimal parameters that were obtained previously. The average Silhouette coefficient was 0.43.

Once the Affinity Propagation algorithm’s optimal parameters were chosen, there was a need to choose which word embedding model should be used for the proposed decomposition method. Several tests with 4 different test cases were conducted. F-Measure was used to compare the performance between Word2Vec and fastText models.
As shown in Figure 4.2, the performance of fastText was better than the performance of Word2Vec in terms of F-Measure. Therefore, fastText was selected to conduct the final tests with the 4 different test cases.

![Figure 4.2: Performance of Word2Vec vs fastText](image)

After performing several tests, 4 different OpenAPI specifications of different applications were evaluated. These applications were the Money Transfer application with 11 operations and 4 microservices, the Kanban Board application, which contained 13 operations and 3 microservices. Both of them were created by Chris Richardson, the author of Microservice Patterns [75] book, serving as a good standard for evaluating the performance of the decomposition method presented in this chapter.

Subsequently, the proposed decomposition method’s application on these two examples gave an excellent result of 100 % precision and 85 % recall for Kanban Board application, and 82 % precision and recall for Money Transfer application. Table 4.2 compares the proposed decomposition using the proposed method against the two applications’ standard design. Comparing these results with the results of Baresi et al. [17], the proposed method performed better in the decomposition of the Kanban Board application by decomposing 12 out of 13 operations correctly, in comparison to the method of Baresi et al. [17], which decomposed only 10 operations correctly. For the Money Transfer application, 10 of the 11 operations were decomposed correctly using the method, while in the research of Baresi et al. [17] only 8 operations were found correctly during the decomposition. Consequently, the proposed method showed a better performance when compared to other methods in the literature, as research of Baresi et al. [17] showed already that their method performed better than Service Cutter [14].

Furthermore, additional test cases were created with real-life examples of applications already used in a real-life environment. Thus, I searched for companies using microservice architecture in their applications; for example, Netflix, Amazon, PayPal, Twitter, and others [76]. Eventually, Amazon Web Services and PayPal were selected as a case study for the proposed decomposition methodology because their API was available in OpenAPI specifications definition. So, these were compatible with the proposed method.
The number of operations in Amazon Web Services API was 318 divided into 47 microservices. On the other hand, PayPal’s API has 110 operations scattered on 16 microservices. This shows the applications’ size and how challenging it is to decompose them manually without any automation.

<table>
<thead>
<tr>
<th>Application</th>
<th>Proposed Decomposition</th>
<th>Optimal Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Money Transfer</td>
<td>addToAccountUsing</td>
<td>addToAccountUsing</td>
</tr>
<tr>
<td></td>
<td>createAccountUsing</td>
<td>createAccountUsing</td>
</tr>
<tr>
<td></td>
<td>createCustomerUsing getAccountsForCustomerUsing</td>
<td>createCustomerUsing getAccountsForCustomerUsing</td>
</tr>
<tr>
<td></td>
<td>getCurrentUserUsing</td>
<td>getCurrentUserUsing</td>
</tr>
<tr>
<td></td>
<td>getCustomerUsing</td>
<td>getCustomerUsing</td>
</tr>
<tr>
<td></td>
<td>getCustomersByEmailUsing</td>
<td>getCustomersByEmailUsing</td>
</tr>
<tr>
<td></td>
<td>getTransactionHistoryUsing</td>
<td>getTransactionHistoryUsing</td>
</tr>
<tr>
<td></td>
<td>doAuthUsing</td>
<td>doAuthUsing</td>
</tr>
<tr>
<td></td>
<td>moneyTransferUsing</td>
<td>moneyTransferUsing</td>
</tr>
<tr>
<td>Kanban Board</td>
<td>getBoardUsing</td>
<td>getBoardUsing</td>
</tr>
<tr>
<td></td>
<td>listAllBoardsUsing</td>
<td>listAllBoardsUsing</td>
</tr>
<tr>
<td></td>
<td>saveBoardUsing</td>
<td>saveBoardUsing</td>
</tr>
<tr>
<td></td>
<td>backlogTaskUsing</td>
<td>completeTaskUsing</td>
</tr>
<tr>
<td></td>
<td>completeTaskUsing</td>
<td>deleteTaskUsing</td>
</tr>
<tr>
<td></td>
<td>deleteTaskUsing</td>
<td>listAllTasksUsing</td>
</tr>
<tr>
<td></td>
<td>listAllTasksUsing</td>
<td>saveTaskUsing</td>
</tr>
<tr>
<td></td>
<td>saveTaskUsing</td>
<td>scheduleTaskUsing</td>
</tr>
<tr>
<td></td>
<td>scheduleTaskUsing</td>
<td>startTaskUsing</td>
</tr>
<tr>
<td></td>
<td>startTaskUsing</td>
<td>updateTaskUsing</td>
</tr>
<tr>
<td></td>
<td>updateTaskUsing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getHistoryUsing</td>
<td>getHistoryUsing</td>
</tr>
</tbody>
</table>

The proposed method presented promising results in terms of precision and recall by performing the decomposition process on these two APIs and comparing the decomposition results with the already available services in each application’s API documentation. For example, the proposed decomposition method’s precision was 74 % and recall was 79 % obtained from decomposing Amazon Web Services API. For PayPal API, the performance was less accurate, precision was 80 % while recall was
CHAPTER 4. THESIS II

66 %. Table 4.3 shows the results of all tests using the 4 different applications. Accordingly, in total, there were 4 applications with 452 operations tested using the proposed decomposition method. By getting the precision and recall, F-measure was calculated as was mentioned before. The averaged F-Measure was 81 % while the averaged precision of all the tests was 84 % and the averaged recall was 78 %. These results showed that the proposed decomposition method is suitable to be a helping tool for software architects by decomposing a monolithic application into a microservices application.

<table>
<thead>
<tr>
<th>Application</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Measure</th>
<th># of Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS</td>
<td>0.74</td>
<td>0.79</td>
<td>0.76</td>
<td>318</td>
</tr>
<tr>
<td>Kanban Board</td>
<td>1</td>
<td>0.85</td>
<td>0.92</td>
<td>13</td>
</tr>
<tr>
<td>Money Transfer</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>11</td>
</tr>
<tr>
<td>PayPal</td>
<td>0.8</td>
<td>0.66</td>
<td>0.72</td>
<td>110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Precision Average</th>
<th>Recall Average</th>
<th>F-Measure Average</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.84</td>
<td>0.78</td>
<td>0.81</td>
<td>452</td>
</tr>
</tbody>
</table>

One limitation of the proposed method is that the operation names need to be clearly defined and in a correct case, and in the case of this research was camel case operation names. If the operation name is not written in a correct way it will be considered as one word which will result in the word embedding model to miscalculate the correct vector of the operation name.

4.4 Combining API Decomposition Method with LCOM and NOO Metrics

This section shows the possibility of extending the proposed decomposition method using two metrics mentioned in Chapter 3. This section proposes a decomposition method based on analyzing the application programming interface “API” through extracting the operations and their parameters, then cluster these operations based on their semantic similarities and propose microservices candidates. Next, evaluate these candidates using lack of cohesion and complexity metrics. Afterward, the algorithm decides if they are appropriate microservices or need more decomposition.

4.4.1 Methodology

The proposed decomposition method consists of three main steps:

1. Extracting information from API and finding semantic similarities.
2. Clustering similar operations to provide microservices candidates.
3. Evaluate the microservices candidates through evaluation metrics.
4.4. COMBINING API DECOMPOSITION METHOD WITH LCOM AND NOO METRICS

The first two steps are the decomposition method that is mentioned in this chapter. So, the Affinity Propagation algorithm clusters semantically similar operations together to create a microservice candidate. These candidates need to be evaluated through some methods. Therefore, lack of cohesion and complexity metrics are used in the next step of the proposed algorithm. The third step of the proposed algorithm is evaluating the proposed microservice from the second step. In order to do that, two metrics are being used, which are lack of cohesion and complexity metrics from chapter 3.

The lack of cohesion metric “LCOM” works by finding how many times a microservice has used a specific operation’s parameter, divided by the product of the number of operations multiplied by the number of unique parameters, see Equation 3.1. The other metric is the Number of Operations which measures the complexity of the proposed microservices by measuring the operational functionality by counting the number of operations included in the microservice. Check Equation 3.7 on how this metric works.

These two metrics checks if the decomposed microservice is suitable for the final phase of a production application. By measuring these metrics against multiple applications' API, a threshold was determined for each metric. The LCOM metric threshold should be between 0 to 0.8, and the value of NOO should be 10 or less, and if it was more than those proposed values, the microservice needs to be decomposed further into two or more microservices. Figure 4.3 shows a description of the proposed decomposition algorithm. These thresholds were found using density distribution through 5 different microservices applications consisting of 82 microservices.

Two applications were used to test the proposed decomposition method. The first one is a Kanban Boards application and the second application is a Money Transfer application. The results of the decomposition method were similar to the actual microservices design that was proposed by the original author. The only difference is the original design had 3 microservices while the proposed design by decomposition method had 4 microservices. The decomposition method clusters similar operations based on their semantic similarity, therefore; it made the operation getHistoryUsingGET as a standalone microservice. Table 4.4 presents the results of how the proposed decomposition method evaluate the proposed microservices in the second step of the decomposition method. Also, it includes a comparison with the results of Baresi’s et al decomposition method.

<table>
<thead>
<tr>
<th>Microservice Name</th>
<th>Original LCOM</th>
<th>Original NOO</th>
<th>Our Decomposition LCOM</th>
<th>Our Decomposition NOO</th>
<th>Baresi Decomposition LCOM</th>
<th>Baresi Decomposition NOO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boards</td>
<td>0.11</td>
<td>3</td>
<td>0.11</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Tasks</td>
<td>0.64</td>
<td>9</td>
<td>0.7</td>
<td>8</td>
<td>0.58</td>
<td>7</td>
</tr>
<tr>
<td>Authentication</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Extra MS</td>
<td>N/A</td>
<td>N/A</td>
<td>0</td>
<td>1</td>
<td>0.35</td>
<td>3</td>
</tr>
</tbody>
</table>

The results show that there are some similarities between the results of the evaluation metrics. For example, the Tasks microservice of Baresi’s et al decomposition
had the better results in terms of metrics evaluation, but our decomposition design was closer to the original design. On the other hand, Baresi’s et al method proposed an extra microservice that were not proposed in the original design. Also, our decomposition method did the same, but the measurements of the extra microservice is better compared to the extra microservice proposed by Baresi et al.

Table 4.5 shows the measurement results of the Money Transfer application and how the decomposition methods compares with the original design.

The proposed decomposition method had better results with the Money Transfer application in terms of decomposition because it was very close to the application’s original design. For example, Table 4.5 shows that the extra microservice created by Baresi et al. is not close to the original design, and the lack of cohesion is considered high compared to the number of operations in the microservice.
4.5. **CONCLUSION**

This research proposed a novel approach to identify microservices in migrating from monolithic architecture to a microservice architecture. The proposed method consists of several steps, starts with the extracted operation names from OpenAPI specifications. The second step is the process of converting the operation names into word representation using word embedding models. The third step is the clustering of semantically similar operation names in order to create candidates of microservices. The proposed method showed significantly better results when compared to other methods from the literature, resulting in an F-Measure of 0.81, a precision of 0.84, and a recall of 0.78. Therefore, this can be an aiding addition for software architects in extracting microservices from monolithic applications. Furthermore, I proposed a new microservices decomposition method using word embedding and hierarchical clustering method to identify potential microservices through analyzing application programming interfaces.

The extended algorithm is including an addition to the algorithm is a third step that evaluates the proposed microservice using cohesion and complexity metrics. The microservice needs to be decomposed further if its measurements exceed the threshold of 0.8 for LCOM and 10 for NOO. Microservices with a high lack of cohesion and complexity need to be decomposed more because they will make the application less stable and produce more errors. The proposed algorithm analyzes the API of a monolithic application to identify microservices candidates by clustering semantically similar API operations. This algorithm can be a helping tool for software architectures and developers to assist them in migrating from monolithic architecture to microservices architecture.

Publications related to this thesis: [CINTI18] [SOSE20] [PP19]
Chapter 5

5.1 Introduction

This research aims to tackle the issue of microservices identification using vector representation of software’s source code. Hence, this research proposed a novel approach to decompose monolith application into a microservices application by using a neural model \[1\] to represent snippets of code as continuous distributed vectors. The monolithic application code would be converted into vector representation using the provided model, and then certain classes would be grouped to provide microservices candidates.

Four monolithic applications with different sizes were decomposed using the provided methodology to verify this approach’s effectiveness. These applications were tested in other research papers before, so they are considered benchmark applications for monolithic applications’ decomposition process. This method used two different metrics to compare the proposed method’s performance with other methods from the literature. Also, this method utilized other metrics to compare the sizes of the mentioned applications in the test, such as the number of classes, number of methods, lines of code "LoC," and the number of microservices. The proposed method is a useful aiding tool for developers in-migration from monolithic to a microservices architecture, which suggests a specific direction for the decomposition process.

5.2 Methodology

Machine learning for code refactoring was used on several other software architectures before \[77–79\]. However, it can be applied in a microservices’ environment as well. This research proposes a new decomposition method for decomposing monolithic applications into microservices applications. The method uses a novel approach for microservice decomposition by using code representation to understand the similarity within the application classes and cluster semantically similar classes together to create microservices candidates. Clustering semantically similar classes together are to resemble the domain model more efficiently \[80\].

The proposed machine learning based method consists of four main steps:

1. extracting the methods and its code from the monolithic application,
2. converting the code to code embeddings or vector representations,
3. aggregating the code embeddings of one class,
4. group together semantically similar classes to obtain microservices candidates.

5.2.1 Extracting Code Embeddings

Methods are extracted from classes and converted into code embeddings using the code2vec \[1\] model. Code embeddings are snippets of codes characterized as a vector-based representation for a machine-learning algorithm to understand these snippets of codes.
Embeddings are a mapping of an object represented as vectors. For example, word embeddings are representations of a word (or sequence of words) as vectors of real numbers [81]. Word embeddings make it possible for textual data to work with a mathematical model. Code embeddings have a similar benefit to word embeddings; these embeddings can capture the source code’s semantics. These code embeddings can be used for several tasks such as malware detection, author identification, and refactoring.

5.2.2 Code Embeddings Model

The proposed method uses the code2vec model created by Alon et al. [1] to obtain code embeddings or continuous distributed vectors of the extracted methods. Code embeddings give us the ability to find a similarity between the extracted classes.

Code2vec is a deep representation learning method, which was used for predicting method names. However, code2vec code embeddings can be used in other tasks as well. Code2vec converts the source code into a set of Abstract Syntax Tree (AST) paths and sums them using an attention mechanism. The attention technique works by giving more weight to the important AST paths that represent the source code. So, the vector representation of a function is an aggregation of weighted AST paths. The attention mechanism shows the important AST paths that need more focus than the other available paths.

AST is represented with branches and leaves similar to a tree. The source code’s functional structure is represented by AST instead of a detailed description of the source code. For example, Figure 5.1 shows an AST representation of a factorial function. The utilization of AST improves the accuracy and training of a machine learning model [81].
The goal of code2vec is to generate code embeddings that keep the semantics of the source code. Code2vec represents the source code as a bag of AST paths; these paths are generated between the AST tree leaves. AST path is a path between two leaves in an AST tree. For example, the colored paths in Figure 5.1 are AST paths. Path-context is a set of three tokens, consisting of two tokens represent the two AST leaves and another token represent the path between these two leaves. For example, the red path in Figure 5.1 can be represented as follows:

\{n, \text{Times} \downarrow \text{MethodCall} \downarrow \text{Minus} \downarrow, n\}

The sign $\downarrow$ represent the path going toward the leaves while $\uparrow$ represent going toward the root of the AST tree. For more details and information check the original paper [1]. Figure 5.2 shows the architecture of code2vec model with all the processes described earlier.
5.2. METHODOLOGY

5.2.3 Aggregation Method

This step combines the code embeddings of the methods in order to reflect the representation of the class. Multiple aggregation functions were used, such as mean, sum, maximum, minimum, standard deviation, and variance. The mean function gave the best results regarding the accuracy of the clustering function in the next step. Figure 5.3 shows the process of aggregating multiple code embeddings into one vector representation using the mean function.

\[
\begin{align*}
[-0.4, 0.6, 0.4] & \quad \rightarrow \quad \text{Mean Function} & \rightarrow & \quad [-0.5, 0.17, 0.16] \\
[-0.5, 0.01, 0.6] & \quad \rightarrow \quad \text{Mean Function} & \rightarrow & \quad [-0.5, 0.17, 0.16] \\
[-0.7, -0.1, -0.5] & \quad \rightarrow \quad \text{Mean Function} & \rightarrow & \quad [-0.5, 0.17, 0.16]
\end{align*}
\]

After this step, the aggregated code embeddings are sent to the next step, the clustering method, where it will generate the microservices candidates.

5.2.4 Clustering Method

A clustering method was applied after converting the source code into code embeddings based on the code2vec model and aggregating code embeddings. Related classes
are clustered together using the clustering method in order to generate a suitable microservice candidate. The Affinity Propagation [68] algorithm was chosen for this process because it identifies the sum of clusters minus the necessity to indicate it in advance. Microservices candidates are identified using the previously mentioned methods combined with the clustering algorithm.

The Affinity Propagation algorithm is based on two concepts passing messages between data points and finding exemplars [68]. Exemplars are the cluster centers, which represent the cluster, and each cluster contains a single exemplar. Also, there are two types of these exchanged messages between the data points. The first type is exchanged between the data points and the candidate exemplars, and these types of messages are called (responsibility) messages. They are used to find the strength of the link between the data points and the exemplars.

Affinity Propagation groups similar code embeddings together in order to generate microservices candidates. The proposed microservices candidates are analyzed using cohesion metrics to be compared with the results of other decomposition methods. The Affinity Propagation algorithm is discussed in more detail in section 4.2.3.

5.2.5 Metrics for Evaluating Clustering Method Performance

Silhouette coefficient, precision, recall, and F-measure were used to determine the clustering method parameters’ efficiency and threshold. Silhouette coefficient [71] is a validation method for clustered data.

5.2.6 Evaluation Metrics

For this section, I chose metrics that were used by other researchers, as well. As a result, the comparison can be suitable with other decomposition methods. These researches [24], [25], and [23] used these metrics.

The first metric is Cohesion at Message Level (CHM), which uses the average cohesion of microservices interfaces at the message level. It is a refined version of Lack of Message Level Cohesion by Athanasopoulos et al. [38]. CHM value can be calculated, as shown in equation 5.1.

\[
CHM = \frac{\sum_{j=1}^{K} n_i CHM_j}{\sum_{i=1}^{K} n_i}
\]

where, \( CHM_j = \begin{cases} \sum_{(k,m)} f_{simM}(Op_k, Op_m) / |I_i| \times (|I_i| - 1) / 2 & \text{if } |I_i| \neq 1 \\ 1 & \text{if } |I_i| = 1 \end{cases} \)

\[
f_{simM}(Op_k, Op_m) = \left( \frac{|res_k \cap res_m|}{|res_k| \cup |res_m|} + \frac{|pas_k \cap pas_m|}{|pas_k| \cup |pas_m|} \right) / 2
\]

\( n_i \) represents the number of the interfaces of a microservice \( i \). \( k \) represents the number of microservices candidates that were generated from the monolithic application.
CHM\textsubscript{j} measures the cohesion of a microservice at the message level. \( Op_k \) and \( Op_m \) represent the operations provided by the interface \( I_i \) of a microservice. The return parameters are represented by \textit{res} and the input parameters are represented by \textit{pas}. The similarity between the output parameters and the input parameters is calculated by the similarity function \( fsimM \). The higher value of the \( CHM \) metric is the better.

The other metric is Cohesion at Domain Level (\( CHD \)), which measures the average of the interfaces’ cohesion at the domain level. It is a modified version of Lack of Domain Level Cohesion by Athanasopoulos et al. [38]. The formal definition of the metric is shown in equation 5.2.

\[
CHD = \frac{\sum_{j=1}^{K} n_i CHD_j}{\sum_{i=1}^{n} n_i}
\]

where, \( CHD_j = \left\{ \begin{array}{ll}
\frac{\sum_{(k,m)} fsimD(Op_k, Op_m)}{|I_i| * (|I_i| - 1) / 2} & \text{if } |I_i| \neq 1 \\
1 & \text{if } |I_i| = 1
\end{array} \right.
\]

\[
fsimD(Op_k, Op_m) = \frac{|T_{Op_k} \cap T_{Op_m}|}{|T_{Op_k} \cup T_{Op_m}|}
\]

\( n_i \) represents the number of the interfaces of a microservice \( i \). \( K \) represents the number of microservices candidates that were generated from the monolithic application. \( fsimD \) function calculates the similarity of the operations at the domain level. \( Op_k \) and \( Op_m \) represents the domain terms that are extracted from the operations. The higher value of the \( CHD \) metric is the better.

\( CHM \) and \( CHD \) metrics were introduced by Jin et al. [24]. These metrics are used to measure the cohesion at message and domain levels of the microservices by analyzing their interfaces.

Figure 5.4 presents a high-level description of the proposed algorithm, which starts with obtaining the methods code snippets from the monolithic application source code. Then these codes are converted to code embeddings using the code2vec model. Furthermore, aggregate the methods code embeddings using the mean function to represent each class’s code of the related methods. Finally, microservices candidates are generated through clustering related class files using a hierarchical clustering algorithm.

![Flowchart](image-url)  
**Figure 5.4: High-Level Representation of the Proposed Algorithm**
5.3 Experiments and Results

The experiment setup consists of testing four applications to compare the performance of the proposed method against other methods in the literature. The first application is JPetStore\(^1\) is a pet store commercial website written in JAVA, and it is a monolithic web application consists of 24 classes. Also, it is the smallest application in the experiment setup. The second application is SpringBlog\(^2\), a blogging website written in JAVA consisting of 46 classes. The third application is JForum\(^3\), a messaging boards application consisting of 335 classes. The last application is Apache Roller\(^4\), a monolithic application that allows multiple users to create blog sites and posts. These applications range from small to big applications with different class numbers, method numbers, and lines of codes. See a detailed comparison of the tested applications in Table 5.1, MS numbers represent the number of microservices in the application.

<table>
<thead>
<tr>
<th>Application</th>
<th>Classes</th>
<th>Methods</th>
<th>LoC</th>
<th>MS numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPetStore</td>
<td>24</td>
<td>290</td>
<td>2059</td>
<td>4</td>
</tr>
<tr>
<td>SpringBlog</td>
<td>46</td>
<td>155</td>
<td>1553</td>
<td>6</td>
</tr>
<tr>
<td>JForum</td>
<td>335</td>
<td>2702</td>
<td>52,719</td>
<td>8</td>
</tr>
<tr>
<td>Roller</td>
<td>153</td>
<td>780</td>
<td>29,154</td>
<td>11</td>
</tr>
</tbody>
</table>

5.3.1 Aggregation Method

Several aggregation methods were tested to find the most effective method for the proposed algorithm. These methods are mean, sum, standard deviation, variance, maximum, and minimum. The experiment setup consisted of comparing the accuracy, precision, and recall of the clustering results against the optimal microservices design of Spring Pet Clinic\(^5\), which have the monolithic application and the microservices design\(^6\) as well. The results of the experiment are shown in Table 5.2. Thus, the mean function is the most suitable aggregation method for this experiment because it has the highest accuracy, precision, and recall scores.

\(^1\)https://github.com/mybatis/jpetstore-6
\(^2\)https://github.com/Raysmond/SpringBlog
\(^3\)https://sourceforge.net/projects/jforum2/
\(^4\)https://github.com/apache/roller
\(^5\)https://github.com/spring-projects/spring-petclinic
\(^6\)https://github.com/spring-petclinic/spring-petclinic-microservices
5.3. EXPERIMENTS AND RESULTS

<table>
<thead>
<tr>
<th>Aggregation Method</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>Silhouette coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>0.70</td>
<td><strong>0.58</strong></td>
<td><strong>0.46</strong></td>
<td><strong>0.49</strong></td>
<td>0.47</td>
</tr>
<tr>
<td>Sum</td>
<td>0.07</td>
<td>0.07</td>
<td>0.008</td>
<td>0.015</td>
<td>N/A</td>
</tr>
<tr>
<td>Maximum</td>
<td>0.15</td>
<td>0.33</td>
<td>0.10</td>
<td>0.14</td>
<td>0.27</td>
</tr>
<tr>
<td>Minimum</td>
<td>0.15</td>
<td>0.33</td>
<td>0.10</td>
<td>0.14</td>
<td>0.23</td>
</tr>
<tr>
<td>Median</td>
<td>0.23</td>
<td>0.56</td>
<td>0.27</td>
<td>0.30</td>
<td>0.17</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.15</td>
<td>0.05</td>
<td>0.33</td>
<td>0.09</td>
<td>N/A</td>
</tr>
<tr>
<td>Variance</td>
<td>0.15</td>
<td>0.05</td>
<td>0.33</td>
<td>0.09</td>
<td>N/A</td>
</tr>
</tbody>
</table>

5.3.2 Clustering Method Parameters

The parameters for refining the Affinity Propagation algorithm’s performance are damping, the maximum number of iterations, and convergence iterations, the values for these parameters were 0.8, 500, and 50, respectively. These values were found using the grid search technique with different setups, configurations, and tests against the monolithic application Spring Pet Clinic mentioned previously. The results for these tests are displayed in Table 5.2. The tests were compared using the silhouette coefficient score.

5.3.3 Decomposition Results

After conducting the previous experiments and tests to find the most optimal aggregation method and the most efficient parameter values, it is the turn of displaying the proposed decomposition methodology’s results. As was mentioned before in Section 5.3, the decomposition method was tested with four different applications (listed in Table 5.1.)

The first application is JPetStore, which was tested by Jin et al. [24] and Saidani et al. [23]. JPetStore application was compared with Jin et al.’s approach in detail. For example, Figure 5.5 shows a comparison between our approach’s decomposition results and their approach. Our approach generated four microservices, while Jin et al.’s approach gave three microservices. Figure 5.5 displays the microservices and their related classes.

For the cohesion side of the comparison, both of the approaches have similar results, but the proposed approach has a slightly better score for CHM. These results in Table 5.3 are concerning the results of only JPetStore application because the decomposition results for JPetStore were described thoroughly in the research of Jin et al. [24].

The results for comparing the proposed method and the other methods using the additional three applications are available in Table 5.4.
The second application is SpringBlog, which consists of 46 classes. The results in Table 5.4 suggest that the proposed approach have a better performance in term of CHM metric compared to the other decomposition methods, but my approach has a less cohesive score, based on the CHD score, compared to the other approaches.

For the JForum application, the proposed method performed the best in terms of cohesion at the message and domain level, as shown in Table 5.4. It scored better scores in both CHD and CHM compared to Jin et al. and Saidani et al. methods. Therefore, this means the proposed method creates better decomposition results in terms of cohesion.

The final application is Apache Roller, where the proposed approach had slightly improved results in terms of CHD, while had a good result for CHM metric. These results show that the proposed method can handle big applications such as JForm and Apache Roller without any issues.

The overall results for tested applications suggest that the proposed approach has some advantages in cohesion in the middle and big applications. For example, Table 5.4 shows that most of the better and good metrics values were related to the introduced approach, except in the small tier application such as JPetStore. The proposed approach scored the best results in four test experiments out of 8, while Saidani et al.’s method scored 4 out of 8, and Jin et al. scored 0. These results show that all the methods have good results, but the proposed method had better ones when compared with the other methods. The proposed method showed better performance in terms of cohesion, which is one of the essential requirements for a good microservices application design because microservices applications need to be loosely coupled and cohesive, according to Newman [9].
5.3. EXPERIMENTS AND RESULTS

Table 5.4: Decomposition Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Metrics</th>
<th>Jin et al</th>
<th>Saidani et al.</th>
<th>My Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPetStore</td>
<td>CHD</td>
<td>0.52</td>
<td><strong>0.65</strong></td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>CHM</td>
<td>0.78</td>
<td>0.55</td>
<td><strong>0.82</strong></td>
</tr>
<tr>
<td>SpringBlog</td>
<td>CHD</td>
<td>0.55</td>
<td><strong>0.67</strong></td>
<td>0.50</td>
</tr>
<tr>
<td></td>
<td>CHM</td>
<td>0.68</td>
<td><strong>0.75</strong></td>
<td>0.73</td>
</tr>
<tr>
<td>JForum</td>
<td>CHD</td>
<td>0.45</td>
<td>0.15</td>
<td><strong>0.52</strong></td>
</tr>
<tr>
<td></td>
<td>CHM</td>
<td>0.70</td>
<td>0.51</td>
<td><strong>0.73</strong></td>
</tr>
<tr>
<td>Roller</td>
<td>CHD</td>
<td>0.52</td>
<td>0.38</td>
<td><strong>0.53</strong></td>
</tr>
<tr>
<td></td>
<td>CHM</td>
<td>0.72</td>
<td><strong>0.78</strong></td>
<td>0.76</td>
</tr>
</tbody>
</table>

Figure 5.6: Metrics Results Comparing the Performance of the Decomposition Methods

Figure 5.6 shows an interpretation of the results shown in Table 5.4. Figure 5.6 shows that my method is performing similar to Jin et al. [24] but in 4 cases has better performance. Also, the results of Saidani et al. [23] fluctuates between 0.1 and 0.8, while the results of the proposed method are between 0.5 and 0.8. Therefore, this means the proposed method has a more stable approach when compared to Saidani et al.’s approach.

The overall results showed that the proposed decomposition method is better performing than Jin et al. and Saidani et al. methods. For example, the proposed method had better results in 4 out of 8 metrics scores, Saidani et al. had 4, and Jin et al. ’s method performed the worst compared to the other methods. In another
interpretation of the results, Table 5.5 presents the averaged results of Table 5.4, which shows that the results of Jin et al. are better on average compared to Saidani et al., but the proposed method has the best results in this case as well.

## 5.4 Evaluation Using Chapter 3 Metrics

In this section, an evaluation of the mentioned decomposition method is made using LCOM, SGM, and NOO metrics mentioned in chapter 3. I used one of the decomposed applications using the proposed decomposition method to show how the metrics from chapter 3 can evaluate the proposed decomposition. Table 5.6 shows the results of the refactored JPetStore application.

<table>
<thead>
<tr>
<th></th>
<th>LCOM</th>
<th>SGM</th>
<th>NOO</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS1</td>
<td>0.27</td>
<td>0.30</td>
<td>7</td>
</tr>
<tr>
<td>MS2</td>
<td>0.3</td>
<td>0.61</td>
<td>4</td>
</tr>
<tr>
<td>MS3</td>
<td>0.6</td>
<td>0.14</td>
<td>16</td>
</tr>
<tr>
<td>MS4</td>
<td>0.47</td>
<td>0.72</td>
<td>3</td>
</tr>
</tbody>
</table>

The results of table 5.6 show that the cohesion is at an acceptable level, and the granularity of the microservices is good as well because the LCOM values are between 0.3 and 0.6. According to chapter 3 results, LCOM values should be between 0.2 and 0.8 if the NOO value were less than 10. Although, the results of MS3 are over the suggested optimal value.

## 5.5 Conclusion

This research proposed a novel decomposition method for refactoring monolithic applications into microservices applications using a neural network based model for creating code embeddings from the monolithic application source code. As a result, semantically similar code embeddings are grouped using a hierarchical clustering algorithm to generate microservices candidates. The quality characteristics of the results were measured using two metrics for measuring cohesion.

The proposed method showed promising results in terms of cohesion when compared to other decomposition methods. The results were compared with two other
methods proposed by Jin et al. [24], and Saidani et al. [23], 8 test cases were conducted, and the proposed method got the highest scores in 4 of them.

In conclusion, the proposed method can be a helpful add-on for developers in the process of migration from a monolithic architecture into a microservices architecture. This method will give the developers insights and directions on the path and the design that the developers need to take to achieve a good microservices design.

Publications related to this thesis: [SCPE21]
Chapter 6

Thesis IV: Dependencies Based Microservices Decomposition Method
6.1 Introduction

This research tackles the challenge of microservice decomposition through analyzing class dependencies by converting them into a weighted graph and clustering the nodes using a clustering algorithm. This research investigates which clustering algorithm is the most suitable for this method by comparing the algorithms’ performance using F-Measure, Newman Girvan Modularity, and the number of microservices. The method was evaluated using 8 different applications with different sizes ranging from small to medium. This research aims to create an accurate microservices decomposition method by analyzing the monolithic application source code using class dependencies to reflect the structure and links between the classes of the application.

6.2 Methodology

This research aims to construct a meaningful representation of the source code structure by extracting class dependency from the source code of the monolithic application. Then represent these dependencies as a network of connected nodes and edges. The next step is to apply a clustering algorithm to cluster classes with strong dependencies to generate microservices candidates. In this section, I present the process of constructing the class dependency graph and display the performance of multiple clustering algorithms to identify suitable microservices by analyzing the class dependency graph. The overall decomposition method is represented in Figure 6.1. In Figure 6.1, the process starts with the source code of the monolithic application, then extracting class dependencies by analyzing the source code. The next step is creating a dependency graph between the classes to represent the relationships between them. Furthermore, a clustering algorithm is used to identify microservices candidates via grouping classes with high dependencies between them.

6.2.1 Constructing the Class Dependency Graph

The class dependency graph can be constructed by analyzing the source code of the monolithic application. This graph represents the relationship between the classes of the targeted application. The classes are represented as nodes, and the edges between these nodes represent the relationships between these classes. If a class
accesses another class via a method or a variable, an edge will occur between them. The weights of the edges are represented as dependencies or references. The process of constructing the dependency graph is represented in Algorithm 3 where the input is the source code of the monolithic application, and the output is a weighted graph $G$ represents the relationships between the classes of the application.

**Algorithm 3:** Construction of Class Dependency Graph

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$classes \leftarrow extractClasses(S)$</td>
</tr>
<tr>
<td>2</td>
<td>$foreach$ class of classes $do$</td>
</tr>
<tr>
<td>3</td>
<td>$methods \leftarrow extractMethods(class)$</td>
</tr>
<tr>
<td>4</td>
<td>$G \leftarrow addVertices(methods)$</td>
</tr>
<tr>
<td>5</td>
<td>$end$</td>
</tr>
<tr>
<td>6</td>
<td>$foreach$ method $\in G$ $do$</td>
</tr>
<tr>
<td>7</td>
<td>$references \leftarrow findReferences(method)$</td>
</tr>
<tr>
<td>8</td>
<td>$foreach$ reference of references $do$</td>
</tr>
<tr>
<td>9</td>
<td>$if$ reference $\in G$ $then$</td>
</tr>
<tr>
<td>10</td>
<td>$G \leftarrow addEdge(method, reference)$</td>
</tr>
<tr>
<td>11</td>
<td>$end$</td>
</tr>
<tr>
<td>12</td>
<td>$end$</td>
</tr>
<tr>
<td>13</td>
<td>$end$</td>
</tr>
</tbody>
</table>

6.2.2 Identifying Microservices

The next part of the proposed decomposition algorithm is to identify microservices through using a graph clustering algorithm. In this chapter, several graph clustering algorithms were tested, and their performance was compared to find the most suitable one for the proposed method.

For Algorithm 4, a weighted graph and an empty set represent the microservices are the inputs. The expected output is microservice candidates. The lines from 1 to 4 represent looping through the graph nodes and edges, creating an empty set for the microservices candidates. Furthermore, the lines between 6 and 11 represent the clustering algorithm which is the Leiden algorithm. Leiden algorithm is the most suitable algorithm for the proposed method because of the promising results in the tests. The results are shown in the results and analysis section 6.3.

The decomposition algorithm’s final output is represented as a clustered graph, with each cluster represented by a unique color. For example, Figure 6.2 shows the SpringBlog application’s decomposition using the proposed method. The nodes represent the classes of the application, and the colored nodes represent the microservices candidates. This concludes that SpringBlog decomposed into 4 microservices because there are four different colors in the generated graph. The original decomposition of this application is also consisting of 4 microservices but with a different
Algorithm 4: Microservices Identification

**Data:** $G = (V, E, w), MS = \emptyset$

**Result:** MS candidates as clusters of classes $C_1, C_2, C_3, \ldots C_{MS}$

1. for $u \in V$ do
   2. $u \leftarrow 0$
   3. end
4. $MS \leftarrow \emptyset$
5. while $u \notin MS$ do
   6. $MS \leftarrow \text{MoveNodesFast}(G, MS)$
   7. $done \leftarrow |P| = |V(G)|$
   8. if not $done$ then
   9. $MS_{\text{refined}} \leftarrow \text{RefinePartition}(G, MS)$
10. $G \leftarrow \text{AggregateGraph}(G, MS_{\text{refined}})$
11. $MS \leftarrow \{\{v|v \subseteq C, v \in V(G)\}|C \in MS\}$
12. end
13. end
14. return $MS$

class organization. The proposed decomposition method had a 0.98 F1 value for this decomposed application, which is a good score because it is remarkably close to the original decomposition.

![Image](image_url)

Figure 6.2: SpringBlog’s Decomposition Result

The decision of choosing a suitable graph clustering algorithm is decided based on the outcomes of the results and analysis section 6.3.

### 6.2.3 Evaluation Metrics

For the evaluation section of the research, I utilized three metrics to compare and check the results’ quality. F-Measure (F1) metric is used to compare the resulted microservices with the reference decomposition. Also, the Newman Girvan Modularity (NGM) metric measures the overall quality of the produced clusters. Finally, the number of microservices will be compared to the reference number of the microservices in the original decomposition.
6.2. METHODOLOGY

F-Measure

Rossetti et al. [83] proposed a novel approach to calculate the harmonic mean between the two communities produced by community identifying algorithms. In a network, an averaged F1 score of the identified pairs can encapsulate the overall similarity between the community set produced by the clustering algorithm and the reference community set.

Newman Girvan Modularity

Modularity metric is used to check the overall quality of the communities or the clusters. Communities with high modularity have a strong connection between their nodes. Newman Girvan Modularity [15] is a good measure for the overall quality of the clusters and tries to explore the community structure of the entire network. This research uses this metric because it is one of the most utilized functions to quantify community detection methods [84]. The bases for this metric is that a random graph cannot have a basic construction of a cluster, so the possible existence of clusters is discovered by the comparison between the definite density of vertices in a community and the expected density to have in the community if the nodes of the network were attached without taking into consideration the structure of the community. Newman Girvan Modularity can be calculated as follows:

\[
Q(S) = \frac{1}{m} \sum_{c \in S} (m_s - \frac{(2m_S + l_S)^2}{4m}) \tag{6.1}
\]

where \(m\) represent the number of network edges, \(m_s\) represent the number of cluster edges, \(l_S\) is the number of edges from vertices in \(S\) to vertices out of \(S\). Finally, If the community edges were structured randomly, we will have \(Q = 0\). If \(Q = 1\), it is indicating a strong community structure.

Microservices Numbers

This metric represents the total number of microservices that an application is made up of. This metric needs to be compared with the optimal number of microservices that an application should have. The definition of this metric is the summation of the number of microservices:

\[
MS = \sum_{n=1}^{m} m \tag{6.2}
\]

where \(m\) represent a microservice in the application.

6.2.4 Clustering Algorithms

Clustering is an essential part of the proposed decomposition procedure in this research. Therefore, several graph clustering algorithms were tested to find the most suitable one for the proposed decomposition method. Most of these clustering algorithms were utilized by other researchers in several refactoring algorithms [31–34].
In this research several clustering methods were tested to find the most optimal clustering method for the proposed decomposition algorithm. These algorithms are Label propagation (LP) algorithm [16], Louvain algorithm (LV) [85], Leiden algorithm (LA) [86], Speaker-Listener Label Propagation Algorithm (SLPA) [87], Leading Eigenvector Algorithm (LE) [88], Girvan–Newman algorithm (GN) [89], Markov clustering algorithm (MC) [90], Rber pots Algorithm (RP) [91], Rb pots Algorithm (RB) [91], Walktrap Algorithm (WT) [92], and Chinesewhispers Algorithm (CW) [93]. These algorithms were implemented using a Python library for community discovery algorithms called CDlib [94]. This library consists of 39 graph clustering algorithms, making it one of the largest libraries available [94].

6.2.5 Test Case Applications

For evaluating the proposed algorithm and testing the performance of different clustering algorithms for the proposed microservices decomposition algorithm. Eight different applications were tested. These applications range from small to medium-sized applications, and all these applications are written using the JAVA programming language.

Acme Air

Acme Air\(^1\) application is a monolithic JAVA application used as a sample airline provide functionalities such as booking and searching flights. This application has a microservices design that can be compared with the proposed decomposition.

SpringBlog

SpringBlog\(^2\) application is a microservices application created using Spring Boot Framework. It is a simple blog website.

Sample Cargo Application

Cargo Application\(^3\) is a monolithic JAVA application. It is a sample application for managing cargos and tracking them. This application was tested in multiple research papers such as Gysel’s et al. [14] Service Cutter paper.

FTGO Application

FTGO application\(^4\) is a sample monolithic restaurant application written in JAVA. This application is from Chris Richardson’s book [75] for designing microservices applications. This application is used as a reference for refactoring monolithic applications to microservices.

---

\(^1\)https://github.com/blueperf/acmeair-monolithic-java
\(^2\)https://github.com/Raysmond/SpringBlog
\(^3\)https://github.com/citerus/dddsample-core
\(^4\)https://github.com/microservices-patterns/ftgo-monolith
6.3. RESULTS AND ANALYSIS

**Everest Application**

Everest\(^5\) is a reference application for refactoring monolithic applications into microservices applications. This application was tested in multiple research papers [24,25].

**JPetStore Application**

JPetStore\(^6\) application is a reference monolithic application created using JAVA programming language. This application has its microservices counterpart, which can be used for comparing the results of this research. This application was used in multiple research papers [24,25].

**Parts Unlimited Application**

Parts Unlimited\(^7\) is a monolithic application developed by Microsoft to showcase the decomposition process of monolithic applications into microservices architecture.

### 6.3 Results and Analysis

The proposed algorithm’s performance was tested with 8 different monolithic applications discussed in section 6.2.5. Additionally, 11 clustering algorithms were tested against each other to find the most suitable clustering algorithm for the proposed decomposition method. The tested algorithms were mentioned in section 6.2.4. Also, three evaluation metrics were used to evaluate the proposed algorithm’s performance and compare the clustering algorithms as well.

Figure 6.3 presents how precise and effective the proposed decomposition algorithm is utilizing different clustering algorithms. F1 shows the harmonic mean of precision and recall. In other words, it is an alternative view of the accuracy of the decomposition method. LA algorithm had the best results compared to the other algorithms, with 0.8 averaged F1 score. MC had the worst performance compared to the other algorithms.

Figure 6.4 shows how well structured the clustering output of the clustering algorithms using the NGM metric. The highest score for this metric was 0.65 across multiple algorithms for the AcmeAir application. This indicates that the classes have strong and obvious relationships among them. Similar to F1 results, MC algorithm had the worst results compared to the other algorithms. LV, CW, and LA scored the highest averaged scores listed respectively as 0.5, 0.46, 0.44.

Figure 6.5 displays the averaged results of F1 and NGM metrics. These results are used to compare the proposed decomposition method’s performance utilizing different clustering algorithms to choose the most optimal one. These results show that LA has the highest F1 score with 0.8 but the third-highest NGM score of 0.44. This

---

\(^5\)https://github.com/arun-gupta/microservices
\(^6\)https://github.com/mybatis/jpetstore-6
\(^7\)https://github.com/microsoft/PartsUnlimitedMRP
Figure 6.3: F1 Results Comparing Clustering Algorithms Results of Different Test Case Application

Figure 6.4: NGM Results Comparing Clustering Algorithms Results of Different Test Case Application
6.3. RESULTS AND ANALYSIS

![Bar chart showing averaged results of F1 and NGM comparing the performance of different clustering algorithms](image)

Figure 6.5: Averaged Results of F1 and NGM Comparing the Performance of Different Clustering Algorithms

means that the LA algorithm is the most accurate compared to the other algorithms but does not produce the most well-structured microservice candidates. Although comparing LA to LV, which has the highest NGM score, LV has a 0.74 F1 score, which means that LA is more accurate, which is the most important feature for the proposed decomposition algorithm. Therefore, the LA clustering algorithm is the most optimal clustering algorithm for this research’s proposed decomposition algorithm. LA clustering algorithm have a weights parameter which I tested it with different values for this parameter but it did not have a noticeable performance change to the accuracy. Therefore, the default parameters for LA algorithm is used for the proposed method.

Table 6.1 demonstrates the averaged results of the different tested clustering algorithms in more detail, which is an extension of the results of Figure 6.5. In this table, the RB algorithm has remarkably similar results to the LA algorithm, but LA is a little more accurate because it has a 0.8 F1 score compare to a 0.79 score. NGM scores range between 0.16 to 0.5, even though most of these scores range between 0.3 and 0.5, excluding GN and MC, which have 0.28 and 0.16, respectively.
Table 6.1: Averaged Results of F1 and NGM Comparing the Performance of Different Clustering Algorithms

<table>
<thead>
<tr>
<th></th>
<th>F1</th>
<th>NGM</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA</td>
<td>0.8</td>
<td>0.44</td>
<td>4.13</td>
</tr>
<tr>
<td>RB</td>
<td>0.79</td>
<td>0.44</td>
<td>4.25</td>
</tr>
<tr>
<td>LE</td>
<td>0.78</td>
<td>0.4</td>
<td>4</td>
</tr>
<tr>
<td>LV</td>
<td>0.74</td>
<td>0.5</td>
<td>4.88</td>
</tr>
<tr>
<td>CW</td>
<td>0.73</td>
<td>0.46</td>
<td>4.75</td>
</tr>
<tr>
<td>LP</td>
<td>0.71</td>
<td>0.3</td>
<td>3.75</td>
</tr>
<tr>
<td>WT</td>
<td>0.7</td>
<td>0.39</td>
<td>5.38</td>
</tr>
<tr>
<td>GN</td>
<td>0.69</td>
<td>0.28</td>
<td>4.5</td>
</tr>
<tr>
<td>RP</td>
<td>0.67</td>
<td>0.39</td>
<td>5.63</td>
</tr>
<tr>
<td>SLPA</td>
<td>0.63</td>
<td>0.32</td>
<td>6.63</td>
</tr>
<tr>
<td>MC</td>
<td>0.55</td>
<td>0.16</td>
<td>7.75</td>
</tr>
</tbody>
</table>

Statistical analysis was made for the proposed decomposition method results in Table 6.2. These results show that the least accurate result for the decomposition method scored 0.56 for the F1 metric. The lowest accurate score was for the cargo application. Also, the highest F1 score was 0.98, which is for the SpringBlog application. These scores show that the proposed decomposition algorithm has promising accuracy results range between medium and high scores. As for the NGM metric, the results were between 0.23 and 0.57, with an average of 0.44. This means that the decomposition results were not random and the generated microservices are well structured. The low standard error indicates that the mean is an accurate indication for the NGM and F1 scores. MS results indicate that the test case applications were consisting of 3 to 6 microservices per application. These applications are small to medium-sized applications, which can be considered the limitation of this study.

Table 6.2: Statistical Analysis of the Proposed Decomposition Method Results

<table>
<thead>
<tr>
<th></th>
<th>F1</th>
<th>NGM</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>0.8</td>
<td>0.44</td>
<td>4.12</td>
</tr>
<tr>
<td>Standard Error</td>
<td>0.05</td>
<td>0.04</td>
<td>0.35</td>
</tr>
<tr>
<td>Median</td>
<td>0.83</td>
<td>0.47</td>
<td>4</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>0.16</td>
<td>0.11</td>
<td>0.99</td>
</tr>
<tr>
<td>Sample Variance</td>
<td>0.02</td>
<td>0.01</td>
<td>0.98</td>
</tr>
<tr>
<td>Minimum</td>
<td>0.56</td>
<td>0.23</td>
<td>3</td>
</tr>
<tr>
<td>Maximum</td>
<td>0.98</td>
<td>0.57</td>
<td>6</td>
</tr>
</tbody>
</table>

The performance of the proposed decomposition application is comparable and better than several methods in the literature. Table 6.3 compares the average F1 value of several methods compared to the proposed decomposition method. The proposed decomposition method has similar performance compared to Selmadji et al. [28] and Baresi et al. [17] methods. The proposed method was tested with 8 applications,
while the other methods were tested with 3 applications. This point indicates that the proposed method was tested with more cases than the other methods. Hence the proposed method has a more accurate score when compared with the other approaches. The proposed method performed better when compared to Nunes et al. [26] method because the proposed method scored 0.8 F1 while Nunes et al. method scored 0.58. These results show that the proposed method has better performance when compared with methods in the literature.

<table>
<thead>
<tr>
<th>Method</th>
<th>Averaged F1</th>
<th># of tested applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Proposed Method</td>
<td><strong>0.80</strong></td>
<td>8</td>
</tr>
<tr>
<td>Nunes et al. [26]</td>
<td>0.58</td>
<td>3</td>
</tr>
<tr>
<td>Selmadji et al. [28]</td>
<td>0.81</td>
<td>3</td>
</tr>
<tr>
<td>Baresi et al. [17]</td>
<td>0.80</td>
<td>3</td>
</tr>
</tbody>
</table>

### 6.4 Evaluation Using Chapter 3 Metrics

In this section, the AcmeAir application was used to demonstrate the effectiveness of chapter 3 metrics to detect good microservices decomposition. Table 6.4 shows promising results for the LCOM metric, but the SGM metric shows high values. These high values can be attributed to the small complexity of the application.

<table>
<thead>
<tr>
<th></th>
<th>LCOM</th>
<th>SGM</th>
<th>NOO</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS1</td>
<td>0.13</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MS2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MS3</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MS4</td>
<td>0.58</td>
<td>0.74</td>
<td>3</td>
</tr>
</tbody>
</table>

FTGO application decomposition was tested with the metrics of Chapter 3 to check the effectiveness of the proposed metrics, the results of the metrics are in the acceptable range that was proposed in chapter 3. The results are in Table 6.5.

<table>
<thead>
<tr>
<th></th>
<th>LCOM</th>
<th>SGM</th>
<th>NOO</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MS2</td>
<td>0.5</td>
<td>0.8</td>
<td>2</td>
</tr>
<tr>
<td>MS3</td>
<td>0.67</td>
<td>0.45</td>
<td>3</td>
</tr>
<tr>
<td>MS4</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
6.5 Conclusion

This research proposed a microservices identification method using class dependencies represented as a graph. This method consists of two main parts. The first part is representing the source code of the monolithic application as a class dependency graph. This graph represents the structure of the monolithic application and the relationships between the classes of the application. The second part of the method is a graph clustering algorithm to identify microservices by analyzing the dependencies between the monolithic application classes and cluster classes with strong relationships to produce microservice candidates.

The method was tested with 8 different applications; these applications were used in other research papers. Also, 11 clustering algorithms were tested to find the most accurate and efficient one suitable for the proposed algorithm. The proposed method had promising results compared to other literature methods with a 0.8 averaged F1 score and a 0.44 averaged NGM score. The F1 score shows that the proposed method has good accuracy in detecting the needed microservices when compared against a reference application. The NGM score shows that the generated microservices candidates are not random and that there are well-defined relationships among the clustered classes of a microservice.

This method can provide helpful insight for developers dealing with the migration process of monolithic applications to microservices architecture. This method requires only the source code of the monolithic application to function, which means it is straightforward to implement and gives meaningful results for the developer.

Publications related to this thesis: [IJCA21]
Chapter 7

Summary of the Results
Three microservices identification methods were introduced in this dissertation and a set of metrics to analyze the quality design of the microservices based on API analysis. Referring to Figure 1.1 the methods can be used based the availability of different software artifacts such as API definitions, source code, and class dependency diagrams.

The method of Thesis II starts with an API as an input and then extracts the operation names from the API definition. Then, the operation names are converted into vector representation using fastText algorithm and these vectors are grouped together using Affinity Propagation algorithm to generate microservices candidates. This method was extended by applying the metrics of Thesis I to analyze the produced microservices and check if their cohesion, complexity, and granularity is within an acceptable range.

In Thesis III, the decomposition method is based on analyzing the source code of the monolithic application and converting it into vector representation using code2vec algorithm. Then Affinity Propagation is applied on the vectors in order to produce microservices candidates through clustering semantically similar classes together.

The algorithm of Thesis IV is based on having a dependency graph of classes that represent the monolithic application and how the classes are related to each other. Then a graph clustering algorithm is applied on the classes dependency graph to group classes with strong relationships with each other to produce microservices candidates. The clustering algorithm that was used in this method was Leiden graph clustering algorithm.
Chapter 8

Future Work and Methods

Applicability
The algorithms described in this dissertation can be a helping tool for developers facing the challenge of migrating from monolithic architectures to microservices. These algorithms can show them possible scenarios based on different circumstances and help them decide the most optimal decomposition arrangement. For example, if a company wants to refactor an old monolithic application into a microservices application. They can use one or a combination of the proposed decomposition algorithms based on their objective or scenarios. One scenario can be that they want to achieve a more scalable and maintainable application, and they want to analyze the dependencies between the classes. So, in that case, they would go with the algorithm of Thesis II if the APIs are accessible or the algorithm of Thesis IV if the APIs are not available, or they can check the results of both algorithms if applicable.

For future work, the Chapter 5 algorithm can be developed further. It can be tested with other programming languages such as Python, C, C++, et al. The tested cases of this research were all written in JAVA. The proposed method is only capable of handling code written in that programming language. Also, the neural network-based model can be trained on the source codes of the microservices application to achieve more precise results.

The proposed algorithm of Chapter 6 can be expanded to include large applications to be tested against the proposed decomposition method for future work. In the chapter, several applications were tested, but their sizes were ranged from small to medium applications.

Regarding the the proposed metrics in Chapter 3, these metrics can be expanded to include a new metric that can measure the coupling between different microservices and their APIs.
Bibliography


[26] Luí­s Nunes, Nuno Santos, and António Rito Silva. From a monolith to a microservices architecture: An approach based on transactional contexts. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 11681 LNCS, pages 37–52. Springer Verlag, 2019.


Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn:


