Université de Montréal

Microservices Identification in Existing Applications Using Meta-heuristics Optimization and Machine Learning

par

Hanifa BARRY

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

Orientation Intelligence Artificielle

March 22, 2024

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Microservices Identification in Existing Applications Using Meta-heuristics Optimization and Machine Learning

présenté par

Hanifa BARRY

a été évalué par un jury composé des personnes suivantes :

Résumé

L'architecture en microservices met en évidence de multiples avantages pour les entreprises et les développeurs. Cela explique pourquoi de nombreuses entreprises technologiques choisissent de migrer leurs logiciels d'une architecture monolithique vers celle des microservices. Cependant, la migration d'un système monolithique ou d'un système hérité vers une architecture en microservices est une tâche complexe, risquée et chronophage. Pour faciliter et améliorer ce processus, notre travail se concentrera sur la conception d'une approche semi-automatique pour détecter et identifier les microservices dans les applications existantes. Il s'agit d'une étape clé vers l'objectif global de migrer un système monolithique vers des microservices. Notre approche consiste à combiner des méthodes de regroupement et d'optimisation de la proximité entre les dépendances structurelles à un niveau de granularité optimal. Dans ce projet, nous nous appuyons également sur plusieurs méthodes d'intelligence artificielle, en particulier des algorithmes d'apprentissage automatique, pour mettre en œuvre notre approche. D'une part, nous effectuons l'extraction de dépendances et le regroupement. D'autre part, nous mettons en œuvre des méthodes qui nous aideront à optimiser la proximité entre éléments constituant un microservice. Pour obtenir les scores de proximité, nous ciblons à la fois les relations sémantiques et les dépendances structurelles. L'analyse des graphes d'appels et des traces d'exécutions peut nous aider à générer les différentes connexions structurelles. En ce qui concerne les connexions sémantiques, nous pouvons tirer parti de techniques d'apprentissage de représentations numériques (embedding) telles que SBERT.

Mots-clés: Microservices, Traitement du Langage Naturel, Décomposition, Architecture Logicielle, Optimisation

Abstract

The microservices architecture highlights multiple benefits for companies and developers. This explains the reason why numerous tech companies choose to migrate their software from a monolithic architecture to one of microservices. However, migrating from a monolithic or a legacy system to a microservices architecture is a complex, risky, and time-consuming task. To ease and improve this process, our work will focus on designing a semi-automatic approach to detect and identify microservices in existing applications. This is a key step toward the overall goal of migrating a monolithic system toward microservices.

Our approach consists in combining methods of clustering and optimization of proximity between structural dependencies at an optimal level of granularity. In this project, we rely on several Artificial Intelligence techniques as well, specifically Machine Learning algorithms, to implement our approach. On one hand, we are performing dependency extraction and clustering. On the other hand, we are implementing methods that will help us optimize the proximity. To obtain the proximity scores, we are targeting both semantic relationships and structural dependencies. Analyzing call graphs and execution traces can help us generate the different structural links or relations. As for the semantic connections, we can take advantage of highly useful embedding such as SBERT.

Keywords: Microservices, Natural Language Processing, Decomposition, Software Architecture, Optimization

Contents

List of tables

List of figures

List of acronyms and abbreviations

MSA	*Microservices Architecture*
NLP	*Natural Language Processing*
ML	*Machine Learning*
MOO	*Multi-Objective Optimization*
NSGA	*Non-dominated Sorting Genetic Algorithm*
NER	Name Entity Recognition

Acknowledgments

I express my deep gratitude to my supervisor, Professor. Houari Sahraoui, for his expertise, guidance, and support throughout this journey. His patience and feedback have been invaluable in shaping this thesis.

I am also thankful to, Professor. Dalila Tamzalit, for her guidance, assistance, and introduction to key resources essential for this work. Her contributions have been invaluable in laying the groundwork for this research.

I extend my heartfelt appreciation to the University of Montreal for the generous support. I would also like to thank the professors and members of the GEODES for fostering a supportive community environment.

My sincere thanks also go to my parents, sisters and close friends for their unwavering affection, understanding, and support throughout this challenging journey. I dedicate this thesis to my parents, whose encouragement and belief in my abilities have been a constant source of motivation.

Thank you to all who, whether directly or indirectly, have contributed to the realization of this endeavor.

Chapter 1

Introduction

1.1. Context

The rise in complex and distributed applications contributed to the emergence of new and improved software architectural styles, such as microservices. Johannes [**61**] describes microservices as small applications with a single responsibility featuring autonomous scalability, isolated testing, and independent deployment.

Microservices or the Microservices Software Architecture (MSA) has become the *de facto* standard for large software systems. The microservices architecture offers a multitude of advantages among which increased scalability particularly stands out [**3, 19, 25, 53**]. The term scalability in the context of microservices refers to a system's ability to adapt to an increasing workload or demand by providing additional resources [**10**]. It is a key factor in the adoption and the success of the microservices architecture [**24**].

In contrast to monolithic applications, which are implemented and executed as a single unit, microservices are designed around a specific and unique capability. Due to the high scalability of microservices, several large companies such as Uber, Spotify, and Netflix migrated their services from their traditional monolithic architecture to microservices. However, transitioning to a microservices architecture from a monolith or legacy system can be difficult and time-consuming.

One of the main challenges in this migration process is identifying microservices in monolithic applications. This requires analyzing the system and its components to effectively extract the candidate microservices. Traditionally, this process relies on the experience of domain experts. However, a variety of criteria and methods can be applied to perform this analysis.

1.2. Problem

Many legacy systems are built using a monolithic architecture, which exhibits a strong coupling and runs as a single unit. This architectural style presents many drawbacks such as scalability and maintenance issues for the system. Migrating such systems to microservices, where components are loosely coupled and run as independent services, is a challenging task that requires multiple steps. The first and most important step for such a migration is to identify potential microservices from the legacy system which is the main focus of our work. This step is vital in ensuring the quality and efficiency of the system migration.

Because of the complexity of the identification task, not all developers have the necessary expertise or experience [**35**] to perform this step effectively. Indeed, it requires a deep understanding of the system architecture, the design principles and properties of microservices, as well as the possible trade-offs to consider.

To address this, several approaches [**29, 49, 56, 57, 62**] have been proposed in the literature to assist developers with the identification of microservices in legacy applications. Some of these approaches are based on the static and dynamic analysis of the source code, while some others leverage machine learning techniques such as clustering or semantic analysis. However, these approaches present some limitations as they are either not considering the right level of granularity (clustering classes instead of methods) or are not considering the semantic similarity when grouping the legacy elements into microservices.

To aid in the migration process, we propose a novel approach to identify microservices through a combination of meta-heuristics and machine learning techniques. Our approach allows the identification of microservices while addressing the limitations of some of the prior approaches.

1.3. Contribution

In this thesis, we propose a methodology to enable the identification of microservices in legacy or monolithic applications. Our work leverages the benefits of combining a state-of-the-art genetic algorithm with a sentence embedding model in Natural Language Processing. To elaborate further, we listed our main contributions with more comprehensive details below:

(1) An **approach** leveraging machine learning and meta-heuristics techniques with an emphasis on the points below:

- Granularity level: our analysis of existing applications focuses on information at the method level. We target the method signatures.

- Semantic analysis: Through the use of NLP methods such as sentence embedding models, we perform a semantic analysis to extract crucial information that is used at a later stage. Semantics play a meaningful role in how we decompose large legacy systems [**23**] because it provides a better understanding of the meaning and potential connections in these systems. Including a semantic analysis in our approach facilitates a more effective and informed decomposition of these systems.

- Genetic Algorithm: To find the best group of potential microservices, we use and harness the power of NSGA-III to model and solve our multi-objective optimization problem. NSGA-III enables the combination of our diverse objectives and provides the means to an optimal solution.

(2) A **semi-automated tool**: A jar [**14**] or war [**48**] file of a monolithic java application is provided as input and a list of potential microservices is given as output.

(3) A **comprehensive evaluation**: we assess our proposed approach in two different ways. This evaluation relies on various metrics and techniques. In the first part, we apply our approach to a set of two benchmark applications to analyze the optimization results obtained. This analysis allows us to gather insights about the decomposition of legacy systems into microservices. On the other hand, we compare our obtained results with the ground truth using evaluation metrics such as precision and recall.

1.4. Outline

The remainder of this thesis is organized as follows. Chapter 2 presents the background necessary to understand our approach. It also describes the related work on microservice identification. Chapter 3 defines our approach using a combination of a genetic algorithm and a language model embedding. In Chapter 4, we evaluate our proposed approach to some candidate applications through the analysis of the optimization results and a comparison of our results to the ground truth using a set of evaluation metrics. Last, in Chapter 5, we summarize our key points and findings while discussing limitations and future work.

Chapter 2

Background and Related Work

In this chapter on the background, we introduce some key notions and concepts that are relevant to our work while providing examples to illustrate them. After defining these important notions, we will cover several approaches and tools similar to our work in a section dedicated to the related works. Last, we will highlight how our approach to the identification of microservices specifically differs from the previously discussed works in the literature.

2.1. Definition and Examples

In the following points, we provide an overview of some essential notions that are useful to better understand our research problem as well as our contributions. Our work lies at the intersection of Software Engineering and Artificial Intelligence, specifically focusing on leveraging machine learning techniques to optimize microservices identification in the domain of software architecture.

2.1.1. Software Architecture

The ultimate goal of our work is to migrate to the microservice architecture. The domain of software architecture involves the different organizations and structures of software [**7**]. In [**52**], Perry and Wolf distinguish software architecture from software design by emphasizing broader principles such as codification, abstraction, standards, formal training for software architects, and style. Multiple styles and patterns can be denoted when it comes to software architecture. Among these, a prominent style currently shaping the scene of modern software architecture is microservices. In contrast, a large family of architectures can be grouped into the category of monolithic architectures.

2.1.2. Monolith architectures

By definition, the word monolith depicts an organized whole that acts as a single unified power or a single large stone [1]; hence, a monolith refers to an entity of a large structure or system. Expanding on this concept, a monolithic architecture is characterized as a cohesive, large-scale structure with a single code base that encompasses all essential business functionalities [**4**]. This architectural style represents the traditional approach to application development.

Figure 2.1 illustrates an example of a simple e-commerce application with a monolithic architecture. In this figure, all components and functionalities (payment, shopping cart, and inventory) of the application are tightly coupled into one single instance. In the event of any modification or issue with one element of the application, the entire application may be impacted, necessitating to be fixed or updated as a whole.

Although monolithic architectures promote simplicity and ease of development, they can encounter several challenges as the application expands. For instance, scalability quickly becomes an issue when there is a surge of requests on the system [**53**]. Moreover, monolithic applications are more prone to complete system disruptions or single points of failure. The failure of a single component can spread and impact the entire application. To address these challenges, transitioning to a microservices architecture can provide an effective solution.

[1]https://www.merriam-webster.com/dictionary/monolith

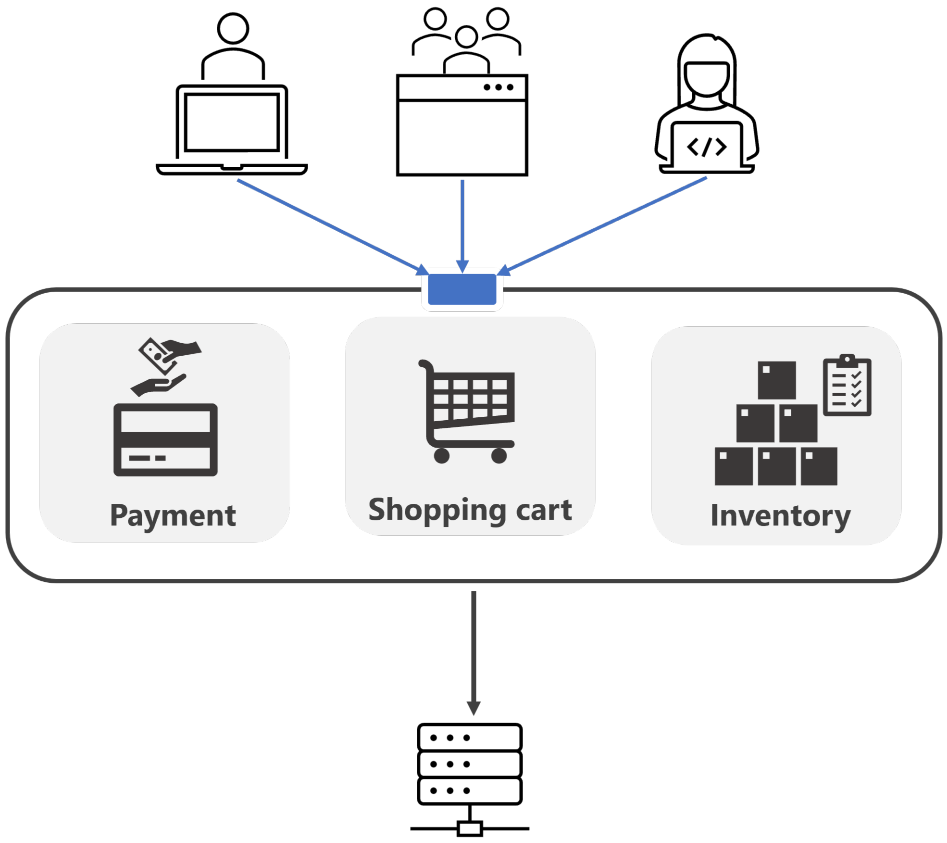


Fig. 2.1. Example of a monolithic architecture.

2.1.3. Microservices

The microservices architecture represents a specific type of software architecture. Microservices are structured as a collection of small, cohesive, and loosely coupled services. Each microservice fulfills only one set of functions in the application and communicates with others using lightweight mechanisms. Additionally, microservices can be developed, deployed, scaled, and fixed independently [**24, 46, 65**]. These characteristics elevate the microservices architecture as a preferred architectural style, presenting numerous advantages over traditional monolithic architectures. For instance, using microservices provides:

- **Increased Flexibility**: since every microservice is a unit, it is easier to bring modifications, maintain, and fix issues in one microservice while the others are still operating. The flexibility provided by microservices mitigates the risks associated with single points of failure.

- **Improved Scalability**: the microservices architecture allows systems to manage an increase in demand and ensure that the quality of service is maintained. This architecture style promotes better resource usage, especially for large software systems.

- **Teams Autonomy**: Because each microservice is a unit on its own and fulfills a specific functionality, this architecture enables different teams to work simultaneously [**22**] without affecting or causing interference.

- **Polyglot Capabilities**: the microservices architecture enables development using multiple programming languages, frameworks, and technologies [**22, 45, 58, 65, 66**].

Multiple properties and principles are associated with the microservices architecture. One of the key features is the single responsibility principle. Figure 2.2 illustrates the single responsibility principle. Each block in the figure is a service and represents a unique responsibility. The services operate independently with their own database while effectively communicating with others.

The single responsibility principle is a guiding concept that states that a service should exist to handle a single major responsibility and only one reason should lead to a potential change of that service [**55**]. In other words, a microservice should have a single and unique responsibility. This guiding principle plays a vital role in ensuring loosely coupled and highly cohesive microservices.

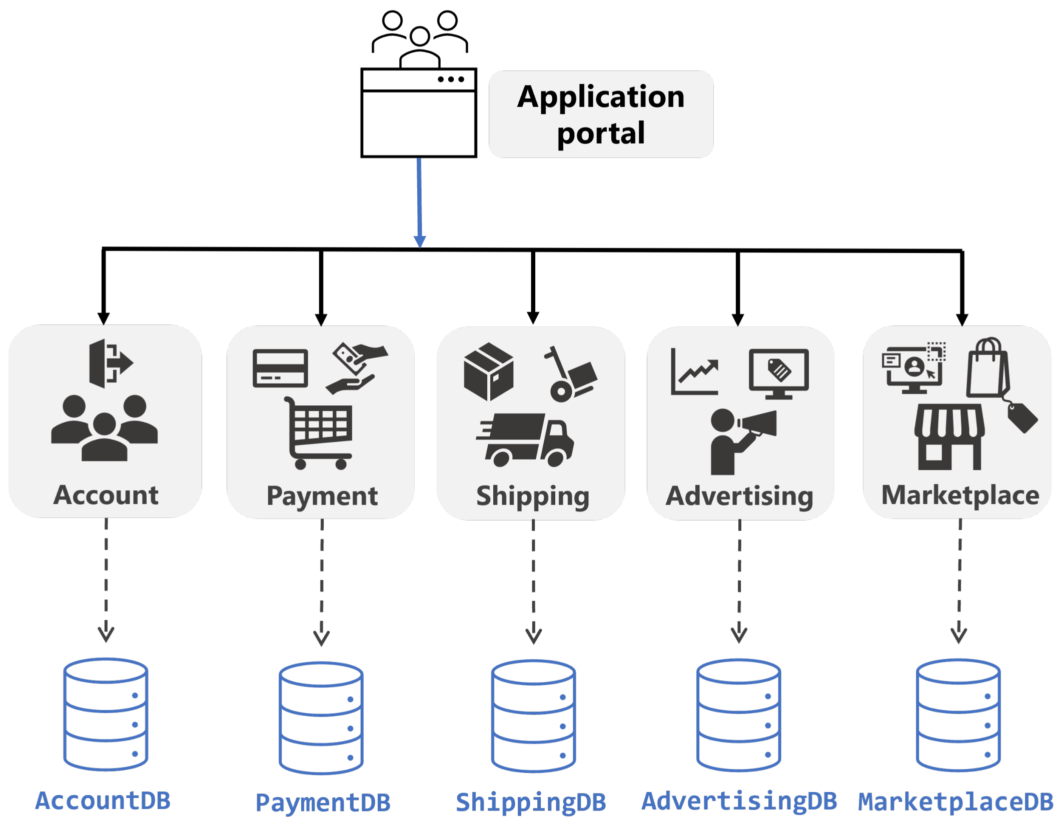


Fig. 2.2. Example of an e-commerce microservices architecture featuring the single responsibility principle.

2.1.4. Genetic Algorithm

In our work, we view the microservice identification as an optimization problem. Genetic algorithms represent a type of *meta-heuristic algorithm* widely used for search and optimization problems in numerous fields. These algorithms were designed based on the process of natural selection and involve genetic operators such as selection, mutation, and crossover [**30, 59**]. Figure 1 depicts an overview of the Genetic Algorithm cycle. Genetic algorithms start with a randomly initialized population and apply genetic operators to the population to generate new and fitter solutions. The process happens through multiple iterations– generations – and eventually converges towards an optimal or near-optimal solution [**33**]. We define below some key terms:

- **Selection** denotes the parents' selection based on fitness characteristics (*fitness function*). It models the survival-of-the-fittest concept in natural selection [**59**].

- **Crossover** is also referred to as recombination because this genetic operator combines individuals from the population in pairs (*parents*) to produce new solutions (*offsprings*).

- **Mutation** mimics the idea of genetic mutations by enabling modifications or alterations to some characteristics of the individuals. Mutation promotes diversity in the population and provides an escape from local optima [**33**].

2.1.5. Non-dominating Sorting Genetic Algorithm III

While classical genetic algorithms aim to optimize a single objective, many optimization problems involve many conflicting objectives. To this end, genetic algorithms have been adapted to solve multi-objective optimization problems. One popular adaptation is the Non-dominating Sorting Genetic Algorithm (*NSGA*), which stands out as a well-established approach.

This family of algorithms was used to solve many software engineering problems, including architectural ones. For example, in the context of software modularization [**11**], Candela et al. explore the limitations of relying solely on only two objectives, i.e., coupling and cohesion. Their findings strongly advocate for considering many objectives to choose the best modularization solutions. These findings and many others motivated us in our choice of a version of NSGA that is capable of handling many objectives, namely NSGA-III [**17**]. Although very popular, the previous version NSGA-II performs well with a few objectives [**12**], generally two. Opting for NSGA-III provides our approach with greater flexibility in implementing more objective functions and exploring the different balanced solutions. The highlight of NSGA-III, as a genetic algorithm, lies in its handling of survival selection, which allows effective trade-offs between our three objective functions.

Algorithm 1 provides an in-depth view of the pseudocode detailing the algorithm that powers NSGA-III survival selection process [**9**]. In NSGA-III, the survival selection process aims to determine the next generation population $P^{(t+1)}$ of size N from the merged population (R(t)), which consists of parent population $P^{(t)}$ and offspring population $Q^{(t)}$. The process begins by sorting individuals in $R^{(t)}$ into non-dominated fronts (F_1, F_2, ...).

The set S, representing the surviving solutions is initially empty. It is gradually filled by iterating through the fronts and appending each front (denoted F_i) to S until the number of individuals reaches or exceeds the population size N. At the end of this iteration, the potential splitting front (F_L) is identified as the front where $|S \cup F_i| \geq N$, which is where the union of the set of surviving solutions (S) and the current front (F_i) contains N or more individuals.

If the total number of individuals in S and F_L equals N, no further splitting is necessary. However, if additional selection is required to determine the surviving individuals, a niching method selects individuals from the potential splitting front (F_L) associated with under-represented reference directions.

Individuals in S are then assigned to reference directions (Z), which represent trade-offs in the M-dimensional objective space. This is based on their normalized values using estimated ideal and nadir points ($\hat{z}^*, \hat{z}^{nad}$). A niche count (ρ) is maintained throughout this process. Finally, the remaining individuals needed to complete the population $P^{(t+1)}$ are selected from the front F_L, hence prioritizing those associated with under-represented reference directions. Overall, NSGA-III prioritizes non-dominated individuals to populate the next generation population $P^{(t+1)}$, aiming to include individuals representing each reference direction near the Pareto-optimal front.

To summarize the updated mating and survival selection in NSGA-III, the mating process involves randomly selecting parents for recombination; and survival selection incorporates the concept of reference directions to represent trade-offs in the M-dimensional objective space, based on objective values [**9**].

Algorithm 1 NSGA-III Survival Selection

Input: Merged Population $R^{(t)}$, Number of surviving individuals N, Reference Directions Z, Ideal Point Estimation $\hat{z}^{*}$, Nadir Point Estimation $\hat{z}^{nad}$

Output: Surviving Individuals $P^{(t+1)}$

```
 1: Compute non-dominated fronts: $(F_1, F_2, \ldots) \leftarrow$ non_dominated_sort($R^{(t)}$)
 2: Initialize $S \leftarrow \emptyset$ and $i \leftarrow 1$
 3: while $|S| + |F_i| < N$ do $S \leftarrow S \cup F_i$; $i \leftarrow i + 1$
 4: $F_L \leftarrow F_i$
 5: if $|S| + |F_L| = N$ then $S \leftarrow S \cup F_L$
 6: else
          /* Normalize objectives space and update boundary estimation*/
 7:           $\bar{S}, \bar{F}_L, \hat{z}^{*}, \hat{z}^{nad} \leftarrow$ normalize($S, F_L, \hat{z}^{*}, \hat{z}^{nad}$)
          /* Compute niche count, assigned $Z_i$, perpendicular dist to $Z_i$*/
 8:           $p, \pi, d \leftarrow 0$
 9:     for $k \leftarrow 1$ to $|S|$ do
10:           $\pi_k, d_k \leftarrow$ associate($\bar{S}_k, Z$); $p_{\pi_k} \leftarrow p_{\pi_k} + 1$
11:     end
          // Remaining individuals from $F_L$ to fill up $S$
12:     $S \leftarrow S \cup$ niching($\bar{F}_L, N - |S|, p, \pi, d$)
13: end
14: $P^{(t+1)} \leftarrow S$
15: return $P^{(t+1)}$
```

2.1.6. Natural Language Processing

In our work, we consider many types of dependencies to identify microservices. Among them, semantic dependencies can be captured with Natural Language Processing (NLP) techniques. NLP is a field of computer science intersecting between artificial intelligence and computational linguistics. NLP encompasses a set of techniques for computers to understand, analyze, and represent human language [**32, 64**]. It involves a large range of natural language tasks such as Part-of-speech POS tagging, named entity recognition, speech recognition, and sentiment analysis. In terms of applications, NLP is used in various cases such as spam detection [**32**], conversational agents, and text summarization. Recently, NLP has seen great advancements with sophisticated techniques and tools generating meaningful and expressive conversations.

2.1.7. Embedding Models

Embedding models play an important role in NLP. They enable the representation of words or sentences under the vector form [**1**]. These vectors encode the syntactic and semantic meaning of words and place words with similar meanings closer to each other in a continuous vector space [**31**]. This capability is directly related to our project as we are seeking semantic dependencies between program constructs. Different types of embeddings are commonly used for everyday tasks such as text classification, sentiment analysis, named entity recognition, and machine translation.

2.1.7.1. Traditional word embeddings-based models.

Word2Vec [**41**] is used in sentiment analysis to capture the tone of words. A few practical applications of Word2Vec range from the detection of hate speech on social media [**43**] to the accuracy improvement of sentiment classification [**50**].

GloVE [**51**] or Global Vectors for Word Representation is an unsupervised learning algorithm, designed to generate vectors for word representation. The algorithm [2] is trained on aggregated global word-word co-occurrence originating from a specific corpus to create word embeddings.

2.1.7.2. Transformers-based models.

BERT or Bidirectional Encoder Representations from Transformers, significantly transformed tasks such as name entity recognition (NER) and machine translation. BERT stands out by being conceptually simple and empirically powerful [**18**]. BERT requires minimal fine-tuning yet produces effective results [**38**]. Compared to traditional word embeddings, BERT employs contextual embeddings, which means its embeddings capture the meaning of a word based on the context surrounding the word.

[2]`https://nlp.stanford.edu/projects/glove/`

SBERT or Sentence-BERT [**54**] is a variant of the BERT model, which utilizes siamese and triplet networks to generate sentence embeddings. SBERT can be used effectively for both lengthier text with multiple sentences and shorter text. Regarding its applications, SBERT is highly efficient and useful for tasks such as semantic textual similarity, paraphrase mining, and semantic search. Figure 2.3 illustrates the architecture of SBERT for assessing similarity, which is a useful measurement for semantic textual similarity.

SBERT is a valuable alternative that mitigates the computational overhead associated with BERT while preserving its high accuracy [**54**]. Table 2.1 provides a summary of the execution time of BERT and SBERT in a clustering task involving 10,000 sentences. On the same dataset, BERT completed the task in 65 hours, considering all the computations of sentence combinations. In contrast, SBERT executed the same task within a time record of 5 seconds. This comparison demonstrates the computational advantage of SBERT, positioning it as an optimal choice for addressing semantic textual similarity [3] in large software systems with a substantial number of methods. In such systems with extensive data, SBERT can prove itself to be highly efficient.

In our work, we use SBERT to measure the similarity between methods that are mapped into sentences.

Model	Sentence count	Execution time
BERT	10,000	65 hours
SBERT-NLI	10,000	about **5 seconds**

Table 2.1. Computational time comparison for a clustering task on a large sentence count between SBERT and BERT [**54**].

[3]Semantic Textual Similarity often requires the computation of millions of method pair combinations to assess the similarity

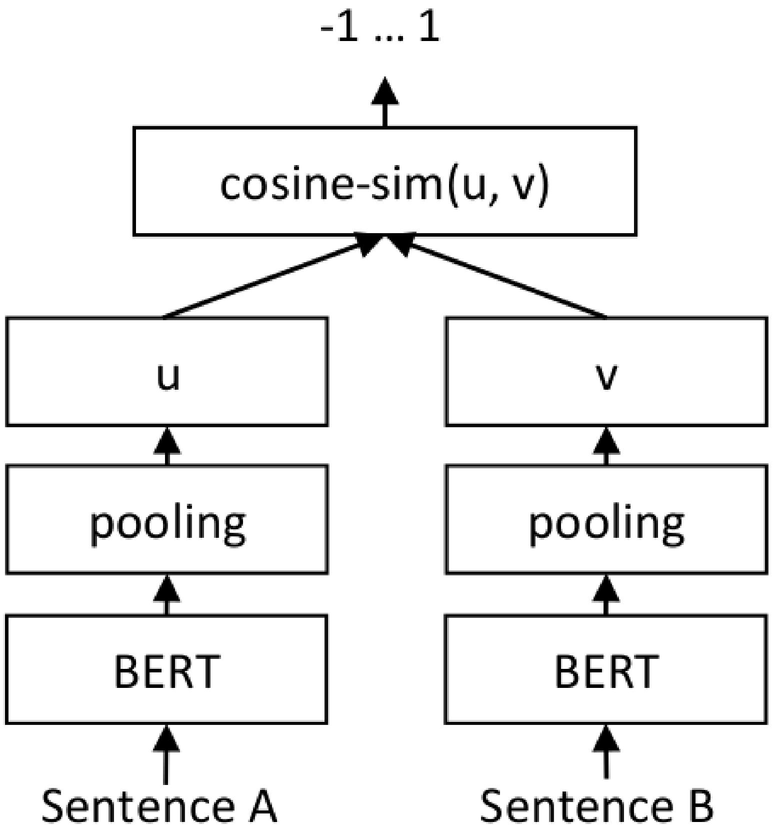


Fig. 2.3. Overview of the SBERT architecture for similarity computation between sentence A and B [**54**]

.

2.2. Related Work

In this section, we present an overview of the most relevant and prominent papers on the identification of microservices. We will describe their work and findings to provide a deeper yet succinct understanding of how they tackle the task of microservices identification.

2.2.1. Mono2Micro

Kalia et al. proposed **Mono2Micro**, which is a tool developed at IBM to help in transitioning monolithic applications to microservices. Mono2Micro uses a spatio-temporal decomposition technique, which relies on business cases and runtime call relations to produce partitions at the class level [**29**]. This technique contributes to creating partitions of classes that are functionally cohesive. After presenting Mono2Micro, the authors conducted an evaluation of their tool by comparing it with four existing techniques on an ensemble of open-source and proprietary applications. As a result, they showed that Mono2Micro can achieve better modularity and independence metrics than the state-of-the-art baselines. Later, Kalia et al. gathered feedback and suggestions from practitioners who have used Mono2Micro. Overall, they reported positive feedback and suggestions on how their tool could be improved.

2.2.2. CARGO

CARGO is an AI graph-partitioning tool developed by Nitin et al. to refine and enrich the partition quality for migrating toward the microservices architecture [**49**]. It is part of *the project Minerva for Modernization* that aims to modernize legacy applications by leveraging the power of AI.

CARGO uses **C**ontext-sensitive l**A**bel p**R**opa**G**ati**O**n to perform community detection in the classes of an application. This technique is a new algorithm for community detection that uses Label Propagation Algorithm (LPA) [**21**] as a foundation to take advantage of the complex dependencies present in the System Dependency Graph (SDG). It helps in distinguishing functional boundaries in the code.

The approach behind CARGO involves three main stages to obtain microservice boundaries. The first step is to build a context-sensitive system dependency graph (SDG). Next, the second stage requires extracting sub-graphs of the previously built SDG to obtain contextual snapshots. In the third and last stage, a variant of the label propagation algorithm is used to cluster functionally bound program components in the different contextual snapshots.

The authors evaluated CARGO on five Java EE applications and showed that their tool can effectively enhance the partition quality of four state-of-the-art microservice partitioning techniques. Furthermore, they demonstrated some other benefits of their tool on database transactional purity as well as run-time performance. Their evaluation showed overall that CARGO can reduce distributed transactions, lower the latency, and increase the throughput of the deployed microservice application.

2.2.3. MSExtractor

In [**56**], Saidani et al. introduce **MSExtractor** as an approach to extract microservices using multi-objective optimization. In their approach, the extraction of potential microservices is presented and modeled as a combinatorial optimization problem. The goal of MSExtractor is to ensure that each class belongs to exactly one microservice. To achieve this, they create empty microservices and use NSGA-II to find the optimal class assignment for these microservices. They define two objective functions in their implementation of NSGA-II: minimizing coupling and maximizing cohesion. Furthermore, Saidani et al. supplemented their approach with an empirical evaluation to measure the efficiency of MSExtractor using a set of architectural metrics, which are connected to cohesion and coupling. The results show that MSExtractor performs better for large-scale software systems compared to other approaches [**2, 27, 39**].

Following the initial development, MSExtractor underwent further refinements to extend the approach and enhance its capabilities. In this evolved version, the authors, Sellami et al. [**57**] used the indicator-based evolutionary algorithm, known as IBEA [**67**], for the decomposition and search process. In their work [**57**], Sellami et al. define three fundamental objectives for the fitness function: finding the optimal granularity of the microservices, minimizing coupling, and maximizing cohesion. Following an evaluation of the approach, the refined version of MSExtractor demonstrated its proficiency in extracting services characterized by loose coupling and cohesion.

2.2.4. MicroMiner

Trabelsi et al. developed a microservices identification approach called **MicroMiner** that focuses on semantics [**62**]. It is a type-based approach, guided by the identification of service types obtained through the prediction of a ML classification model such as Support Vector Machine (SVM) and Graph Convolutional Network (GCN). The semantic analysis in the approach relies on a pre-trained model of Word2Vec based on Google News.

Moreover, Trabelsi et al. describe the MicroMiner approach in three phases. The first phase focuses on class typing, where they classify the classes from the system by using a label assignment. They list the different labels under the following three layers: the Business layer, the Entity layer, and the Utility layer. The second phase involves Typed-Service identification. In this phase, application, entity, and utility services in each of the previous layers are identified with a clustering technique. Services to Microservices mapping represents the third and last phase of the approach. This step is responsible for generating the microservices. To do this, they employ soft clustering, which is a particular type of clustering that allows each element to belong to more than one cluster.

In terms of evaluation, the authors of MicroMiner apply their approach to four systems and show that microservices with high accuracy and completeness can be retrieved when compared to two other existing approaches.

2.3. Synthesis

Table 2.2 illustrates a comparison of various approaches to microservices identification. The comparison includes some essential features of the approaches discussed previously while providing a brief overview of their differences.

In this section, we discuss the differences between our work and the approaches outlined in the related works. Our approach includes a structural analysis, which reveals the different connections between entities of a software, precisely the connections between all methods of such systems. While structural dependencies can provide valuable information on the connections within a system, they capture only a partial view. To address this limitation, we add a semantic analysis to our approach to further explore the meanings and functionalities of the system's methods.

The semantic analysis complements the information obtained from structural dependencies by providing insights into why these methods are linked and whether they may serve a shared functionality. This combination of structural and semantic analysis in our approach can enhance the process of clustering methods more accurately by considering both connections and functionality. Therefore, using semantic analysis in our approach brings more depth to our analysis.

Comparatively, MicroMiner also employs semantic analysis in their approach. They utilize Word2Vec, whereas we leverage SBERT to conduct semantic analysis. SBERT, a variation of BERT, offers a more contextual and efficient alternative for capturing the semantics.

In contrast to MSExtractor, our approach employs NSGA-III for the search and optimization process. In [**57**], Sellami et al., the authors of MSExtractor explain their preference for IBEA over other multi-objective algorithms as they highlight the superior performance of IBEA over both NSGA-II [**16**] and SPEA-II [**68**]. However, our choice of using NSGA-III proves to be more advantageous in addressing the limitations identified in their analysis.

Moreover, NSGA-III outperforms the alternatives and establishes itself as a more effective choice [**42**].

Similarly to Sellami et al. [**57**], our optimization focuses on three objectives. In addition to the standard objectives of coupling and cohesion, we extend our analysis to include the semantic distance among the components within our dataset as a third objective.

In comparison to Mono2Micro and CARGO, our approach leverages the power and diversity of using multi-objective optimization. Choosing NSGA-III allows us to consider several objectives simultaneously and find the best trade-offs among a large diversity of solutions.

In contrast to all the aforementioned approaches, which took into account the classes in the monolithic applications, our approach aims to explore the microservices identification at a fine-grained level, i.e., method level. Indeed, a class can be involved in many microservices in the monolithic design.

Approach	Mono2Micro	CARGO	MSExtractor	MicroMiner
Main Technique	Spatio-temporal decomposition	Context-sensitive label	Multi-objective combinatorial optimization	Identification of service types: type-based approach
Highlighted Analysis	Dynamic analysis for runtime dependencies	Static analysis of the System Dependency Graph (SDG)	Static and semantic (tf-idf) analysis	Semantic analysis: pre-trained model of Word2Vec
Clustering Method	Hierarchical clustering: SLINK	Community detection: Context-based LPA	Indicator-based evolutionary algorithm: IBEA	ML classification models: SVM, GCN
Granularity	Recommendations for partitioning classes	Clustering of functionally bound program classes	Extraction of coarse-grained services at the class-level	Clustering of services (application, entity, and utility classes)
Results	Improved modularity and independence of microservices	Reduced distributed transactions and latency; increased throughput and partition quality	Cohesive and loosely coupled services	Accuracy and completeness of candidate microservices

Table 2.2. Features summary of the approaches

Chapter 3

Approach: Monoliths to Microservices

In this chapter, we present our approach to identifying microservices in existing applications. We perform this task by combining machine learning and meta-heuristics techniques. Our approach consists of four main steps: input preparation, a two-level analysis, problem and algorithm implementation, and clustering solution. We start our approach with input preparation, where we build and clean our data. Next, we continue with a two-level analysis through structural and semantic computations. Then, we model our problem and define specific functions and parameters. This is followed by our implementation of NSGA-III. Finally, we proceed to generate and explore the results of the optimization as clusters, each representing a potential microservice. Figure 3.1 summarizes the key steps of our approach, illustrating its core concepts.

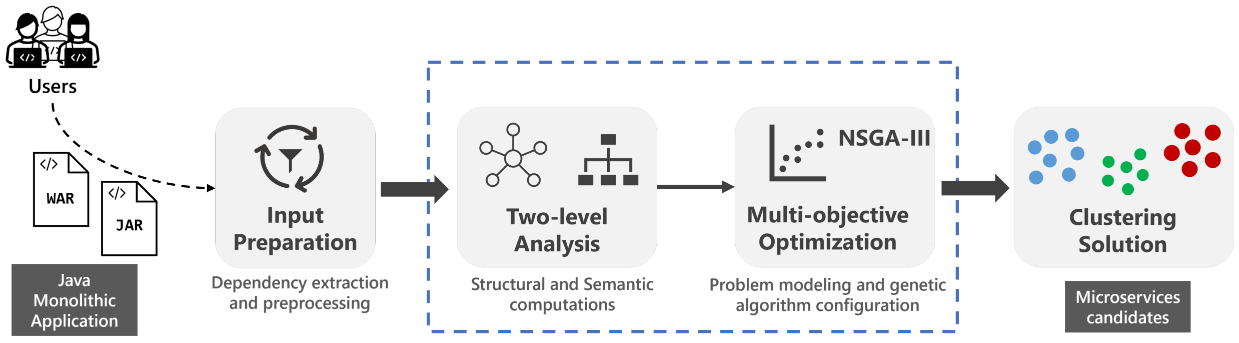

Users
WAR
JAR
Java Monolithic Application
Input Preparation
Dependency extraction and preprocessing
Two-level Analysis
Structural and Semantic computations
NSGA-III
Multi-objective Optimization
Problem modeling and genetic algorithm configuration
Clustering Solution
Microservices candidates

Fig. 3.1. Overview of the microservices identification approach

3.1. Input preparation

3.1.1. Initialization & Source Code

The first step is to build the Java application used for input. Depending on the configuration, we can build it with Maven or Gradle to obtain the jar or war file. From there, we use a static code analyzer to extract static dependencies. Numerous tools and libraries, such as Soot [**34**] and WALA [1], can perform this task. In our work, we use an open-source static code analyzer for Java applications, *javacg*[2] to provide us with information relative to the classes and methods present within the application. After using the static code analyzer, we obtain a text file containing the different dependencies in the input application. Table 3.1 shows an example of the static dependencies data obtained after running the static code analyzer, javacg, on a Java application. These dependencies are explained in 3.1.2.

C:com.ibm.websphere.samples.daytrader.beans.MarketSummaryDataBean com.ibm.websphere.samples.daytrader.util.FinancialUtils
C:com.ibm.websphere.samples.daytrader.beans.MarketSummaryDataBean com.ibm.websphere.samples.daytrader.entities.QuoteDataBean
M:com.ibm.websphere.samples.daytrader.direct.TradeDirect:buy(java.lang.String,java.lang.String,double,int) (M)com.ibm.websphere.samples.daytrader.entities.QuoteDataBean:getPrice()
M:com.ibm.websphere.samples.daytrader.direct.TradeDirect:sell(java.lang.String,java.lang.Integer,int) (M)com.ibm.websphere.samples.daytrader.entities.HoldingDataBean:getQuantity()

Table 3.1. Example of static dependencies

3.1.2. Data Preprocessing

Building a robust dataset is essential for further analysis in any data-driven research. In our approach, we initially focus on understanding the relationships between the different entities present in the input application. Preprocessing our data allows us to transform our

[1]https://github.com/wala/WALA

[2]https://github.com/gousiosg/java-callgraph

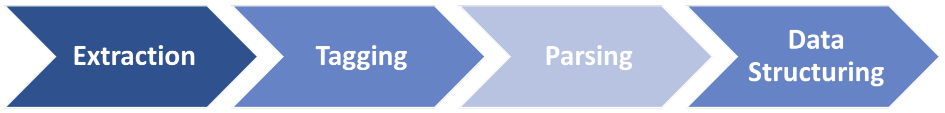


Fig. 3.2. Data Preprocessing pipeline

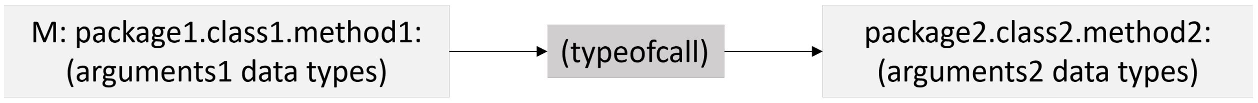

M: package1.class1.method1: (arguments1 data types)
(typeofcall)
package2.class2.method2: (arguments2 data types)

Fig. 3.3. Structure of Extracted Method Dependencies

raw data into a well-structured and insightful dataset. To create our dataset, we select the methods from our text file, which contains both sets of the classes and methods information of the application. In Table 3.1, we have a sample of the various dependencies. Two types of dependency lines are present in our data file, labeled as follows: C for class descriptions and M for method calls.

As illustrated in Figure 3.2, the data preprocessing task unfolds into four main steps:

(1) **Extraction**

Here, we select the interactions of interest from our dependency file. These are the lines labeled with M, which denote the relationships between methods.

(2) **Tagging**

Each extracted method line follows the structure shown in Figure 3.3. In essence, it can be stated that *method1* tagged as M in *class1* from *package1* invokes *method2* tagged (M) located in *class2* within *package2*. Based on the format of the dependency information, we add a tag for each method to make the distinction between caller methods and callee methods. We store these caller-callee methods into a dataframe using pandas[3].

[3]https://pandas.pydata.org/docs/

(3) **Parsing**

This step involves deconstructing the extracted method information into individual parts and assigning clear labels to each component. These labels correspond to the package, class, and method name. For example, consider the parsing process applied to the extracted information in the third row of Table 3.1. In this instance, the caller method is identified and associated with the package labeled *direct*, the class *TradeDirect*, and the method name *buy()*. Conversely, the callee method is linked with the *entities* package, the class labeled as *QuoteDataBean*, and the method name *getPrice()*.

(4) **Data Structuring**

We now have supplemental information about the class, package, and call types of the different methods. In this step, we build a dataframe with pandas to hold the information needed to build our call graph. Several types of connections are present between the methods. Two examples of relationships between methods are described in Figure 3.4. On the left, we denote a one-to-one relationship between method A (caller) and method B (callee) whereas the right side depicts a one-to-many relationship between one caller method and callee methods 1,2, and 3.

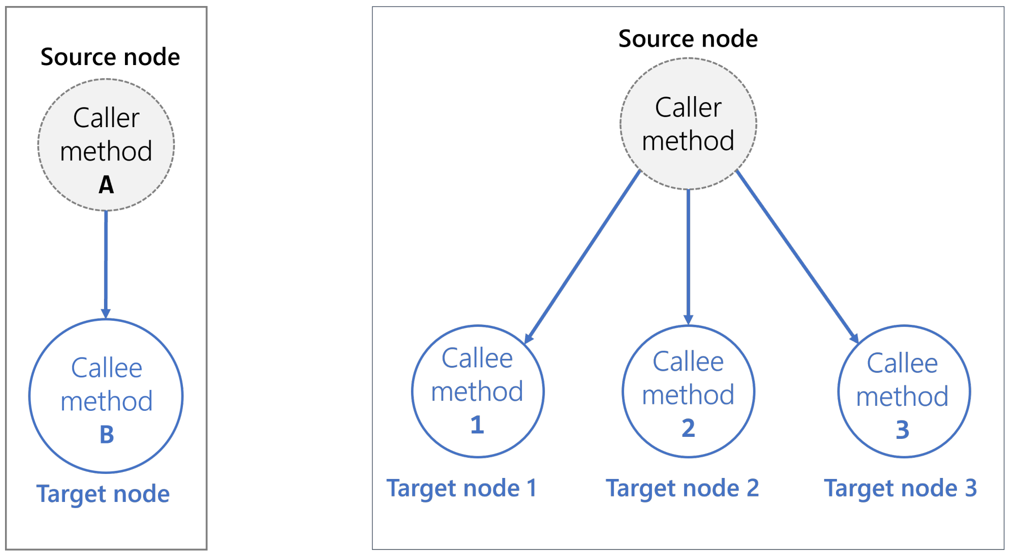


Fig. 3.4. Overview of node relationships

3.2. Analysis

3.2.1. Dependency Graph

Utilizing the data contained in the previously built dataframe, we proceed to the construction of a dependency graph to represent the input. To illustrate the **complexity** and **density** of the systems we work with, the dependency graph of a **large-scale** application is shown in Figure 3.5. This example further clarifies the context in which our proposed approach will be implemented. In the dependency graph, the methods are represented as nodes and the different interactions between these methods as edges. We identify two types of nodes in our graph: source nodes for *caller* methods and target nodes for *callee* methods.

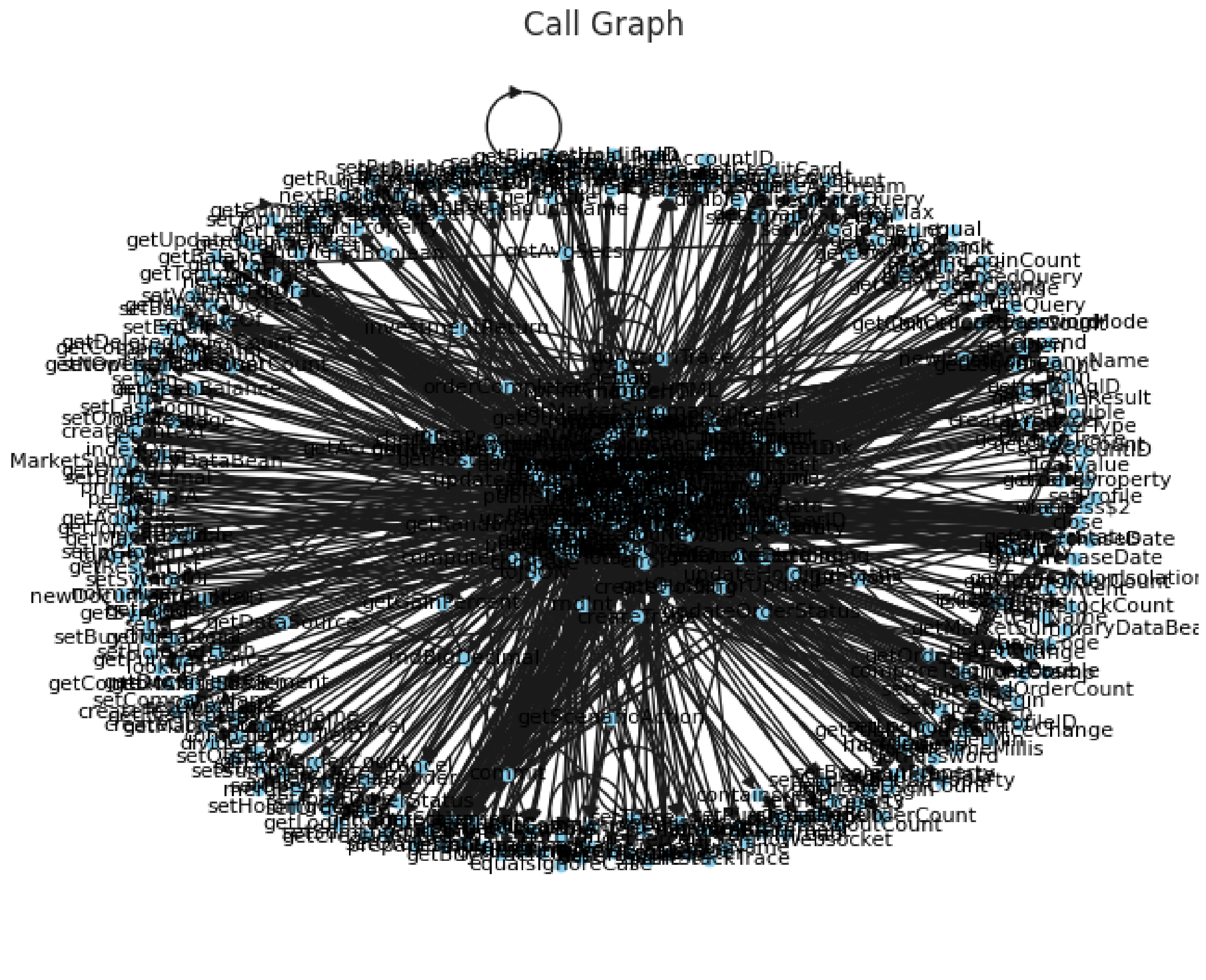


Fig. 3.5. Methods visualization of the call graph generated from a large Java application

3.2.2. Structural Analysis

Following the construction of our call graph, we analyze the structural dependencies so that we can identify subgraphs of methods. During this analysis, we introduce and define several concepts and properties crucial to identifying these subgraphs. Among these, coupling and cohesion are two concepts essential for enhancing the architecture and performance of an application.

3.2.2.1. Coupling.

Coupling refers to the various connections that exist within a call graph from one subgraph to the others. In essence, it measures the degree of interdependence between methods across the different entities or partitions. This concept becomes particularly important when considered as an internal quality metric. Entities are described as being highly coupled when their inter-dependencies are so correlated that a modification in one triggers changes in others reliant on it [**13, 26**]. High coupling can present challenges in modifying, changing, or even updating one module without affecting others. Therefore, we aim to achieve low coupling, which translates to greater independence between subgraphs and hence, better microservices. The formal definition of coupling and calculation examples are provided in Section 3.3.

3.2.2.2. Cohesion.

Cohesion in a call graph is defined as the extent to which nodes are connected within a specific subgraph, focusing on a single functionality. It is an internal metric for dependence within a single entity and [**13**] serves as a valuable metric to assess the structural quality of software. Cohesion is a key concept, especially when considering its direct implications on system refactoring such as the transition from a monolith to microservices. We aim for high cohesion as it promotes better maintainability, enhance feature understanding and improve robustness in microservices. The formal definition of cohesion and calculation examples are provided in Section 3.3. These two metrics contribute to analyzing the structural similarity of our graph. Further details about the computation of the structural similarity are provided in the problem section of this chapter.

3.2.3. Semantic analysis

The semantic analysis involves assessing the semantic textual similarity (STS) among the various methods within our legacy application. We refer to semantic similarity as the extent to which two methods share similar meanings and functionalities. In our approach, we employ sentence embeddings to represent method signatures. These embeddings form the core of our semantic analysis, complementing the previously discussed structural analysis of the software system. Specifically, we use the SBERT model [**54, 60**] for its computational efficiency on large volumes of data. Computing similarity with SBERT is done by using the cosine similarity measure to evaluate the closeness of embeddings within a semantic space. A high similarity score means that the embeddings are closely related to a functionality and suggests that they could belong to the same potential microservice.

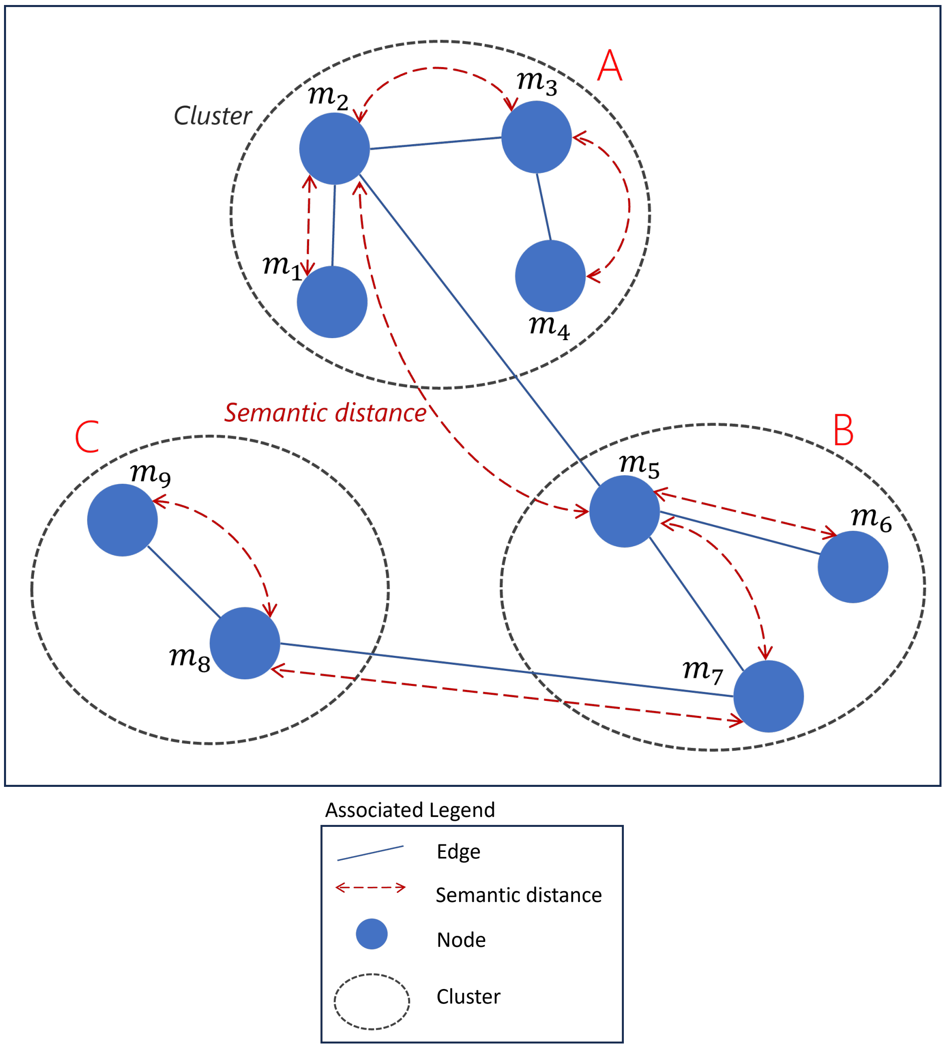


Fig. 3.6. Illustration of a sample graph with three clusters

3.3. Multi-Objective Optimization: NSGA-III

We use the Pymoo [**8**] framework for our implementation of the NSGA-III algorithm described in 2.1.5. First, we set and define our custom problem, then we specify parameters for NSGA-III and proceed to generate and explore the solution set, guided by the fundamental principles of the algorithm.

3.3.1. Problem Definition

3.3.1.1. Adjacency Matrix.

Converting our call graph into a matrix presents several advantages as it provides an efficient representation for our metrics computation. We can use multiple matrix operations to establish relationships between the methods of our large input. Essentially, the call graph is best for visual representation and the adjacency matrix is best suited for the different computations [**5**] we make in our implementation of NSGA-III. We provide below an example of the adjacency matrix for the sample graph illustrated in Figure 3.6. This adjacency matrix holds the encoding of the diverse connections between the methods. 1 denotes a connection between the corresponding pair of nodes whereas 0 implies that there is no existing relationship. For example, ($m3$,$m4$) = 1 indicates that $m3$ and $m4$ are connected by an edge.

$$\begin{bmatrix} & m1 & m2 & m3 & m4 & m5 & m6 & m7 & m8 & m9 \\ m1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ m2 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ m3 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ m4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ m5 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ m6 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ m7 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ m8 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ m9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

3.3.1.2. Problem class definition.

We start by defining the problem as an object and providing parameters such as the number of clusters and the adjacency matrix. We also provide metadata information such as the number of variables (n_var), the lower bound (xl) and higher bound (xu), and the number of objectives (n_obj), which are further detailed next.

3.3.1.3. Objective functions.

In 3.6, we provide a sample graph to illustrate how we compute structural and semantic measures. In the example provided, we have a call graph with 9 nodes representing methods labeled $m_1, m_2, m_3, ..., m_9$. These nodes are spread into three sub-graphs characterized by clusters A, B, and C. To ensure clarity, we use the following terms:

- $V_{cluster}$ for a set of nodes in a particular cluster,
- E_{intra} for a set of edges inside a cluster,
- E_{inter} for a set of edges connecting nodes from different clusters.

Our optimization considers three points: coupling, cohesion, and semantic distance. To perform this, we implemented 03 functions, referred to as objectives functions, to define the optimization for each of these points. Note that in the implementation, we leverage the information of our adjacency matrix to compute the aforementioned measures.

(1) **Minimizing Coupling**.

The first objective function is to minimize coupling. In graph G, illustrated in Figure 3.6, certain nodes within one cluster are interconnected with nodes belonging to other clusters. For example, node $m_2 \in clusterA$ is connected to node $m_5 \in clusterB$. In this case, we talk about *inter-dependence* between the clusters A and B.

To obtain the coupling value, we will compute the coupling $coupling_{(x,y)}$ for each pair of clusters, as a first step, before calculating the overall coupling $coupling_G$ for the graph. Because we work at the method level, the coupling for each pair of clusters is equivalent to the number of edges connecting nodes from one cluster to another:

$$coupling_{(x,y)} = \mid E_{inter} \mid \tag{3.3.1}$$

, with x and y representing different clusters and $\mid E_{inter} \mid$ as the number of edges connecting these clusters.

With the coupling measures for all pairs of clusters, we can compute the average coupling by applying this formula:

$$coupling_G = \frac{\sum coupling_{(x,y)}}{\mid E_{intra} + E_{inter} \mid} \tag{3.3.2}$$

, with $\mid E_{intra} + E_{inter} \mid$ as the total number of edges in graph G, and (x, y) as a pair of clusters x and y from the same graph.

Table 3.2 provides the coupling measure of each pair within our sample graph G. Applying the formula in 3.3.2, the overall coupling across the entire graph G is represented by:

$$coupling_G = \frac{1+0+1}{6+2} = \frac{2}{8} = 0.25$$

Inter-dependence	Value	Intra-cluster dependency	Value
$coupling_{(A,B)}$	1	$cohesion_{(A)}$	$\frac{3}{4}$
$coupling_{(A,C)}$	0	$cohesion_{(B)}$	$\frac{2}{3}$
$coupling_{(B,C)}$	1	$cohesion_{(C)}$	$\frac{1}{2}$

Table 3.2. Measure of inter-dependence and intra-cluster dependency in the graph G

(2) **Maximizing Cohesion**.

Next, we want to maximize cohesion as our second objective function. Calculating the cohesion in a group means evaluating the degree to which the nodes in that subgroup are connected. Using our sample graph G in Figure 3.6, we start by examining the relationships between the nodes of each cluster. In each of these clusters, we have a set of nodes connected by edges. For example, we refer to the nodes m_8 and m_9 in cluster C as an example of *intra-cluster dependency.*

The overall cohesion in the graph is equivalent to the average of the relational cohesion [**37**] in each cluster. To assess the cohesion of cluster x, we compute the proportion of internal edges over the total number of nodes in that cluster:

$$cohesion(x) = \frac{E^x_{intra}}{V_{cluster\_x}} \tag{3.3.3}$$

After calculating the cohesion for each cluster with the formula 3.3.3, we can find the cohesion in the graph G as follows:

$$Cohesion_G = \frac{\sum cohesion(i)}{\sum clusters} \tag{3.3.4}$$

, with i denoting a particular cluster and $\sum clusters$ the total number of clusters present in the graph.
Using the computed cohesion measure for each cluster, as presented in Table 3.2, we derive the overall cohesion of the graph as follows:

$$Cohesion_G = \frac{\frac{3}{4} + \frac{2}{3} + \frac{1}{2}}{3} = 0.63\overline{8} \approx 0.64$$

(3) **Maximizing the semantic distance**.
Lastly, our third objective function targets the semantic distance and aims to maximize this measure. The evaluation of the semantic distance using SBERT involves computing the cosine similarity between pairs of connected nodes. The goal is to achieve a low semantic similarity score for nodes belonging to different clusters, inferring a high semantic distance.

For instance, considering the pair $(m2, m5)$ in graph G, the semantic similarity score is expected to be low as these nodes belong to different clusters ($m2 \in Cluster(A)$ and $m5 \in Cluster(B)$). In this, a low similarity score implies a large semantic distance, which ensures that there is functionality uniqueness for potential microservices.

The process involves the following steps:

- Creating a list to store node information.
- Embedding text: with pre-trained contextual embeddings, node information is transformed into dense vector representations.
- Assessing semantic similarity: After embedding the node information, we evaluate the semantic similarity by applying the cosine similarity measure to the obtained vector representations of the nodes. For reference, the cosine similarity measures the cosine of the angle between two vectors [**63**]. The cosine similarity formula between vectors α and β is:

$$Cosine\,Similarity = \frac{\alpha \cdot \beta}{\parallel \alpha \parallel \cdot \parallel \beta \parallel} \tag{3.3.5}$$

, where $\parallel \alpha \parallel$ and $\parallel \beta \parallel$ represent the norm or length of the vectors α and β.

The cosine similarity score ranges from -1 to 1 and offers insights into the degree of closeness. A 0 score suggests that the vectors are orthogonal and they have no similarity. A score of -1 indicates exact opposite and complete dissimilarity whereas a score close to 1 suggests high similarity.

Building upon the computation process for the different objectives (coupling, cohesion, semantic similarity in graph G), we establish our objective functions as:

$$\begin{cases} f_1 = min\,(coupling) \\ f_2 = max\,(cohesion) \\ f_3 = max\,(semantic\ distance) \end{cases}$$

3.3.2. Algorithm Parameters

After defining our problem and its objective functions, we proceed to configure the parameters for our algorithm. In this section, we discuss the concepts guiding the setup of the NSGA-III algorithm and its execution.

3.3.2.1. Algorithm Initialization.

The initialization of the NSGA-III algorithm is a critical step in setting up the optimization process. For this step, we establish key parameters such as the population size and the reference directions. The population size is based on information from the dataset, specifically the dimensions of our adjacency matrix represented by the number of nodes (methods) in the dataset.

Furthermore, the reference directions play a crucial role in guiding the search for non-dominated solutions. The reference directions represent trade-offs between different objectives, which enables NSGA-III to explore diverse solutions effectively based on the objective values [**9**]. A simple way to grasp the concept behind the reference directions in NSGA-III is to visualize them as vectors that point towards different trade-offs in the objective space.

For example, in a three-dimensional objective space, reference directions could represent different trade-offs between the objectives defined. We use the *das-dennis* method [**15**] to define these reference directions, set the number of objectives ($M = 3$), and the desired number of partitions (*n_partitions*). Moreover, with NSGA-III, custom crossover and mutation operators can be set and integrated to our optimization problem.

3.3.2.2. Algorithm Execution.

This step involves initiating the optimization process of the NSGA-III algorithm with the defined problem and other supporting parameters. During the execution, NSGA-III iterates through multiple generations and leverages the reference directions to guide its search and evolution of the population.

3.4. Clustering Solution

After configuring the NSGA-III algorithm, we execute it for a predefined number of generations to achieve population evolution. Following the execution of the NSGA-III algorithm, the result is a set of solutions known as the Pareto-optimal set [**40, 47**], obtained through the optimization process.

To comprehend the methodology behind the formation of the solution set, it is essential to note that NSGA-III uses a non-dominated sorting approach. This approach means it ranks the different solutions based on the dominance relationships. Therefore, a solution is included in the Pareto set if no alternative solution can improve at least one objective without compromising any other objective [**47**].

Moreover, we can further refine the solutions obtained with NSGA-III by applying a clustering process. It involves grouping similar solutions into a predetermined number of clusters based on their proximity in the objective space. By identifying clusters in the Pareto-optimal set, using NSGA-III provides insights into the solutions distribution along the reference directions in the objective space, guiding decision-making and solution selection.

In summary, a clustering solution is represented by a decomposition of a set of methods into clusters. This decomposition results from the optimization of trade-offs between objectives such as minimizing coupling, maximizing cohesion, and maximizing semantic distance. Each method is then mapped to the corresponding cluster based on this optimization and each cluster represents a potential microservice.

For example, let's consider a sample food delivery system where a solution can be represented as $S = \{3,3,1,1,1,1,2,2,2\}$. This implies that solution S contains 3 clusters with 4 methods in Cluster 1, 3 methods in Cluster 2, and 2 methods in Cluster 3. Each cluster represents a potential microservice M. To illustrate, Table 3.3 summarizes the content of each cluster. Cluster 1 contains methods related to order management, such as placeOrder, trackOrder, cancelOrder, and applyPromoCode, while Cluster 2 includes methods related to menu browsing such as browseMenu, viewSpecials, and searchItem. Cluster 3 includes methods related to user account management, more specifically updatePaymentInfo and setDefaultAddress. It is important to note that while methods within a cluster may not always have the strongest semantic connection, they could be structurally connected and still be part of the same cluster since our optimization considers the combination of structural and semantic connections.

Cluster 1	Cluster 2	Cluster 3
placeOrder	browseMenu	updatePaymentInfo
trackOrder	viewSpecials	setDefaultAddress
cancelOrder	searchItem	
applyPromoCode		

Table 3.3. Example of a clustering solution content for a sample food delivery system

3.5. Conclusion

In this chapter, we have outlined our approach to identifying candidate microservices within applications with monolithic architectures. By leveraging the evolutionary capabilities of NSGA-III and the semantic analysis provided by SBERT, we can effectively explore various trade-offs among potential solutions. Through the power of NSGA-III optimization, we generate optimal and non-dominated solutions, which are then organized into clusters. These clusters of methods serve as candidate microservices, offering a solution with more manageable and scalable components.

In the upcoming chapter, we will assess the efficacy of our proposed approach across a selection of monolithic architecture applications. This evaluation will involve comparing our results against ground truth data, providing valuable insights into the effectiveness and applicability of our methodology.

Chapter 4

Evaluation

In this chapter, we evaluate our approach and discuss our results. In the first section on benchmarks, we apply our approach to a set of monolithic applications to evaluate the optimization of metrics based on the number of partitions. Then in the second section, we compare the results of our approach with the ground truth data, compute evaluation metrics, and discuss the scores obtained. Lastly, we present some threats to the validity of our evaluation.

4.1. Benchmark applications

RQ1: How can the architectural metrics optimization guide the choice of an optimal number of partitions?

4.1.1. Input overview

To evaluate our approach, we use two benchmark applications that differ in scaling. We apply the methodologies outlined in our approach to these two Java enterprise applications: Day Trader[1] and Plants[2]. These applications, further detailed below, are built using a monolithic architecture:

[1]https://github.com/WASdev/sample.daytrader7

[2]https://github.com/WASdev/sample.plantsbywebsphere

- **DayTrader**: This application developed with Java EE7 mimics a system for online stock trading. It allows users to interact with the platform through multiple functionalities such as signing in, overseeing their portfolio, and managing stock share transactions.
- **Plants**: This is a small Java EE6 version of the Plants by WebSphere Sample, which uses a relational database to simulate an e-commerce application for purchasing plants. Customers can manage their accounts, browse for items to purchase, and place orders. We use the original version running on Liberty.

Additional information is provided in Table 4.1 regarding our input applications, DayTrader and Plants by WebSphere.

Application	Domain	classes	methods	Version
Daytrader	Stock e-trading	109	385	7
Plants	Plants e-commerce	33	338	6

Table 4.1. Input summary of the monolithic applications

4.1.2. Implementation and Results

We use the Java static code analyzer *javacg* on the Daytrader jar file and Plants war file to obtain the application dependencies, required as input for our tool. To run the experiment, the user provides the number of partitions, needed to set the number of clusters for the distribution of methods. We use the terms *partitions* and *clusters* interchangeably. Both terms refer to the same concept in this chapter.

The number of partitions aims to represent the application's functionalities or business capabilities as closely as possible. However, it is important to note that the desired number of functionalities might not always be explicitly known in a monolithic application, and the scale (i.e., the number of classes, or methods) of an application can be an indicator of the potential range of partitions to use as input. Varying the number of partitions based on the scale of an application and observing the optimization results for coupling, cohesion, and semantic distance can help gain insights into the application functionalities and hence provide a starting point for the number of microservices.

We apply our approach to our two benchmarks (Daytrader and Plants) to evaluate the impact of the number of clusters on the optimization of the objectives (i.e., coupling, cohesion, semantic distance). We define the potential range of clusters to 3, 5, 8, 10 beginning with 3—the minimum count of known functionalities shared by both monolithic applications, as listed in the descriptions in 4.1.1. We run the optimization for 50 generations and obtain the results for the Pareto front. Table 4.2 and Table 4.3 summarize the results obtained from our objectives functions based on the number of clusters.

	Coupling	Cohesion	Semantic Distance
num_clusters 3	0.3	1.071	0.296
	0.315	1.106	
	0.255	1.033	
	0.226	1.03	
num_clusters 5	**0.352**	**0.708**	**0.282**
	0.341	**0.652**	
	0.333	**0.623**	
num_clusters 8	0.368	0.454	0.337
	0.357	0.404	
num_clusters 10	0.399	0.343	0.313
	0.351	0.306	
	0.395	0.331	
	0.43	0.344	
	0.367	0.323	

Table 4.2. Daytrader - Partitions count and objective functions

	Coupling	Cohesion	Semantic Distance
num_clusters 3	**0.326**	**0.653**	**0.339**
	0.265	**0.643**	
	0.336	**0.692**	
num_clusters 5	0.357	0.371	0.417
	0.37	0.408	
	0.315	0.34	
num_clusters 8	0.346	0.229	0.342
	0.409	0.274	
	0.35	0.249	
	0.36	0.253	
	0.394	0.264	
num_clusters 10	0.373	0.182	0.325
	0.395	0.197	
	0.427	0.207	
	0.337	0.177	
	0.456	0.214	
	0.425	0.202	

Table 4.3. Plants - Objective functions and partitions count

4.1.3. Discussion

The Pareto front of our multi-objective optimization problem represents a set of non-dominated solutions with a trade-off between low coupling, high cohesion, and high semantic distance. In refining our results, our focus is on identifying a solution that exhibits the highest cohesion among the already optimized results of the Pareto Front. Selecting an optimized solution with the highest cohesion can provide internal consistency in the microservices.

It is important to note that, as presented in 3.2.2.2 and 3.3.1.3, our cohesion metric measures the intra-cluster dependency of method calls. Because this metric addresses internal edges, it may yield in some cases scores above 1, which reflects highly dense interconnectivity within a cluster. We aim to select a solution with a cohesion score close to, but less than 1 for a balanced interconnectivity. The number of clusters associated with such a solution can help guide us toward an optimal microservices structure. For instance, when using 3 clusters with the DayTrader application, we observe cohesion scores above 1. This is not optimal as we target cohesion values close to, but not exceeding 1, suggesting that a solution with more clusters might be necessary to better manage the application's complex and diverse functionality. For the DayTrader application, which consists of 109 classes and 381 methods, using 5 clusters offers the most favorable cohesion scores, namely 0.708, 0.652, 0.623.

In contrast, Plants, a relatively smaller application with 33 classes and 338 methods, demonstrates that 3 clusters are more effective, as it yields the best optimization results. This indicates that the functionalities may be managed better with fewer microservices. These findings highlight the relationship between the scale of a monolithic application and an optimal number of microservices.

4.2. Ground Truth

RQ2: How do the results from our approach compare to the ground truth?

4.2.1. Experimental Setup

We use two Java Spring boot versions of the Spring PetClinic application [3] as ground truth. The Spring PetClinic project has a monolithic version, spring-petclinic [4], and a distributed version for microservices, spring-petclinic-microservices[5]. PetClinic is a sample application for a pet clinic management system handling veterinarians' and pets' data.

[3]https://spring-petclinic.github.io/

[4]https://github.com/spring-petclinic/spring-petclinic-angularjs

[5]https://github.com/spring-petclinic/spring-petclinic-microservices

The distributed version comprises the following 7 microservices: vets-service, visits-service, customers-service, api-gateway, admin-server, config-server, and discovery-server. These microservices represent the different functionalities of the application. We use the number of microservices in the PetClinic distributed version as our input number of desired partitions.

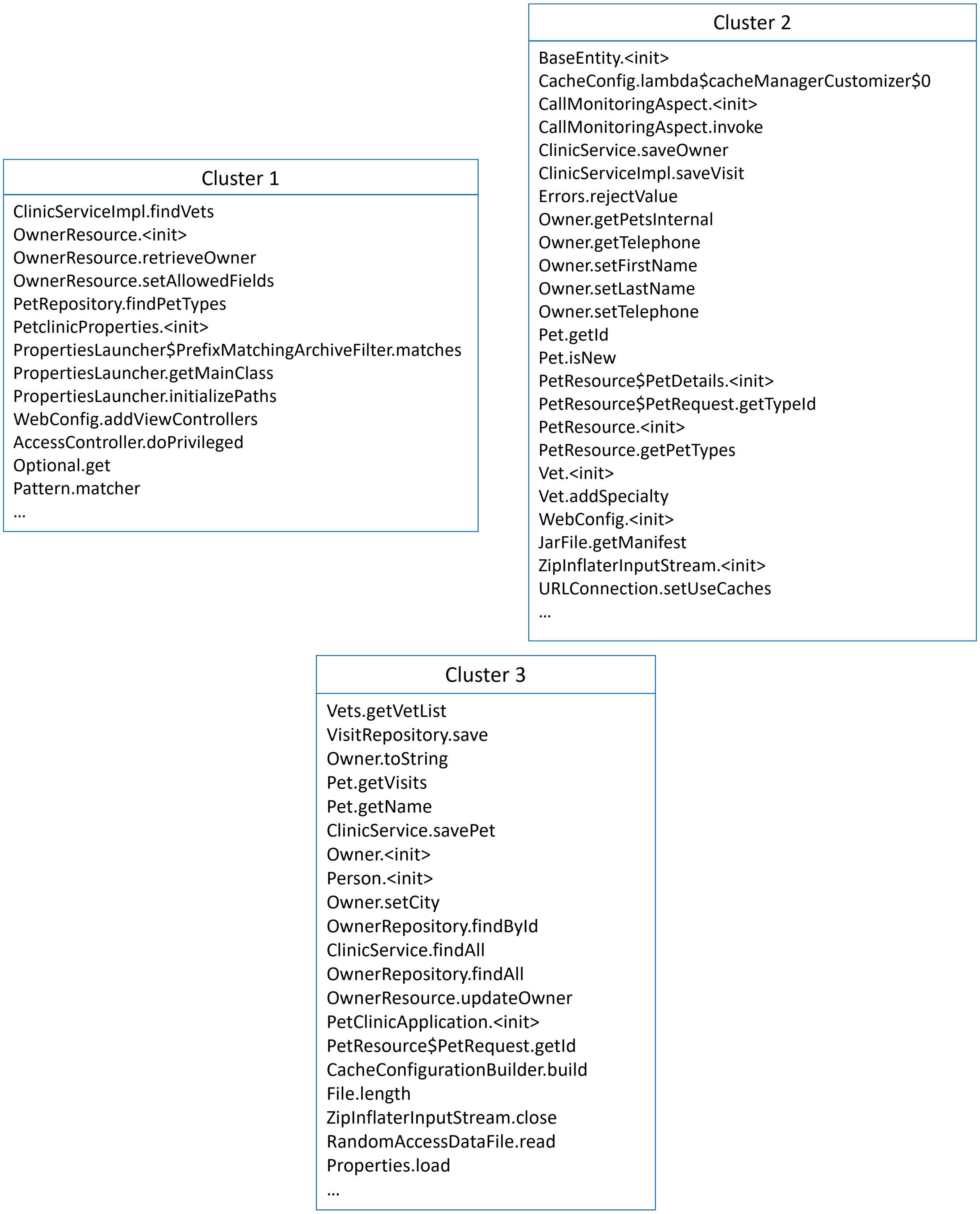
To evaluate our approach against the ground truth, we generate the jar files of the applications mentioned previously. Then, we apply our approach to the monolithic spring-petclinic with our tool Mic_ID to generate clusters.

4.2.2. Results Overview

Figure 4.1 illustrates the clusters obtained from our approach to the monolithic version of PetClinic. They represent the potential microservices for the PetClinic application. The three best clusters 2, 3, and 5 correspond to the results for the microservices *vets service*, *customers service*, and *visits service*.

A cluster contains a list of methods following the structure *class.method*. When analyzing the clusters' content, we observe that methods are not solely grouped based on the semantic relationships, but also with consideration of the structural relationships.

For example, we have methods in cluster 2 such as *setFirstName* and *setLastName* from the *Owner* class which showcase a semantic similarity. In contrast, we also have methods such as *getTelephone* from the *Owner* class and *getId* from the *Pet* class which may not share the strongest semantic connection but can be linked through a structural relationship. Furthermore, we observe that certain classes, including *Owner*, *Pet*, *ClinicService*, and *Vet*, among others appear across different clusters as depicted in Figure 4.1.



Cluster 1

ClinicServiceImpl.findVets
OwnerResource.<init>
OwnerResource.retrieveOwner
OwnerResource.setAllowedFields
PetRepository.findPetTypes
PetclinicProperties.<init>
PropertiesLauncher$PrefixMatchingArchiveFilter.matches
PropertiesLauncher.getMainClass
PropertiesLauncher.initializePaths
WebConfig.addViewControllers
AccessController.doPrivileged
Optional.get
Pattern.matcher
…

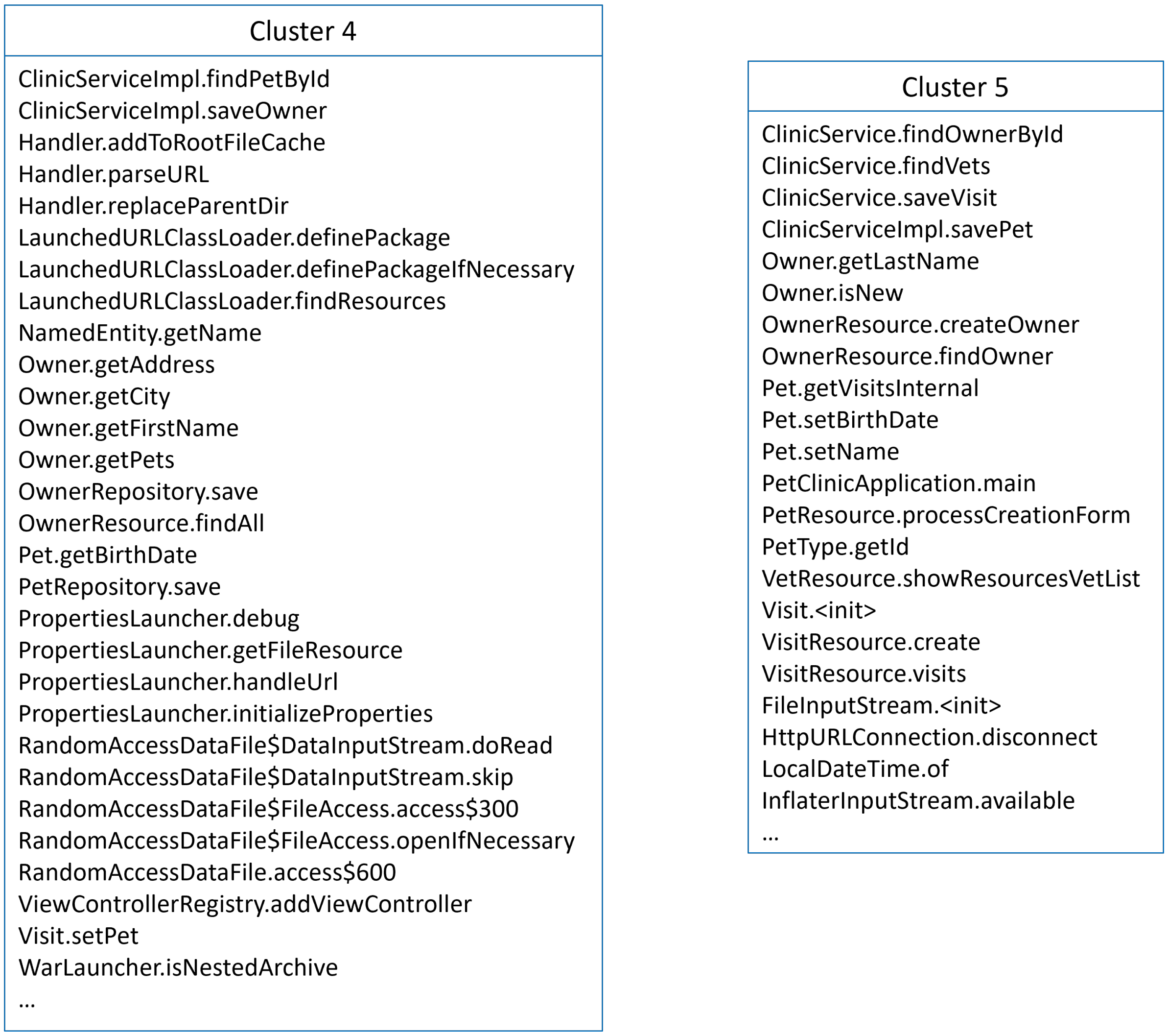
Cluster 2

BaseEntity.<init>
CacheConfig.lambda$cacheManagerCustomizer$0
CallMonitoringAspect.<init>
CallMonitoringAspect.invoke
ClinicService.saveOwner
ClinicServiceImpl.saveVisit
Errors.rejectValue
Owner.getPetsInternal
Owner.getTelephone
Owner.setFirstName
Owner.setLastName
Owner.setTelephone
Pet.getId
Pet.isNew
PetResource$PetDetails.<init>
PetResource$PetRequest.getTypeId
PetResource.<init>
PetResource.getPetTypes
Vet.<init>
Vet.addSpecialty
WebConfig.<init>
JarFile.getManifest
ZipInflaterInputStream.<init>
URLConnection.setUseCaches
…

Cluster 3

Vets.getVetList
VisitRepository.save
Owner.toString
Pet.getVisits
Pet.getName
ClinicService.savePet
Owner.<init>
Person.<init>
Owner.setCity
OwnerRepository.findById
ClinicService.findAll
OwnerRepository.findAll
OwnerResource.updateOwner
PetClinicApplication.<init>
PetResource$PetRequest.getId
CacheConfigurationBuilder.build
File.length
ZipInflaterInputStream.close
RandomAccessDataFile.read
Properties.load
…

Fig. 4.1. PetClinic - Overview of the clusters generated by our tool Mic_ID



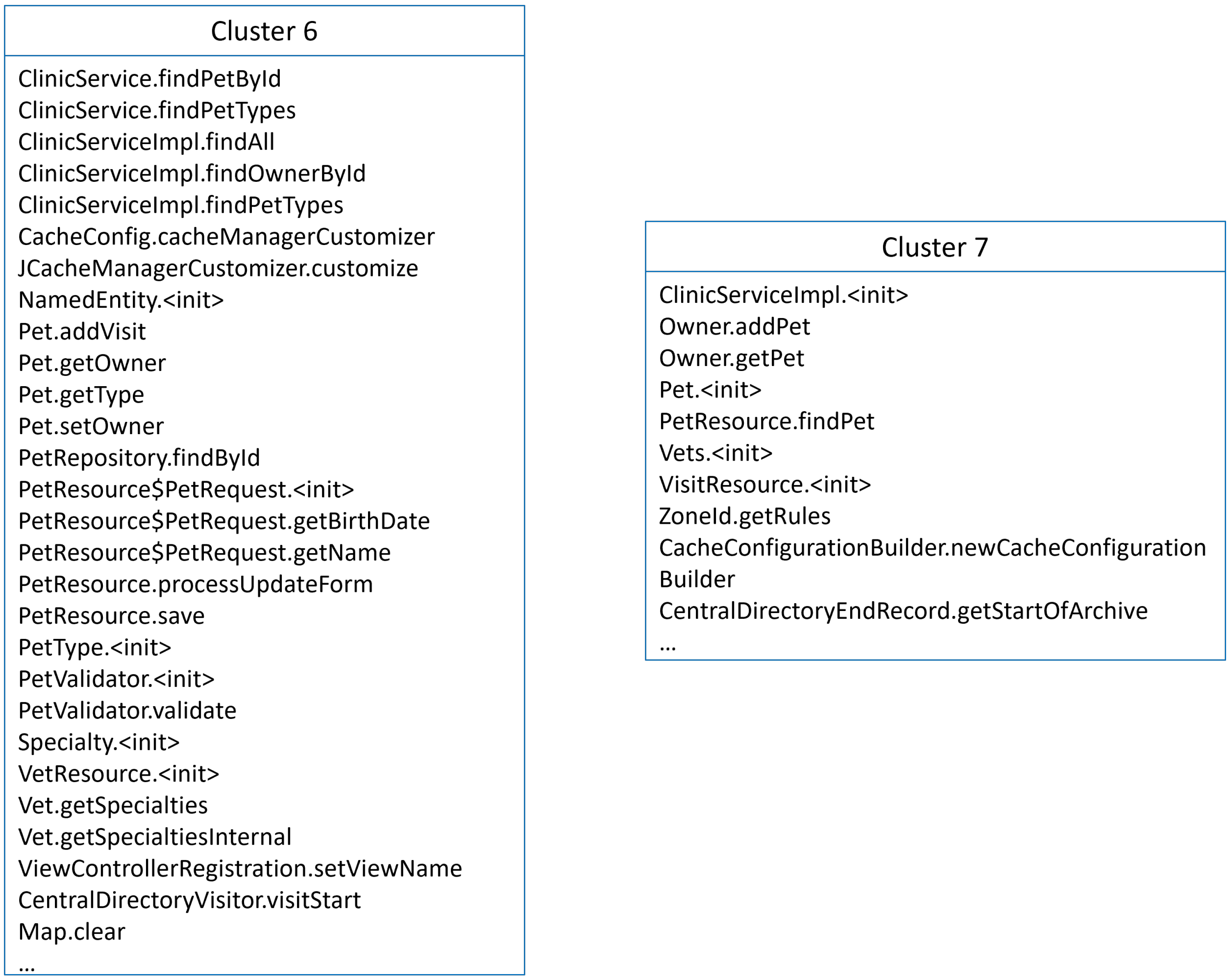
Cluster 4

ClinicServiceImpl.findPetById
ClinicServiceImpl.saveOwner
Handler.addToRootFileCache
Handler.parseURL
Handler.replaceParentDir
LaunchedURLClassLoader.definePackage
LaunchedURLClassLoader.definePackageIfNecessary
LaunchedURLClassLoader.findResources
NamedEntity.getName
Owner.getAddress
Owner.getCity
Owner.getFirstName
Owner.getPets
OwnerRepository.save
OwnerResource.findAll
Pet.getBirthDate
PetRepository.save
PropertiesLauncher.debug
PropertiesLauncher.getFileResource
PropertiesLauncher.handleUrl
PropertiesLauncher.initializeProperties
RandomAccessDataFile$DataInputStream.doRead
RandomAccessDataFile$DataInputStream.skip
RandomAccessDataFile$FileAccess.access$300
RandomAccessDataFile$FileAccess.openIfNecessary
RandomAccessDataFile.access$600
ViewControllerRegistry.addViewController
Visit.setPet
WarLauncher.isNestedArchive
…

Cluster 5

ClinicService.findOwnerById
ClinicService.findVets
ClinicService.saveVisit
ClinicServiceImpl.savePet
Owner.getLastName
Owner.isNew
OwnerResource.createOwner
OwnerResource.findOwner
Pet.getVisitsInternal
Pet.setBirthDate
Pet.setName
PetClinicApplication.main
PetResource.processCreationForm
PetType.getId
VetResource.showResourcesVetList
Visit.<init>
VisitResource.create
VisitResource.visits
FileInputStream.<init>
HttpURLConnection.disconnect
LocalDateTime.of
InflaterInputStream.available
…

Figure 4.1 (continued): clusters 4-5



Cluster 6

ClinicService.findPetById
ClinicService.findPetTypes
ClinicServiceImpl.findAll
ClinicServiceImpl.findOwnerById
ClinicServiceImpl.findPetTypes
CacheConfig.cacheManagerCustomizer
JCacheManagerCustomizer.customize
NamedEntity.<init>
Pet.addVisit
Pet.getOwner
Pet.getType
Pet.setOwner
PetRepository.findById
PetResource$PetRequest.<init>
PetResource$PetRequest.getBirthDate
PetResource$PetRequest.getName
PetResource.processUpdateForm
PetResource.save
PetType.<init>
PetValidator.<init>
PetValidator.validate
Specialty.<init>
VetResource.<init>
Vet.getSpecialties
Vet.getSpecialtiesInternal
ViewControllerRegistration.setViewName
CentralDirectoryVisitor.visitStart
Map.clear
…

Cluster 7

ClinicServiceImpl.<init>
Owner.addPet
Owner.getPet
Pet.<init>
PetResource.findPet
Vets.<init>
VisitResource.<init>
ZoneId.getRules
CacheConfigurationBuilder.newCacheConfiguration Builder
CentralDirectoryEndRecord.getStartOfArchive
…

Figure 4.1 (continued): clusters 6-7

4.2.3. Evaluation metrics

4.2.3.1. Overview.

We evaluate our results by using metrics such as precision and recall. Before computing these metrics, we start by generating a correlation matrix to get a sense of matching between the identified clusters and the microservices.

In the context of identifying microservices in a monolithic application, we refer to the precision and recall metrics as follows:

- **Precision** measures the level of accuracy of our tool in identifying the methods of the generated clusters (from the monolithic version) that belong to the microservices. A high score in precision conveys that our tool correctly clusters methods together that are part of a microservice and does not wrongly assign methods that belong to other microservices. We compute precision as follows:

$$Precision = \frac{\text{Number of common\_elements}}{\text{Total number of elements in set(cluster)}} \tag{4.2.1}$$

- **Recall** measures the coverage and completeness of our microservices identification. It evaluates how completely our tool identifies all methods that should belong to a microservice. A high recall would mean that our tool covers all methods (from a cluster of the monolith) that are part of a specific microservice without missing any. We assess the recall by applying the following formula:

$$Recall = \frac{\text{Number of common\_elements}}{\text{Total number of elements in set(microservice)}} \tag{4.2.2}$$

In the equations for precision (4.2.1) and recall (4.2.2), the number of common elements denotes the intersection between a cluster set (obtained from the monolithic architecture) and a microservice set. Precision and recall are calculated for every matched pair of cluster-microservice .

4.2.3.2. Results and Discussion.

We obtained high precision scores for the per-cluster analysis ranging from 78.6% to 87% and low recall scores (7% - 17.7%). Table 4.4 summarizes the precision and recall scores obtained for each matching pair, {*cluster*, *microservice*}. We discuss some points considered in our approach and potential causes that could explain the low recall scores we obtained.

Clusters (Monolith)	cluster 6	cluster 4	cluster 1	cluster 3	cluster 7	cluster 2	cluster 5
Microservices	admin server	api gateway	config server	customers service	discovery server	vets service	visits service
Precision	78.6%	81.9%	79.6%	87%	79.6%	84.3%	82.1%
Recall	16.9%	12.6%	7%	17.3%	7%	17.7%	17.3%

Table 4.4. Results - Precision and recall

- **Our ground truth, PetClinic, does not represent a direct migration.**
  The methods used in the monolithic version of PetClinic are not automatically mapped when transitioning to its microservices version. Instead, the content of the microservices version is built with reference to its monolithic version. In fact, it does not directly implement the methods based on their lexical format.
  Given this context, it is both correct and expected that we observe low recall scores, as not all methods are retrievable in the microservices version.
  In the future, we plan to develop an additional computation metric better suited to our context and the available ground truths. This work will help us further evaluate our approach.

- **Two methods have the same method name but different functionalities.**
  In our approach, we treated the same method name occurring in different classes and packages within the monolithic application as distinct nodes. Beyond the method name, the class and package association help define more accurately the boundaries of these independent services. These methods can have the same name but serve different functionalities depending on their class and package association. This explains why we might see the same method name across multiple clusters of the

monolithic application after using our tool.

- **Disparity in unique method counts: Individual vs. collective set of microservices**
  In the distributed version of the ground truth, several methods can be found multiple times across the different microservices. This led to a significant difference in the total count of unique methods within each microservice compared to the aggregate count of unique methods across all microservices collectively. An example of such recurring methods includes those from the Java and Javax packages, which are commonly used for utility purposes.
  This impacted the recall scores obtained when comparing the extracted clusters with the microservices. These methods are essential to the integrity of the application's static dependencies but do not directly define the business logic.
  One solution we propose would be to assign all these methods to a separate and dedicated ***Utility*** cluster. This would allow the analysis to remain comprehensive while focusing significantly more on the business logic. Adding the ***Utility*** cluster could lead to an improvement in our recall scores.

- **A few methods in the microservices version of our ground truth cannot be retrieved in the monolithic version.**
  After analyzing our ground truth datasets, we identify some methods exclusive to the monolithic version, as well as other methods that are unique to the microservices version.
  These methods can be attributed to how the monolithic version handles data access compared to how it is done in the distributed version of the application.
  Additionally, some methods can be split or changed to better suit the specific architecture. For example, in the microservices version of our ground truth, some methods may have been split into smaller and more modular pieces to better fit the microservices architecture. This could explain why the microservices version has a higher method count than the monolithic version, as shown in Table 4.5.

PetClinic	All Methods count	Unique Methods Count
Monolithic version	750	750
Distributed version	4483	**821**

Table 4.5. Differences in the method count across the microservices

It is important to note that following the identification of distinct clusters of methods as potential microservices, a subsequent step might be required to package these clusters into microservices. This process could involve the clusters' encapsulation into deployable units to ensure interconnectivity through light communication. We made our implementation and evaluation experiments available on a GitHub repository [**6**] .

4.2.3.3. Comparison to other approaches.

Our approach is implemented at a fine-grained level, specifically at the method level, in contrast to most approaches in the literature, which focus on the class level. This difference in level of granularity can complicate or prevent direct comparisons and may lead to potential inaccuracies with metrics used for evaluating microservices at the class level.

Moreover, there is an overlap between the objective functions of our approach and the metrics used for evaluating and comparing other methodologies. Our approach uses the genetic algorithm NSGA-III, which requires us to define objective functions first before proceeding to the optimization of the solution search. We set three key objectives: two structural metrics (coupling and cohesion), and one semantic metric (semantic distance). In contrast, most other approaches in the literature do not incorporate these structural metrics in their methodology design but rather include them in their evaluations.

Therefore, it would not be suitable to compare our approach with the other approaches using these metrics to evaluate coupling and cohesion [**49**] such as CoHesion at Message level (CHM) and CoHesion at Domain level (CHD) [**28, 39, 57, 62**].

4.3. Threats to validity

4.3.1. Benchmark applications

We assessed our approach using a limited set of applications, which may not fully represent the diversity of all monolithic applications. To provide a more comprehensive analysis, we aim to use a more exhaustive set of legacy applications featuring a monolithic architecture. Furthermore, exploring a wider and more diverse range of applications beyond well-documented open-source applications can be done to ensure more inclusive and representative findings.

4.3.2. Programming language and framework

The existing applications we used, which feature a monolithic architecture, are exclusively Java projects. This limitation may impact the generalization of our approach to monolithic applications developed in other programming languages and frameworks. Additional evaluation on applications that are not using Java can help validate our results and improve generalization.

4.3.3. Level of granularity

Our approach operated at the method level of monolithic applications, differing in the level of granularity from our ground truth and some other approaches in the literature. This difference in granularity may have posed challenges when comparing our results directly with the ground truth.

4.4. Conclusion

In this chapter, we evaluated our proposed approach, first using two benchmarks to gain insights from our optimization and then comparing the generated clusters against the ground truth.

The first part of the evaluation explored the relationship between the scale of a monolithic application and the optimal number of partitions, leveraging the objectives' optimization results for coupling, cohesion, and semantic distance.

Additionally, we performed a comparative analysis between the generated clusters from our approach and the defined microservices representing the ground truth. This analysis yielded high precision scores, with the highest reaching 87%, indicating the effectiveness of our approach in identifying relevant methods for specific microservices. We also noted low recall scores ranging from 7% to 17.7%, which we addressed in a discussion on the underlying causes and suggested potential fixes and improvements.

Chapter 5

Conclusion

5.1. Summary

In this thesis, we have introduced an approach to identify microservices in existing applications through the combination of meta-heuristic techniques and machine learning. Our approach leverages the optimization capabilities of NSGA-III, a state-of-the-art multi-objective genetic algorithm, and the efficiency of SBERT semantic analysis, while also considering structural dependencies. We define three objectives to guide the search for solutions in our approach: minimizing coupling, maximizing cohesion, and maximizing semantic distance.

When handling monolithic applications where the number of functionalities is not explicitly showcased, we can use the scale (i.e., a combination of the class and method count) of the application and the values of the optimization metrics obtained from our approach to guide the search for an optimal number of clusters to represent the microservices structure.

In terms of results, our approach presents high precision and low recall scores when compared to the ground truth. The results for the identification of microservices at the method level are 78.6% - 87% for precision and 7% - 17.7% for recall. However, considering a utility cluster where all utility methods (e.g., methods from the Java standard library) are grouped could reduce the repetition of methods across the collective set of microservices; thus, it could be interesting to consider this point as one way to mitigate the low recall scores obtained.

Our approach and tool can offer valuable insights for the identification of microservices in existing applications while building a strong foundation for decomposing these monolithic applications into a distributed microservices version.

5.2. Limitations and Future Works

While our approach benefits from the diversity of solutions—offered by the genetic algorithm NSGA-III—and the analytical power of language models, there are some limitations that future research can address to advance our approach further.

The optimization of the semantic analysis can present an interesting avenue for improvement. Enhancing our approach could involve extending the scope of the semantic analysis to not only method signatures but also to the body of the method for potentially deeper insights.

Moreover, expanding our search criteria by incorporating additional objective functions and constraints can enrich the refinement of our approach. These additional objectives and constraints would enable tailoring the search to more specific goals, or domain-specific requirements. This could be facilitated by the use of NSGA-III in our approach, which has been applied in optimization problems with a large number of objectives [**20, 36**].

Furthermore, exploring the different database connections and interactions in monolithic applications could help uncover a more comprehensive understanding and representation of microservices boundaries [**44**].

Bibliography

[1] Felipe ALMEIDA et Geraldo XEXÉO : Word embeddings: A survey. *arXiv preprint arXiv:1901.09069*, 2019.

[2] P. ANDRITSOS et V. TZERPOS : Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, 31(2):150–165, 2005.

[3] Imam ASROWARDI, Septafiansyah Dwi PUTRA et Eko SUBYANTORO : Designing microservice architectures for scalability and reliability in e-commerce. *Journal of Physics: Conference Series*, 1450, 2020.

[4] ATLASSIAN : Microservices vs. monolithic architecture | atlassian. `https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith`, 2023. Accessed: 2023-12-04.

[5] Ravindra B BAPAT : *Graphs and matrices*, volume 27. Springer, 2010.

[6] Hanifa BARRY : Mic_id. `https://github.com/roxyHan/Mic_ID.git`, 2024. GitHub repository.

[7] Len BASS, Paul CLEMENTS et Rick KAZMAN : *Software architecture in practice*. Addison-Wesley Professional, 2003.

[8] J. BLANK et K. DEB : pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.

[9] Julian BLANK, Kalyanmoy DEB et Proteek Chandan ROY : Investigating the normalization procedure of nsga-iii. *In International Conference on Evolutionary Multi-Criterion Optimization*, pages 229–240. Springer, 2019.

[10] André B BONDI : Characteristics of scalability and their impact on performance. *In Proceedings of the 2nd international workshop on Software and performance*, pages 195–203, 2000.

[11] Ivan CANDELA, Gabriele BAVOTA, Barbara RUSSO et Rocco OLIVETO : Using cohesion and coupling for software remodularization: Is it enough? *ACM Trans. Softw. Eng. Methodol.*, 25(3), jun 2016.

[12] Pranava CHAUDHARI, Amit K THAKUR, Rahul KUMAR, Nilanjana BANERJEE et Amit KUMAR : Comparison of nsga-iii with nsga-ii for multi objective optimization of adiabatic styrene reactor. *Materials Today: Proceedings*, 57:1509–1514, 2022.

[13] Istehad CHOWDHURY et Mohammad ZULKERNINE : Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[14] Oracle CORPORATION : Jar file overview. `https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html`, 2024. Accessed: 2024-01-12.

[15] Indraneel DAS et John E DENNIS : Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM journal on optimization*, 8(3): 631–657, 1998.

[16] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap et T. Meyarivan : A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Trans. Evol. Comput.*, 6:182–197, 2002.

[17] Kalyanmoy Deb et Himanshu Jain : An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. *IEEE transactions on evolutionary computation*, 18(4):577–601, 2013.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee et Kristina Toutanova : Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[19] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin et Larisa Safina : Microservices: How to make your application scale. *In Perspectives of System Informatics: 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers 11*, pages 95–104. Springer, 2018.

[20] Jonathan E Fieldsend : University staff teaching allocation: formulating and optimising a many-objective problem. *In Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1097–1104, 2017.

[21] Sara E Garza et Satu Elisa Schaeffer : Community detection with the label propagation algorithm: a survey. *Physica A: Statistical Mechanics and its Applications*, 534:122058, 2019.

[22] L. Giamattei, A. Guerriero, R. Pietrantuono et S. Russo : Automated grey-box testing of microservice architectures. *In 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 640–650, 2022.

[23] Jean-Philippe Gouigoux et Dalila Tamzalit : "functional-first" recommendations for beneficial microservices migration and integration lessons learned from an industrial experience. *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 182–186, 2019.

[24] Sara Hassan, Rami Bahsoon et Rajkumar Buyya : Systematic scalability analysis for microservices granularity adaptation design decisions. *Software: Practice and Experience*, 52(6):1378–1401, 2022.

[25] Wilhelm Hasselbring et Guido Steinacker : Microservice architectures for scalability, agility and reliability in e-commerce. *In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246. IEEE, 2017.

[26] Ayaz Isazadeh, Islam Elgedawy, Jaber Karimpour et Habib Izadkhah : An analytical security model for existing software systems. *Applied Mathematics & Information Sciences*, 8(2), 2014.

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

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

[29] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic et Debasish Banerjee : Mono2micro: a practical and effective tool for decomposing monolithic java applications to microservices. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[30] Sourabh Katoch, Sumit Singh Chauhan et Vijay Kumar : A review on genetic algorithm: past, present, and future. *Multimedia tools and applications*, 80:8091–8126, 2021.

[31] Harveen KAUR : Fake news detection using semantic analysis and machine learning techniques. *In 2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–6, 2023.

[32] Diksha KHURANA, Aditya KOLI, Kiran KHATTER et Sukhdev SINGH : Natural language processing: State of the art, current trends and challenges. *Multimedia tools and applications*, 82(3):3713–3744, 2023.

[33] Abdullah KONAK, David W COIT et Alice E SMITH : Multi-objective optimization using genetic algorithms: A tutorial. *Reliability engineering & system safety*, 91(9):992–1007, 2006.

[34] Patrick LAM, Eric BODDEN, Ondrej LHOTÁK et Laurie HENDREN : The soot framework for java program analysis: a retrospective. *In Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, 2011.

[35] Xabier LARRUCEA, Izaskun SANTAMARIA, Ricardo COLOMO-PALACIOS et Christof EBERT : Microservices. *IEEE Software*, 35(3):96–100, 2018.

[36] Oskar MARKO, Dejan PAVLOVIĆ, Vladimir CRNOJEVIĆ et Kalyanmoy DEB : Optimisation of crop configuration using nsga-iii with categorical genetic operators. *In Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 223–224, 2019.

[37] Robert Cecil MARTIN : Agile software development: principles, patterns, and practices. *Computing Reviews*, 46(2):91, 2005.

[38] V.Valli MAYIL et T.Ratha JEYALAKSHMI : Pretrained sentence embedding and semantic sentence similarity language model for text classification in nlp. *In 2023 3rd International conference on Artificial Intelligence and Signal Processing (AISP)*, pages 1–5, 2023.

[39] Genc MAZLAMI, Jürgen CITO et Philipp LEITNER : Extraction of microservices from monolithic software architectures. *In 2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531, 2017.

[40] Kaisa MIETTINEN : *Nonlinear multiobjective optimization*, volume 12. Springer Science & Business Media, 1999.

[41] Tomas MIKOLOV, Kai CHEN, Greg CORRADO et Jeffrey DEAN : Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[42] Mohamed Wiem MKAOUER, Marouane KESSENTINI, Slim BECHIKH, Kalyanmoy DEB et Mel Ó CINNÉIDE : High dimensional search-based software engineering: Finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii. *In Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, page 1263–1270, New York, NY, USA, 2014. Association for Computing Machinery.

[43] Zewdie MOSSIE et Jenq-Haur WANG : Vulnerable community identification using hate speech detection on social media. *Information Processing & Management*, 57(3):102087, 2020.

[44] Antonios MPARMPOUTIS et George KAKARONTZAS : Using database schemas of legacy applications for microservices identification: A mapping study. *In Proceedings of the 6th International Conference on Algorithms, Computing and Systems*, pages 1–7, 2022.

[45] Gunter MUSSBACHER, Benoit COMBEMALE, Jörg KIENZLE, Lola BURGUEÑO, Antonio GARCIA-DOMINGUEZ, Jean-Marc JÉZÉQUEL, Gwendal JOUNEAUX, Djamel-Eddine KHELLADI, Sébastien MOSSER, Corinne PULGAR *et al.* : Polyglot software development: Wait, what? *IEEE Software*, 2024.

[46] Gastón Márquez et Hernán Astudillo : Actual use of architectural patterns in microservices-based open source projects. *In 2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 31–40, 2018.

[47] P. Ngatchou, A. Zarei et A. El-Sharkawi : Pareto multi objective optimization. *In Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*, pages 84–91, 2005.

[48] Patrick Niemeyer et Daniel Leuck : War files and deployment - learning java, 4th edition [book]. `https://www.oreilly.com/library/view/learning-java-4th/9781449372477/ch15s03.html`, 2024. Accessed: 2024-01-12.

[49] Vikram Nitin, Shubhi Asthana, Baishakhi Ray et Rahul Krishna : Cargo: ai-guided dependency analysis for migrating monolithic applications to microservices architecture. *In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[50] B. Oscar Deho, A. William Agangiba, L. Felix Aryeh et A. Jeffery Ansah : Sentiment analysis with word embedding. *In 2018 IEEE 7th International Conference on Adaptive Science & Technology (ICAST)*, pages 1–4, 2018.

[51] Jeffrey Pennington, Richard Socher et Christopher D Manning : Glove: Global vectors for word representation. *In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[52] Dewayne E. Perry et Alexander L. Wolf : Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, oct 1992.

[53] Francisco Ponce, Gastón Márquez et Hernán Astudillo : Migrating from monolithic architecture to microservices: A rapid review. *In 2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7, 2019.

[54] Nils Reimers et Iryna Gurevych : Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[55] Chaitanya K Rudrabhatla : Impacts of decomposition techniques on performance and latency of microservices. *International Journal of Advanced Computer Science and Applications*, 11(8), 2020.

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

[57] Khaled Sellami, Ali Ouni, Mohamed Aymen Saied, Salah Bouktif et Mohamed Wiem Mkaouer : Improving microservices extraction using evolutionary search. *Information and Software Technology*, 151:106996, 2022.

[58] Juan P. Sotomayor, Sai Chaithra Allala, Patrick Alt, Justin Phillips, Tariq M. King et Peter J. Clarke : Comparison of runtime testing tools for microservices. *In 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 356–361, 2019.

[59] Kit-Sang Tang, Kim-Fung Man, Sam Kwong et Qianhua He : Genetic algorithms and their applications. *IEEE signal processing magazine*, 13(6):22–37, 1996.

[60] Nandan Thakur, Nils Reimers, Johannes Daxenberger et Iryna Gurevych : Augmented sbert: Data augmentation method for improving bi-encoders for pairwise sentence scoring tasks. *ArXiv*, abs/2010.08240, 2020.

[61] Johannes Thönes : Microservices. *IEEE Software*, 32(1):116–116, 2015.

[62] Imen TRABELSI, Manel ABDELLATIF, Abdalgader ABUBAKER, Naouel MOHA, Sébastien MOSSER, Samira EBRAHIMI-KAHOU et Yann-Gaël GUÉHÉNEUC : From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis. *Journal of Software: Evolution and Process*, page e2503, 2022.

[63] Peipei XIA, Li ZHANG et Fanzhang LI : Learning similarity with cosine similarity ensemble. *Information sciences*, 307:39–52, 2015.

[64] Tom YOUNG, Devamanyu HAZARIKA, Soujanya PORIA et Erik CAMBRIA : Recent trends in deep learning based natural language processing [review article]. *IEEE Computational Intelligence Magazine*, 13(3):55–75, 2018.

[65] Uwe ZDUN, Pierre-Jean QUEVAL, Georg SIMHANDL, Riccardo SCANDARIATO, Somik CHAKRAVARTY, Marjan JELIĆ et Aleksandar JOVANOVIĆ : Detection strategies for microservice security tactics. *IEEE Transactions on Dependable and Secure Computing*, pages 1–17, 2023.

[66] Guogen ZHANG, Kun REN, Jung-Sang AHN et Sami BEN-ROMDHANE : Grit: Consistent distributed transactions across polyglot microservices with multiple databases. *In 2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 2024–2027, 2019.

[67] Eckart ZITZLER et Simon KÜNZLI : Indicator-based selection in multiobjective search. *In Parallel Problem Solving from Nature*, 2004.

[68] Eckart ZITZLER, Marco LAUMANNS et Lothar THIELE : Spea2: Improving the strength pareto evolutionary algorithm. *TIK report*, 103, 2001.