Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Electronics Technology

Microservices Identification Methods and Quality Metrics

PhD Dissertation Booklet

Omar Al-Debagy

Scientific Supervisor
Dr. Péter Martinek

2021

Overview

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, besides it can lead to longer time of deploying new features [1]. Therefore, many companies 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) [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 [3]. Martin Fowler defined Microservices Architecture as an approach to develope 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 own process [4].

Microservices need a simpler routing mechanism without the need for global governance when compared to SOA [5]. 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 [6].
- 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 [6].
- 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 [6].
- 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 [6].

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. These metrics were designed to measure the cohesion, complexity, and granularity of the microservices application.
- 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 and offer different methods, 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. The previously mentioned advantages of microservices are maintainability, reusability, scalability, availability, and automated deployment [3]. 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 [7].

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 and design metrics to assess the decomposition of microservices applications.

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:

- **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 [8]. Also, the small size leads to reducing the number of bugs in the application [4].
- **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 shows the flow of how to choose an appropriate decomposition method.

Firstly, developers need to set their objectives of the decomposition process. 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 [9]. 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.

Based on the diagram shown in Figure 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.

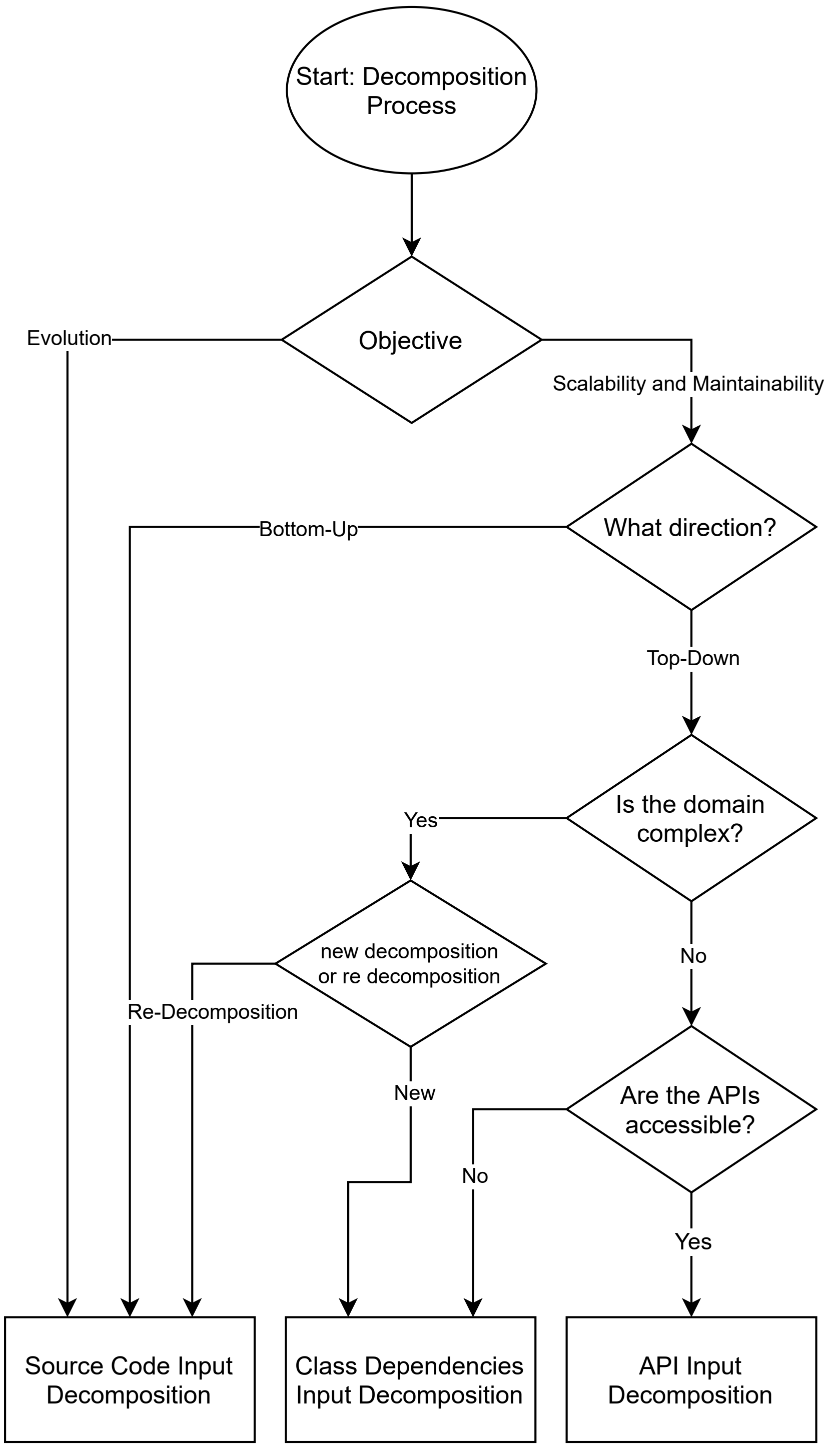
Start: Decomposition Process
Objective
Evolution
Scalability and Maintainability
What direction?
Bottom-Up
Top-Down
Is the domain complex?
Yes
No
new decomposition or re decomposition
Re-Decomposition
New
Are the APIs accessible?
No
Yes
Source Code Input Decomposition
Class Dependencies Input Decomposition
API Input Decomposition

Figure 1: Decision Guide for Decomposition Methods

The New Scientific Results Formulated into Theses

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.

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). These 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 calculated on five applications with different sizes and business cases. I have defined limits for the SGM metric 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.

The Lack of Cohesion Metric "LCOM"

LCOM measures the cohesiveness or, in other words, the similarity between the operations in specific services 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 designed 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 1.

$$LCOM = 1 - \frac{\sum_{i=1}^{n} OP_i}{M * F} \tag{1}$$

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.

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 [10] with some modifications to be suitable for microservices architecture design.

Fine-grained and coarse-grained parameters define the whole idea of the DGS metric. DGS is defined as follows:

$$DGS = \frac{IPR}{\sum_{i=1}^{n} FP_i} + \frac{OPR}{\sum_{i=1}^{n} CP_i} \tag{2}$$

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 is close to 1 it indicates coarse-grained data in the microservice. While the value of DGS is close to 0 indicates fine-granular data.

In order to measure the functional granularity of each operation in a microservice, the FGS metric is defined as follows:

$$FGS = \frac{OT}{\sum_{i=1}^{n} O_i} \tag{3}$$

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. 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.

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 4.

$$SGM = \sum_{i=1}^{n} DGS_i * FGS_i \tag{4}$$

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. 5). Similar to the WMC metric [11], 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.

$$NOO = \sum_{i=1}^{n} M_i \tag{5}$$

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

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 scores to compare the performance of the clustering parameters. 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.

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 embeddings using the fastText model. fastText [12] and Word2Vec [13] 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 [14]. Algorithm 1 presents a general overview of the proposed decomposition method.

In this research, Word2Vec [14] was trained on Google News corpus, and fastText was trained on Wikipedia 2017, UMBC webbase corpus, and statmt.org news dataset. Nevertheless, before converting the operation names into vectors, removing the stop words was an initial step because keeping stop words 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.

Accordingly, in total, there were 4 applications with 452 operations tested using the proposed decomposition method. The performance of the algorithm was measured using precision, recall, and F-measure metrics. The averaged F-Measure was 81 % while the averaged precision of all the tests was 84 % and the averaged recall was 78

Algorithm 1: The proposed decomposition algorithm

```
Data: OpenAPI Specifications
Result: microservices' candidates
1 sentences←∅
2 foreach operationName do
3     sentences←ConvertToLowerCase(operationName);
4     sentences←RemoveStopWords(sentences);
5     sentences←ShorttextToVector(operationName);
6 end
7 microserviceCandidates←AffinityPropagation(sentences);
8 return microserviceCandidates
```

%. 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 1 shows the performance of the algorithm when decomposing different applications.

Table 1: The performance of the proposed decomposition methodology

Application	Precision	Recall	F-Measure	# of Operations
AWS	0.74	0.79	0.76	318
Kanban Board	1	0.85	0.92	13
Money Transfer	0.82	0.82	0.82	11
PayPal	0.8	0.66	0.72	110
	Precision Average	**Recall Average**	**F-Measure Average**	**Total**
	0.84	0.78	0.81	452

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 generated better results in 5 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 from the literature.

Methodology

Machine learning for code refactoring was used on several other software architectures before [15–17]. 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 [18].

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.

Code2vec [19] 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. Figure 2 shows the architecture of code2vec model.

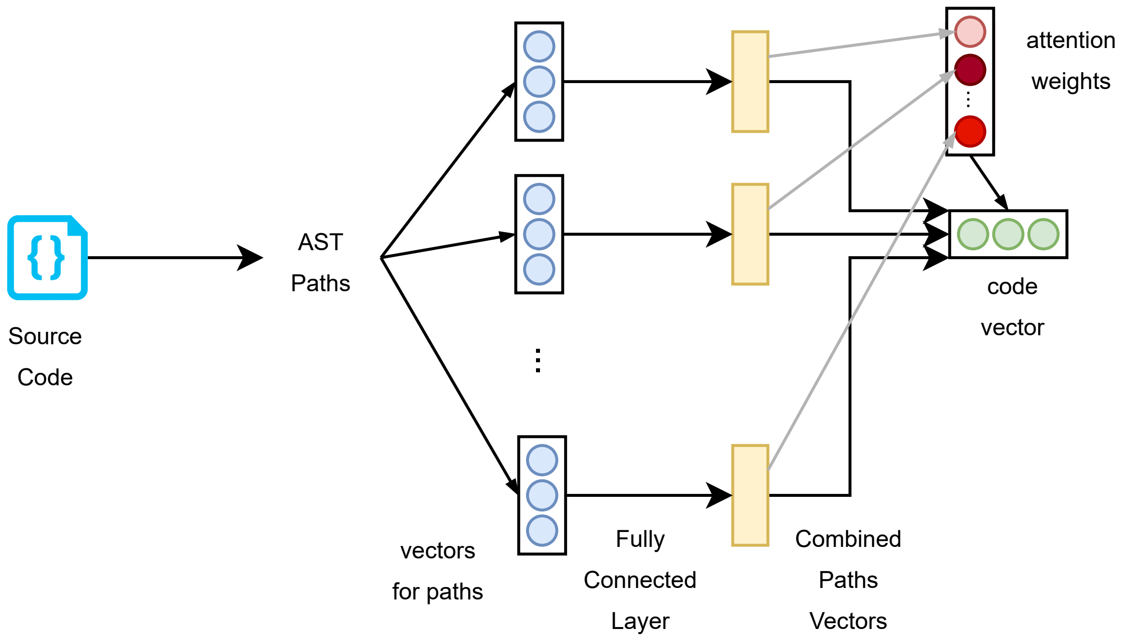

Figure 2: code2vec Model [19]

Figure 3 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.

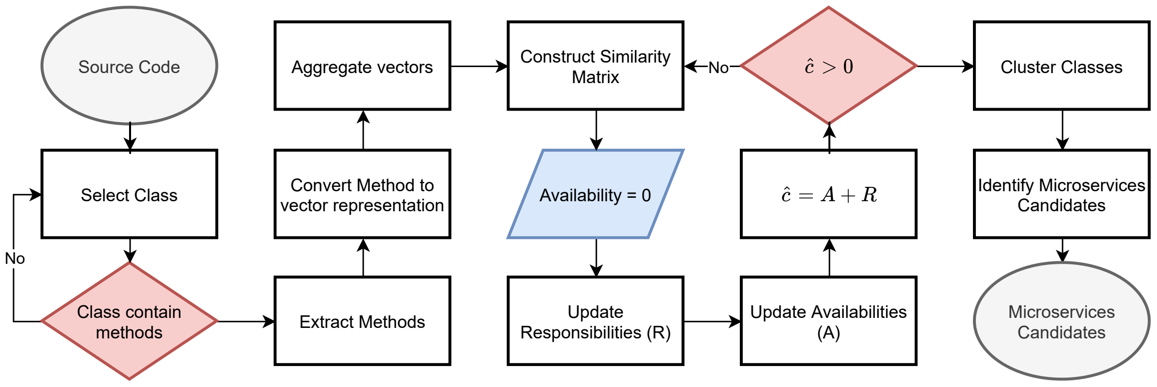

Figure 3: High - Level Representation of the Proposed Algorithm

The overall results for tested applications suggest that the proposed approach has some advantages in cohesion in the middle-sized and big-sized applications. For example, Table 2 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,

Table 2: Decomposition Results

Application	Metrics	Jin et al	Saidani et al.	My Method
JPetStore	CHD	0.52	**0.65**	0.52
	CHM	0.78	0.55	**0.82**
SpringBlog	CHD	0.55	**0.67**	0.50
	CHM	0.68	**0.75**	0.73
JForum	CHD	0.45	0.15	**0.52**
	CHM	0.70	0.51	**0.73**
Roller	CHD	0.52	0.38	**0.53**
	CHM	0.72	**0.78**	0.76

while Saidani et al.'s [20] method scored 4 out of 8, and Jin et al. [21] 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 [6].

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.

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. The overall decomposition method is represented in Figure 4. In Figure 4, 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.

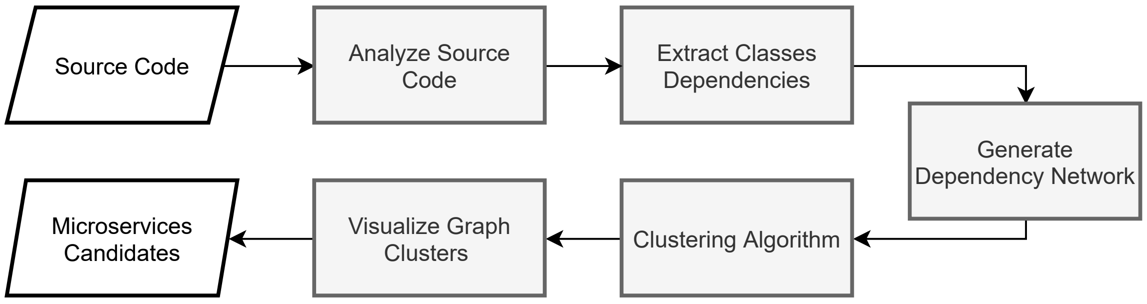

Figure 4: An overall View of the Proposed Decomposition Method

For Algorithm 2, 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.

Algorithm 2: Microservices Identification

```
   Data: $G = (V, E, w), MS = \emptyset$
   Result: MS candidates as clusters of classes $C_1, C_2, C_3 \dots C_{MS}$
1  for $u \in V$ do
2      $u \leftarrow 0$
3  end
4  $MS \leftarrow \emptyset$
5  while $u \notin MS$ do
6      $MS \leftarrow MoveNodesFast(G, MS)$
7      $done \leftarrow |P| = |V(G)|$
8      if $not\ done$ then
9          $MS_{refined} \leftarrow RefinePartition(G, MS)$
10         $G \leftarrow AggregateGraph(G, MS_{refined})$
11         $MS \leftarrow \{\{v|v \subseteq C, v \in V(G)\}|C \in MS\}$
12     end
13 end
14 return $MS$
```

The performance of the proposed decomposition application is comparable and better than several methods in the literature. Table 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. [22] and Baresi et al. [23] 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. [24] method because the proposed method scored 0.8 F1 while Nunes et al. method scored 0.58.

Table 3: Comparison with Other Decomposition Methods from the Literature

Method	**Averaged F1**	**# of tested applications**
The Proposed Method	**0.80**	**8**
Nunes et al. [24]	0.58	3
Selmadji et al. [22]	0.81	3
Baresi et al. [23]	0.80	3

Publications related to this thesis: [IJCA21]

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 algorithm presented in Chapter 5 could be developed further in the future. 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.

Publications

Journal articles related to the theses

[PP19] Omar Al-Debagy and Péter Martinek. A new decomposition method for designing microservices. *Periodica Polytechnica Electrical Engineering and Computer Science*, 63(4):274–281, 2019.

[JWE20] Omar Al-Debagy and Péter Martinek. A metrics framework for evaluating microservices architecture designs. *J. Web Eng.*, 19:341–370, 2020. (IF: 0.39)

[SCPE21] Omar Al-Debagy and Péter Martinek. A Microservice Decomposition Method Through Using Distributed Representation of Source Code. *Scalable Computing: Practice and Experience*, 22(1):39–52, 2021.

[IJCA21] Omar Al-Debagy and Péter Martinek. Dependencies Based Microservices Decomposition Method. *International Journal of Computers and Applications*, 1-8, 2021.

Proceedings articles related to the theses

[CINTI18] Omar Al-Debagy and Péter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium* *on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154, 2018.

[SOSE20] Omar Al-Debagy and Péter Martinek. Extracting microservices' candidates from monolithic applications: Interface analysis and evaluation metrics approach. In *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*, pages 289–294, 2020.

References

[1] Davide Taibi and Kari Systä. From monolithic systems to microservices: A decomposition framework based on process mining. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science*, pages 153–164. SCITEPRESS - Science and Technology Publications, 2019.

[2] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[3] R. Chen, S. Li, and Z. Li. From monolith to microservices: A dataflow-driven approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475, 2017.

[4] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[5] Tomas Cerny, Michael J. Donahoo, and Jiri Pechanec. Disambiguation and comparison of soa, microservices and self-contained systems. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, RACS '17, page 228–235, New York, NY, USA, 2017. Association for Computing Machinery.

[6] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 1st edition, 2015.

[7] D. Taibi, V. Lenarduzzi, and C. Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[8] Bill Thomas and Scott Tilley. Documentation for software engineers: What is needed to aid system understanding? In *Proceedings of the 19th Annual International Conference on Computer Documentation*, SIGDOC '01, page 235–236, New York, NY, USA, 2001. Association for Computing Machinery.

[9] Anfel Selmadji. *From monolithic architectural style to microservice one : structure-based and task-based approaches*. Theses, Université Montpellier, October 2019.

[10] Saad Alahmari, Ed Zaluska, and David De Roure. A metrics framework for evaluating soa service granularity. In Hans-Arno Jacobsen, Yang Wang, and Patrick Hung, editors, *IEEE SCC*, pages 512–519. IEEE Computer Society, 2011.

[11] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah. Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. *IEEE Transactions on Software Engineering*, 44(6):534–550, 2018.

[12] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 427–431, Valencia, Spain, April 2017. Association for Computational Linguistics.

[13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26, pages 3111–3119. Curran Associates, Inc., 2013.

[14] Chenglong Ma, Weiqun Xu, Peijia Li, and Yonghong Yan. Distributional representations of words for short text classification. In *Proceedings of the 1st Workshop on Vector Space Modeling for Natural Language Processing*, pages 33–38, Denver, Colorado, June 2015. Association for Computational Linguistics.

[15] Brahmaleen Kaur Sidhu, Kawaljeet Singh, and Neeraj Sharma. A machine learning approach to software model refactoring. *International Journal of Computers and Applications*, 0(0):1–12, 2020.

[16] Boukhdhir Amal, Marouane Kessentini, Slim Bechikh, Josselin Dea, and Lamjed Ben Said. On the use of machine learning and search-based software engineering for ill-defined fitness function: A case study on software refactoring. In Claire Le Goues and Shin Yoo, editors, *Search-Based Software Engineering*, Lecture Notes in Computer Science, pages 31–45. Springer International Publishing, 2014.

[17] Yasemin Kosker, Burak Turhan, and Ayse Bener. An expert system for determining candidate software classes for refactoring. *Expert Systems with Applications*, 36(6):10000–10003, 2009.

[18] T.L. McCluskey and J.M. Porteous. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence*, 95(1):1 – 65, 1997.

[19] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[20] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Aymen Saied. Towards automated microservices extraction using muti-objective evolutionary search. In Sami Yangui, Ismael Bouassida Rodriguez, Khalil Drira, and Zahir Tari, editors, *Service-Oriented Computing*, volume 11895, pages 58–63. Springer International Publishing, 2019.

[21] Wuxia Jin, Ting Liu, Qinghua Zheng, Di Cui, and Yuanfang Cai. Functionality-oriented microservice extraction based on execution trace clustering. In *2018 IEEE International Conference on Web Services (ICWS)*, pages 211–218, 2018.

[22] Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, Christophe Dony, and Rahina Oumarou Mahamane. Re-architecting OO Software into Microservices. In Kyriakos Kritikos, Pierluigi Plebani, and Flavio de Paoli, editors, *Service-Oriented and Cloud Computing*, pages 65–73, Cham, 2018. Springer International Publishing.

[23] Luciano Baresi, Martin Garriga, and Alan De Renzis. Microservices identification through interface analysis. In Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen, editors, *Service-Oriented and Cloud Computing*, pages 19–33, Cham, 2017. Springer International Publishing.

[24] 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.