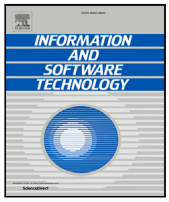
Information and Software Technology 151 (2022) 106996

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



# Improving microservices extraction using evolutionary search

Khaled Sellami [a], Ali Ouni [b], Mohamed Aymen Saied [a,*], Salah Bouktif [d], Mohamed Wiem Mkaouer [c]

[a] Laval University, Quebec, QC, Canada
[b] ETS Montreal, University of Quebec, Montreal, QC, Canada
[c] Rochester Institute of Technology, Rochester, NY, USA
[d] College of Information Technology, UAE University, United Arab Emirates

ARTICLE INFO

*Keywords:*
Microservices
Search-based software engineering
Legacy decomposition
Microservices architecture

ABSTRACT

**Context:** Microservices constitute a modern style of building software applications as collections of small, cohesive, and loosely coupled services, *i.e.*, modules, that are developed, deployed, and scaled independently.
**Objective:** The migration from legacy systems towards the microservice-based architecture is not a trivial task. It is still manual, time-consuming, error-prone and subsequently costly. The most critical and challenging issue is the cost-effective identification of microservices boundaries that ensure adequate granularity and cohesiveness.
**Method:** To address this problem, we introduce in this paper a novel approach, named *MSExtractor*, that formulates microservices identification as a multi-objective optimization problem. The proposed solution aims at decomposing a legacy application into a set of cohesive, loosely-coupled and coarse-grained services. We employ the Indicator-Based Evolutionary Algorithm (IBEA) to drive a search process towards optimal microservices identification while considering structural and semantic dependencies in the source code.
**Results:** We conduct an empirical evaluation on a benchmark of seven software systems to assess the efficiency of our approach. Results show that *MSExtractor* is able to carry out an effective identification of relevant microservice candidates and outperforms three other existing approaches.
**Conclusion:** In this paper, we show that MSExtractor is able to extract cohesive and loosely coupled services with higher performance than three other considered methods. However, we advocate that while automated microservices identification approaches are very helpful, the role of the human experts remains crucial to validate and calibrate the extracted microservices.

## 1. Introduction

With the wide adoption of modern software development technologies, software systems have moved from legacy applications to small, lightweight, single-purpose, and process-driven components called *microservices*. Microservices is a style of software architecture, which makes application development easier and offers scalable, highly flexible and easy-to-maintain applications [1,2].

Such architectural style has attracted many software companies such as eBay, Netflix and Airbnb, and the move towards microservices is well underway. However, practitioners are facing several challenges to migrate their legacy applications into microservices [3,4]. Indeed, today's organizations are undertaking serious efforts for manually moving from their current legacy architectures to microservices-oriented application landscapes [5].

One major challenge, in the current migration attempts, is the distinction of components, from the legacy software architecture, which are cohesive enough to be clustered into one standalone service, with adequate granularity [6,7].

This process is by nature subjective, manual and error-prone. It needs a multi-dimensional analysis that involves multiple factors reflecting software quality, software design, microservices functionalities, etc [8]. Such a process heavily relies on expert opinions. Indeed the expertise of software developers plays the main roles and decisions in the task of microservices extraction [9] as well as in the preservation of the original system quality. Indeed, C. Richardson [10] pointed out that:

> *"Deciding how to partition a system into a set of services is very much an art but there are number of strategies that can help".*

* Corresponding author.
*E-mail addresses:* khaled.sellami.1@ulaval.ca (K. Sellami), ali.ouni@etsmtl.ca (A. Ouni), mohamed-aymen.saied@ift.ulaval.ca (M.A. Saied), salahb@uaeu.ac.ae (S. Bouktif), mwmvse@rit.edu (M.W. Mkaouer).

https://doi.org/10.1016/j.infsof.2022.106996
Received 12 November 2021; Received in revised form 21 June 2022; Accepted 29 June 2022
Available online 5 July 2022
0950-5849/

In particular, there is no silver bullet to establish microservices boundaries, which make microservices extraction more complex and challenging. Indeed, in a migration process, the first and hardest task is to identify the microservices boundaries to successfully decompose a legacy application. If microservices boundaries are not properly identified, the migration could lead to a more complex new application, and thus a degradation of the initial quality.

Research in the field of automated microservices extraction is still in its infancy, and there are still no a cost-effective and automatic techniques available for microservices extraction. [11–15]. Most of the existing approaches have formulated the microservices identification as a clustering problem based on the structural and/or semantic similarity of foreseen functionality described through the applications API specifications [16], dynamic analysis [12], coupling-based graph clustering [9,17,18], or feature modularization and network overhead-based identification [15] with the ultimate goal to find functionally related classes, *i.e.*, candidate microservices. However, most of these approaches rely on ad-hoc criteria that fail to identify relevant microservices. First, finding the optimal number of candidate microservices with the appropriate granularity and loose coupling is still their major limitation. Second, classes in a legacy system can have different purposes such as classes intended for internal (*i.e.*, inner) or external (*i.e.*, interface) use [19,20] making services extraction more challenging since legacy systems are not generally designed with the vision of service. Furthermore, finding a trade-off between granularity, coupling and cohesion within the extracted microservices limits their performance and practicality. Indeed, decomposing a software system into smaller components always has been a challenge in software engineering, and known as complex problems which are best suited to search-based software engineering (SBSE) [21–31].

In this paper, we found the current contribution on our previous work published in the *17th International Conference on Service-Oriented Computing (ICSOC)* [11]. In particular, we extend our approach, i.e., *MSExtractor*, that formulates the microservices extraction as a multi-objective combinatorial optimization problem to decompose an OO legacy application into a set of cohesive, loosely-coupled microservices. We employ the Indicator-Based Evolutionary Algorithm (IBEA) [32], as a search method to lead the decomposition process in finding the near-optimal trade-off solution considering three objectives (1) microservices granularity, (2) minimize coupling (inter-service dependencies), and (3) maximize cohesion (intra-services dependencies) while leveraging the structural and semantic information embodied in the source code. To better capture the boundaries of the extracted microservices, we distinguish between (*i*) classes that are intended to expose the microservice functionalities to the outer world (*i.e.*, other microservices, external clients, etc.) and (*ii*) classes for internal use within the microservice. In particular, we employ the concept of *inner* classes and *interface* classes [20] to distinguish between classes intended for internal use in a microservice and others that serve as parts of the microservice interface, *i.e.*, to communicate with other microservices. MSExtractor aims at supporting software developers and architects by providing a decision-making support in their design decisions for their microservices migration.

To evaluate our approach, we conducted a set of experiments. First, we assess the efficiency of our approach in identifying candidate microservices on a benchmark of four open source Java legacy applications. The obtained results show that MSExtractor is able to extract cohesive and loosely coupled microservices with higher performance than three other existing approaches. Moreover, we evaluate the performance of IBEA compared to two other widely-used algorithms, the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [33], and Strength Pareto Evolutionary Algorithm 2 (SPEA2) [34]. Statistical results show that IBEA is the most efficient and thus represents a better choice for this problem at hand. Furthermore, we evaluate the efficiency of MSExtractor in retrieving microservices on a benchmark of three SpringBoot projects originally designed according to the microservices paradigm by human experts. The obtained results show that MSExtractor was able to identify a large number of the correct placement of the classes with regards to the original microservices to which they belong. However, we also identified corner cases in which MSExtractor was not successful in retrieving the microservices.

It is worth to remind that among the contributions of this paper, we extend our previous conference-published work [11] in the following ways:

- We extended our problem formulation by introducing a new fitness function that aims at controlling the granularity of the extracted microservices. We also adopt a new algorithm, IBEA [32] which has shown better performance with the extended problem formulation than NSGA-II [33] and other multi-objective algorithms.
- To better capture the boundaries of the extracted microservices, we distinguish between classes that are intended to expose the microservice functionalities to the outer world (*i.e.*, other microservices, external clients, etc.) and other classes for internal use. In particular, we incorporate the concept of *inner* classes and *interface* classes [19,20] for an effective identification of candidate micro-services. However, for minimizing coupling, the inner classes will not be considered as they do not externally expose functionalities provided by a candidate microservice.
- We extend our experimental study to compare our approach with different existing approaches on four legacy systems. We also assess the efficiency of our approach in identifying microservices as compared to developers design on three microservice-based open source software systems.

The rest of the paper is organized as follows. Section 2 provides background on microservices and the work related to our study. We describe our approach using IBEA in Section 3. We present our empirical evaluation of our proposed approach in Section 4. The threats to validity are discussed in Section 6. Finally, Section 7 concludes the paper and outlines our future work.

## 2. Background and related work

This section provides the necessary background and relevant related work.

### 2.1. Definitions

**Microservices architecture.** The most widely adopted definition of a microservices architecture is "*an approach for developing a single application as a collection of small services, each running in its own process and communicating with lightweight mechanisms, often a RESTful API*" [35]. Microservices can be developed using different technology stacks, and can address the drawbacks of legacy applications. By being small, a microservice can restart faster at the time of upgrade or failure recovery. Microservices are also loosely coupled, and failure of one microservice will not deeply affect other microservices of the system. Moreover, the fine-grained architecture makes scaling flexible as each service can evolve at its own workload pace [16,36–38].

**Microservice.** Microservices are a realization of Service-Oriented Architecture (SOA) principles that are better suited for agile software development. The microservices technology constitutes the next generation for designing scalable, easy-to-maintain applications [1,39,40] achieved by creating small, distributed single-purpose services. Each service has its own functionality and data, along with being supported by a fully automated deployment machinery, with minimum centralized management. Each microservice is designed to be independent of all other services, exposing only a well-defined interface [1,2,41].

**Microservices migration process.** A microservice migration, also termed as "microservitization" in [42]. It consists primarily of decomposing legacy applications into services that are cheap to evolve

and easy to throw away. The guiding theme behind this movement is to decentralize change management and reduce conflicts that tend to cause roadblocks in an SOA-based platform. The migration process towards microservices architecture consists of a number of following key steps [43]:

(1) Identification of microservices boundaries,
(2) Prioritizing microservices for migration,
(3) Handling data synchronization during the transition phase,
(4) Handling user interface integration, working with old and new user interfaces,
(5) Handling of reference data in the new system,
(6) Testing strategy to ensure the business capabilities are intact and correctly reproduced,
(7) Identification of any prerequisites for microservice development such as microservices capabilities, frameworks, processes, and so on.

The first step, *i.e.*, the identification of microservices boundaries constitute the scope of the present work. In fact, the challenging task to run a migration is correctly identifying potential microservices in the legacy software, *i.e.*, the partitioning process.

### 2.2. Related work

A number of approaches have been proposed in the literature for software decomposition/modularization and services extraction. Most of these approaches can be distinguished by three main axes. The first difference lies within the input used in their approach with source code being the most common one. The second main difference is the process used to transform and handle the input. This phase is usually dependent on the input and is usually categorized into static, dynamic or semantic analysis. The final difference lies within the algorithm that uses the result of the previous phase to identify the potential microservices with clustering algorithms being the most common. Jin et al. [12] focused on an approach, called *FoME*, based on analyzing the execution traces of the original monolithic application in order to generate the candidate microservices. They feed the execution traces to clustering algorithms which group together the functionally dependent entities. While the proposed approach provides interesting results in terms of microservices identification, they heavily rely on the used technologies in the industrial case study. Hence, their adaptation to other contexts might not be straightforward. In addition, since their dynamic analysis relies exclusively on the execution traces, their approach is limited to specific test case scenarios and as such excludes static dependencies and any component that is not covered by the traces. Carvalho et al. [14] proposed an approach named *toMicroservices*, that aims at optimizing various measures such as cohesion and coupling to feature modularization, reuse and network overhead. The proposed method was tailored for an industrial case study and showed that the extracted microservices fit with the developers' preferences. To calculate these measures, *toMicroservices* requires as input both the source code of the project as well as labeled use cases. A Multi-Objective Evolutionary Algorithm based on these measures was used to generate the decompositions. This research was extended by Assuncao et al. [15] who focused on an industrial case study and proposed a semi-manual approach using the measures of the previous paper. Omar et al. [44] proposed an approach called *Code2vec* which transforms the source code of the legacy application into Abstract Syntax Trees which are then fed as input to a code embeddings model. The output of this model is used in conjunction with a clustering algorithm to generate the decompositions. Anup et al. [45] based their approach, *Mono2micro*, on analyzing labeled execution traces and a customized hierarchical clustering algorithm. The solution *CO-GCN* proposed by Desai et al. [46] similarly used execution traces in addition to the source code in order to build and train their Graph Convolution Neural model which is used for clustering the legacy application. Other approaches focused on taking advantage of the semantic relationships within the code and API of the legacy projects to generate the decompositions. For example, Brito et al. [47] provided a decomposition method that can be generalized across all legacy software systems. The proposed method uses topic modeling and clustering techniques in order to create microservices with similar domain terms taking in as input the source code of the legacy applications. Baresi et al. [16] leverage a reference vocabulary and semantic similarity of foreseen functionality described through OpenAPI specifications based. They generate a semantic mapping using the application's API and the OpenAPI specifications and then use a clustering algorithm to group semantically similar APIs. However, this approach excludes any structural dependencies such as cohesion and coupling.

On the other hand, some approaches utilize unique types of input. Mazlami et al. [9] proposed an extraction approach called *MEM* based on three formal coupling strategies (logical, semantic, and contributor couplings) to construct a graph-based clustering algorithm to find potential microservices. This approach relies on the git commit history of the projects as well as their source code to generate the graphs. Gysel et al. [17] proposed a service decomposition approach, namely *ServiceCutter*, based on 16 different coupling criteria represented as an undirected, weighted graph to find densely connected clusters through graph cutting. Similarly, this approach uses a clustering algorithm for the decomposition but takes in as input the design documents and other artifacts of the design and analysis phases of the input project. However, both of these approaches rely on a type of input that is rarely available in the legacy software. One of the major challenges when working with legacy applications is that their design documents are often outdated and do not represent the current state of the application.

Besides microservices extraction, several studies concentrated on software remodularization/decomposition. *LIMBO* [48], for instance, proposes an algorithm that decomposes a large software into smaller clusters that can be used by developers to have a better understanding of the system at hand. This task is achieved by using information loss to measure the distance between classes and by creating a custom clustering algorithm that can group together entities with similar structural dependencies. Given the similarity of this problem to ours, the proposed approach can be adapted to microservices decomposition. Mkaouer et al. [25] use a search-based approach through semantic, logical and structural dependency for software packages remodularization. Harman et al. [22] proposed a multi-objective software module clustering approach, in which several different objectives (including cohesion and coupling) are optimized separately. Shatnawi et al. [49] propose an approach that aims to identify reusable software components based on the dynamic analysis of interactions between client applications and the targeted API.

More recently, Freitas et al. [50] proposed a method that takes as input the list and composition of a targeted microservices architecture and the source code of the legacy Java application to generate a microservices-based application. This research tackles the problem of refactoring Java Spring applications into microservices compatible applications rather than focusing on the identification of the microservices. Their method analyzes the source code and the proposed microservices decomposition and generates REST calls between methods and classes that are part of different microservices.

While there is a large body of prior works on traditional software remodularization, they are formulated for hierarchical package clustering and not designed to capture microservices boundaries. On the other hand, recent studies on microservices extraction deal with the problem using traditional clustering methods and specific metrics which shows their limitations in solving combinatorial problems. This research gap is tackled by this paper mainly through the adoption of a search-based approach that leverages the use of multiple cohesion, coupling and granularity criteria of modularization while distinguishing between

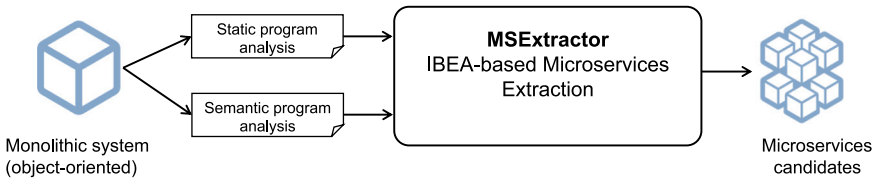



**Fig. 1.** MSExtractor: Approach Overview.

inner and interface classes in a microservice. Given that the source code is most often available when trying to decompose legacy software, we use it as input in order to provide a solution that can be used in most settings. In addition, since dynamic analysis often excludes a large part of the components, our approach combines static and semantic analysis in order to take advantage of both methods.

## 3. Approach

In this section, we provide an overview of the proposed approach, MSExtractor, then we present the problem formulation and the algorithm adaptation.

Decomposing software systems into loosely coupled, highly cohesive and distributed units is a challenging task for software developers. Service-oriented architectures and their microservices deployments tackle many related problems, but remain vague on how to cut a system into independent, discrete, and autonomous services. Theoretically, every possible set of classes where each microservice should be contained in one and only one microservice. This is described as a set-partitioning problem.

**Problem complexity.** Finding the best partitioning of classes into cohesive coarse-grained microservices is not a trivial task for software architects/developers as the number of possible class combinations can be very large causing a combinatorial explosion. The search space tends to be enormous as the number of possible partitions is given by:

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k \tag{1}$$

where $B_n$ counts the number of different possibilities of how a given set of $n$ classes can be divided into $k$ microservices. The order of the partitions, *i.e.*microservices, as well as the order of the classes within a microservice do not need to be considered. Such problems are well suited to Search-based Software Engineering (SBSE) [21,30,51].

### 3.1. Approach overview

We formulate the automated extraction of microservices from a legacy application as a combinatorial optimization problem, where a search algorithm will explore and evaluate alternative combinations of classes from an input legacy system. Given legacy system formed by a set of classes to be decomposed into microservices, there are many ways in which the microservice boundaries can be drawn leading to a number of possible class combinations. This decomposition is equivalent to a graph partitioning problem, which is known to be complex and therefore well suited to be solved using meta-heuristic searches [22,25].

Fig. 1 provides an overview of the MSExtractor approach to the microservices identification problem. MSExtractor aims at exploring a large search space to find a set of optimal microservices candidates, by grouping classes with high cohesion into separate microservices.

To identify such instances of candidate microservices, MSExtractor proceeds with (*i*) creating a set of new empty microservices, and (*ii*) assigning each class to a unique microservice. The process should assign each class to exactly one microservice, not allowing empty microservices. Then, MSExtractor uses IBEA in order to find the optimal solutions that provide the best trade-off between our three objective functions which are cohesion, coupling and granularity that measure respectively the cohesion of classes within a microservices, the coupling between different microservices and the distribution of classes. In the following, we define our problem formulation and provide the details on our IBEA adaptation to extract candidate microservices.

| 1 | 1 | 3 | 2 | 2 | 3 | 1 |
|---|---|---|---|---|---|---|
| InitFilter | IPBanFilter | CalendarTag | FileContent | MediaFile | CalendarModel | User |

**Fig. 2.** An example of a microservice decomposition solution (snippet) from `JPetstore`.

### 3.2. Problem formulation

We define a microservices architecture $\mathcal{M}$ as a decomposition of the set of classes $C$ into a set of microservices $m$, where each microservice represents a single unit in the form of a container of classes. We define the microservice size, *size(m)*, by the number of its identified classes.

Consider a legacy software system with set of classes $C = \{c_1, c_2, \ldots, c_n\}$ where $n$ is the number of classes in the system. The set of possible modules, i.e., microservices, is represented by $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$ where $k$ is the number of microservices, and each microservice has its unique number $1, 2, \ldots, k$. A possible decomposition solution for this problem is defined by the decision variables $X = \{x_1, x_2, \ldots, x_n\}$, where $x_i = m_i$ indicates that $c_i$ belongs to the microservice $m_i$.

Fig. 2 shows a simple microservices decomposition example. A simple solution $X = \{1, 1, 2, 3, 1, 1, 2\}$, for example, denotes a decomposition of seven classes into three microservices. The classes `InitFilter`, `IPBanFilter` and `User` are in the microservice $m_1$, `Product`, `CarItem` and `Category` are in $m_2$, and finally, `Order` and `Catalog` are in $m_3$. Moreover, different class dependencies exist in order to implement the required functionalities by the microservice. An appropriate decomposition should maximize the cohesion within a microservice while minimizing coupling between the extracted microservices. We adapted these quality metrics to the contexts of microservices architecture We considered that a microservice is composed of two types of classes. The first type is Inner classes that only have internal connections to other classes of the same microservice. Whereas the second type of classes define the functionality provided by the microservice to other microservices, these are the classes that define the microservice interface. We encode the classes types as follow $T = \{t_1, t_2, \ldots, t_n\}$, where $t_i = -1$ indicates that the class $c_i$ is an *inner class*, and $t_i = -0$ indicates that $c_i$ is an *interface class* within the considered decomposition. To maximize the cohesion within a microservice, we consider both the inner and the interface classes. On the other side, to achieve a more precise coupling measurement, the inner classes will not be considered as they do not expose functionalities provided by the microservice. Inner classes only interact with those within the same microservice. Thus, by construction, it is more efficient to exclude them and only include interface classes when calculating the coupling

Source code dependencies are widely used in software engineering to measure how strongly related are the elements of a software system, *i.e.*methods, classes, packages, APIs, libraries etc. [52–54]. MSExtractor is based on a combination of *structural* and *semantic* similarity measures to detect the dependencies among classes.

***A. Structural similarity*** ($Sim^{STR}$). We define the structural relatedness (*i.e.*, dependency) [55] between two classes in terms of their shared method calls. Let $calls(c_i, c_j)$ be the number of calls performed by the methods of the class $c_i$ to the methods of $c_j$, and $calls_{in}(c_i)$ be the total number of incoming calls to $c_i$. Formally, the structural similarity $Sim^{STR}$ between $c_i$ and $c_j$ is defined as follows:

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

$Sim^{STR}(c_i, c_j)$ values are in the interval $[0, 1]$.

***B. Semantic similarity*** ($Sim^{SEM}$). The concept of bounded context originating in domain driven design (DDD) [56] is used as a basic design rationale for defining microservices boundaries [2,9,57]. That is, each microservice should match one single defined bounded context from the problem domain. Based on this concept of bounded context, we define the semantic similarity $Sim^{SEM}$ measure based on the hypothesis that the responsibilities of a software component are embodied in the domain vocabulary and terminology used to implement the concerned component [56,58–61]. Two classes are deemed to be semantically related if their domain semantics are similar, *i.e.*they perform related/similar actions. To measure $Sim^{SEM}$, a tf-idf (term frequency-inverse document frequency) model is used to represent each class as a vector that spans a space defined by the vocabulary extracted from the class content. The vocabulary of a class, as an important source of information [16,47,62–67], is extracted for its identifiers of fields, methods, variables, parameters, and comments. Each term goes through a pre-precessing step to apply camel case splitting, followed by filtering common stop words, and stemming. The semantic similarity $Sim^{SEM}$ between two classes $c_i$ and $c_j$ is then computed as the cosine of the angle between their corresponding vectors as follows :

$$Sim^{SEM}(c_i, c_j) = \frac{\vec{c_i} \cdot \vec{c_j}}{\|\vec{c_i}\| \times \|\vec{c_j}\|} \in [0, 1] \tag{3}$$

where $\vec{c_i}$ and $\vec{c_j}$ are the vectors corresponding to the classes $c_i$ and $c_j$, respectively, and $\|\vec{c_i}\|$ and $\|\vec{c_j}\|$ refer to the Euclidean norm of the vectors $\vec{c_i}$, $\vec{c_j}$.

Both similarity functions encode the dependencies between the classes but each approaches it in a different view. Since the objective functions are based on the dependencies regardless of how they were detected, each objective function will have two values for each similarity function. This approach will significantly increase the complexity of the search space. However, both of these measures encode a specific similarity and share the same input. For these reasons, we decided to merge the non-conflicting objective functions by defining a new similarity function as the weighted sum of both of these measures which could prove to be a simple yet effective compromise. In addition, the weights can be added as user-defined parameters that control the influence of each similarity.

We define the class similarity as a weighted sum between the structural and semantic similarity between classes, $Sim^{STR}$ and $Sim^{SEM}$, respectively. Class similarity $CS$ between two classes $c_i$ and $c_j$ is defined as follows :

$$CS(c_i, c_j) = \alpha \times Sim^{STR}(c_i, c_j) + \beta \times Sim^{SEM}(c_i, c_j) \in [0, 1] \tag{4}$$

where $\alpha + \beta = 1$, and their values reflect the confidence *i.e.* weight, in each similarity measure.

### 3.3. MOEA adaptation

To adapt a search algorithm, such as IBEA, to a specific problem, the following elements need to be defined: (1) the solution representation, (2) the fitness (or objective) function, and (3) the change operators.

#### 3.3.1. Solution representation

A candidate solution to the problem is a candidate legacy application decomposition into a set of candidate microservices, each comprises a set of classes. A valid solution assigns each class to exactly one microservice, and has no empty microservices. To represent individuals, we adopt the *label-based integer* encoding [68], where a candidate solution is encoded as an *integer array* of $n$ elements, where $n$ is the number of classes composing a microservice. Each element in the array corresponds to a specific class. The integer values in the array represent the microservice to which the classes belong. For instance, the encoded example in Fig. 2 shows an array, encoded as "12231132", that consists of eight classes assigned to three microservices.

#### 3.3.2. Initial population

The initial population is entirely random, where a max number of microservices $n$ is fixed, then each class in the legacy application is randomly assigned to a unique microservice. Furthermore, we define the parameter $minSize$ as the minimum number of classes per microservice.

#### 3.3.3. Objective functions

To evaluate the quality (*i.e.*, the fitness) of a candidate microservices decomposition solution, we define a fitness function that evaluates multiple objective and constraint dimensions. Each objective dimension refers to a specific value that should be either minimized or maximized for a solution to be considered "better" than another solution. In our approach, we optimize the three following objectives:

1. **Cohesion**: The cohesion objective function is a measure of the overall cohesion of a candidate microservices decomposition. The cohesion of a candidate microservice $m$ is denoted by $Coh(m)$ and defined as the complement of the average of all pairs of classes belonging to the microservice $m$. Formally, $Coh(m)$ is defined as follows:
$$Coh(m) = 1 - \frac{\sum_{\forall (c_i, c_j) \in m, c_i \neq c_j} CS(c_i, c_j)}{\frac{|m| \times (|m|-1)}{2}} \tag{5}$$
where $CS(c_i, c_j)$ denotes the class similarity between the two classes $c_i$ and $c_j$ as defined in Eq. (4).
Then, the cohesion objective function corresponds to the average cohesion value of all microservice candidates in a decomposition $\mathcal{M}$ and is computed as follows:
$$Cohesion(\mathcal{M}) = 1 - \frac{\sum_{\forall m_i \in \mathcal{M}} Coh(m_i)}{|\mathcal{M}|} \tag{6}$$
where $Coh(m_i)$ denotes the cohesion of the microservice $m_i$ given by Eq. (5), and $|\mathcal{M}|$ is the total number of microservices in the decomposition $\mathcal{M}$. This objective function should be maximized to ensure that each candidate microservice contains strongly related classes and does not contain classes that are not part of its functionality.
2. **Coupling**: The coupling objective function measures the overall coupling among the microservice in a decomposition $\mathcal{M}$. We define the coupling between two microservices $m_1$ and $m_2$ as the average similarity between all possible pairs of classes from $I_1$ and $I_2$, where $I_1$ and $I_2$ are the interfaces of $m_1$ and $m_2$ respectively. Formally, the coupling, $C$, is defined as follows:
$$C(m_1, m_2) = \frac{\sum_{\forall c_i \in m_1, \forall c_j \in m_2} CS(c_i, c_j)}{|m_1| \times |m_2|} \tag{7}$$

where $|\ m_1\ |$ denotes the number of classes in the interface $I_1$, and $CS(c_i, c_j)$ is defined as the weighted sum of the structural and semantic class similarity measures defined in Eq. (4).
The coupling objective function corresponds to the average coupling measures between all possible pairs of microservices in the decomposition $\mathcal{M}$ and is calculated as follows:

$$Coupling(\mathcal{M}) = \frac{\sum_{\forall (m_i, m_j) \in \mathcal{M},\ m_i \neq m_j} C(m_i, m_j)}{\frac{|\mathcal{M}| \times (|\mathcal{M}|-1)}{2}} \quad (8)$$

where $C(m_i, m_j)$ denotes the coupling between the microservices $m_i$ and $m_j$ given by Eq. (7), and $|\ \mathcal{M}\ |$ is the total number of microservices in the decomposition $\mathcal{M}$.
This objective function is to be minimized. The lower the coupling value between all candidate microservices, the better is the decomposition quality.

3. **Granularity**: The granularity of the decomposition $\mathcal{M}$ refers to the ratio of the original legacy systems size (in terms of number of classes $C$) by the total number of candidate microservices, and defined as follows :

$$Granularity(\mathcal{M}) = \frac{|\ C\ |}{|\ \mathcal{M}\ |} \quad (9)$$

This objective function is to be minimized by the algorithm. Our approach also allows the values of acceptable target granularity of candidate decomposition solutions to be configured by the user within a particular range.

To showcase the process used by our solution to calculate the fitness functions we consider the following example. Let us assume that we have a monolithic application with five classes. After the static analysis of the source code, we will measure both the structural similarity and the semantic similarity which we combine to create the following class similarity matrix

$$\begin{bmatrix} 1. & 0. & 0.75 & 0. & 0.75 \\ 0. & 1. & 0.75 & 0.5 & 0. \\ 0.75 & 0.75 & 1. & 0. & 0.25 \\ 0. & 0.5 & 0. & 1. & 0.75 \\ 0.75 & 0. & 0.25 & 0.75 & 1. \end{bmatrix}$$

After a few iterations in the IBEA algorithm, we find the following individual in the population: $I$ : "11122" which encodes a decomposition with two microservices. In order to evaluate this individual compared to the others in the current population, we will calculate the fitness functions:

- Cohesion: First we measure coh for each microservice so
  $coh(M1) = 1 - \frac{CS(c_1,c_2)+CS(c_1,c_3)+CS(c_2,c_3)}{\frac{3*(3-1)}{2}} = 1 - \frac{0+0.75+0.75}{3} = 0.5$
  $Coh(M2) = 1 - \frac{CS(c_4,c_5)}{\frac{2*(2-1)}{2}} = 0.25$
  Then we measure the cohesion:
  $Cohesion(I) = 1 - \frac{coh(1)+coh(2)}{2} = 1 - \frac{0.5+0.25}{2} = 0.635$
- Coupling: we start by measuring the coupling between each couple of microservices. Since we have only two of them in this example, our result would be:
  $C(m_1, m_2) = \frac{CS(c_4,c_1)+CS(c_4,c_2)+CS(c_4,c_3)+CS(c_5,c_1)+CS(c_5,c_2)+CS(c_5,c_3)}{3*2} = \frac{0+0.5+0+0.75+0+0.25}{3} = 0.5$
  Then we measure the coupling:
  $Coupling(I) = \frac{C(m_1,m_2)}{\frac{2*(2-1)}{2}} = \frac{0.5}{1} = 0.5$
- Granularity is simpler to measure in this case which is just $\frac{5}{2} = 2.5$

For the sake of simplicity, the values for the similarity matrix were chosen randomly between 0, 0.25, 0.5, 0.75 and the individual does not necessarily represent a pareto-optimal individual.

#### 3.3.4. Change operators

We define crossover and mutation as follows.

***Crossover.*** A crossover operator acts on pairs of candidate individuals. It aligns the individuals, cuts them at a single and random point, and exchanges the fragments between the individuals. To see how it works, consider Fig. 3 with the two "parents" to crossover, called *Parent 1* and *Parent 2*. A "child" is built by incorporating elements randomly from both parents. The other child is obtained by inverting the choices made for the other one.

***Mutation.*** A mutation varies the values of one or more dimensions within the array representing an individual. This variation is applied at random points of the individual, generating a new individual with minor differences with the original one. As illustrated in Fig. 4, an offspring is generated by mutating the dimension at index 4 representing the class `FileContent` which was assigned to microservice 1 instead of microservice 2. Both crossover and mutation operators may impact the classes type $T$. Thus, the new decomposition is analyzed to update the classes types by identifying the `Inner`, and `Interface` ones.

## 4. Empirical evaluation

In this section, we present the results of our evaluation of the proposed approach, MSExtractor. The goal of this evaluation is to assess the effectiveness of our approach in identifying appropriate microservices and compare it with other existing approaches.

### 4.1. Research questions

The study aims at answering the following research questions:

- **RQ1.** To what extent can MSExtractor identify relevant microservices from a functional independence perspective as compared to prior approaches?
- **RQ2.** How does IBEA perform compared to state-of-the-art search algorithms?
- **RQ3.** How similar is the artificial decomposition identified by MSExtractor to originally designed microservices?

### 4.2. Experimental setup

#### 4.2.1. Method analysis for RQ1

**Subjects.** To evaluate our approach, we conduct an experimental study on a benchmark of four following legacy web applications: JPetstore,[1] Springblog,[2] JForum,[3] and Roller.[4] JPetstore is a pet store website and represents the simplest example in our subjects. Springblog is a blogging website. JForum is a discussion board system. Roller on the other hand is a multi-user and group-blog server. Both JForum and Roller are widely used and have a much larger scale when compared to JPetStore for example. More details on these application is provided in Table 1.

We selected these four open-source legacy systems because (*i*) such large scale legacy web applications usually have modularity, maintainability and scalability issues due to their degraded design and growing complexity. They (*ii*) differ in scale ranging from 24 to 534 classes and (*iii*) They were used as a benchmark in recent studies on microservices extraction [12,44,45] and (*iv*) they are monolithic server side applications built with a multi-layered architecture which represents the most common case of legacy applications that have to be updated.

For our experimental setup, we defined equal weights for $\alpha$ and $\beta$, with a value of 0.5.

[1] https://github.com/mybatis/jpetstore-6.
[2] https://github.com/Raysmond/SpringBlog.
[3] https://github.com/rafaelsteil/jforum2.
[4] https://github.com/apache/roller.

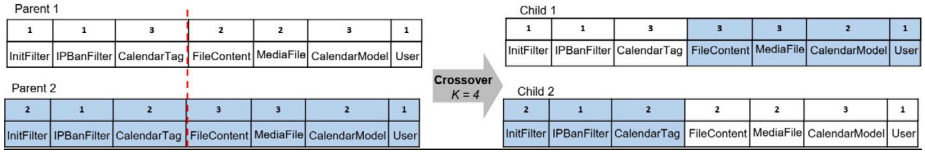



**Fig. 3.** An example of crossover operator.


**Fig. 4.** An example of mutation operator.

**Table 1**
Studied legacy systems benchmark.

| System | Version | Domain | SLOC | # of classes |
|---|---|---|---|---|
| JPetstore | 6.0.2 | e-store | 1438 | 24 |
| Springblog | 2.8.0 | blogging | 3583 | 85 |
| JForum | 2.1.9 | forum | 29550 | 340 |
| Roller | 5.2.0 | blogging | 47602 | 534 |

**Evaluation protocol.** To answer RQ1, we employ four evaluation metrics to assess the quality of the extracted microservices based on measuring their *functional independence*. This measure assesses the extent to which microservices exhibit a bounded context and present their own functionalities with low coupling with other microservices. In particular, the four metrics that we are utilizing, were commonly used in recent studies [2,12,69,70] to assess the quality of Web service interfaces.

- ***CHM*** *(CoHesion at Message level)* : CHM is inspired by $LoC_{msg}$, a widely used metric to measure the cohesion of a service at the message level [12,69,70]. Formally, CHM measures the average cohesion of the set of extracted microservices $\mathcal{M}$ and is formally defined by the following equation:

$$CHM = \frac{1}{|\mathcal{M}|} \times \sum_{\forall m_i \in \mathcal{M}} CHM(m_i) \tag{10}$$

where $CHM(m_i) = \frac{\sum_{\forall (op_j, op_k) \in m_i \ op_j \neq op_k} Sim_{msg}(op_j, op_k)}{|m_i| \times (|m_i|-1)/2}$ if $|\ m_i\ | \neq 0$, and $Sim_{msg}(op_j, op_k)$ calculates the similarity between two public microservice operations $op_j$ and $op_k$ (*i.e.* belong to its interface) at the message level. It is the average of similarity in terms of input parameters and output return types. The higher the CHM, the better the service design cohesion.

- ***CHD*** *(CoHesion at Domain level)* : CHD is inspired by $LoC_{dom}$, a widely used metric to measure the cohesion of a service at the domain level [12,69]. Formally, CHD measures the average cohesion of the set of extracted microservices $\mathcal{M}$ and is defined as follows :

$$CHD = \frac{1}{|\mathcal{M}|} \times \sum_{\forall m_i \in \mathcal{M}} CHD(m_i) \tag{11}$$

where $CHD(m_i) = \frac{\sum_{\forall (op_j, op_k) \in m_i \ op_j \neq op_k} Sim_{dom}(op_j, op_k)}{|m_i| \times (|m_i|-1)/2}$ if $|\ m_i\ | \neq 0$, and $Sim_{dom}(op_j, op_k)$ calculates the similarity between two public microservice operations $op_j$ and $op_k$ (*i.e.*belong to its interface) at the domain level. It is the intersection between the set of domain terms extracted from the operations signatures. The higher the CHD, the better the service design cohesion.

- ***OPN*** *(OPeration Number)*: OPN computes the average number of public operations [12,20] exposed by an extracted microservice to other candidate microservices. The smaller the OPN, the better the microservice quality.

$$OPN = \frac{1}{|\mathcal{M}|} \times \sum_{\forall m_i \in \mathcal{M}} \left| \bigcup_{\forall op_i \in m_i} op_j \right| \tag{12}$$

- ***IRN*** *(InteRaction Number)* : IRN represents the number of method calls among all pairs of extracted microservices [2,12]. The smaller the IRN, the better the quality of candidate microservices, as it reflects loose coupling.

$$IRN = \sum_{\substack{\forall (m_i, m_j) \in \mathcal{M} \\ m_i \neq m_j}} \sum_{\substack{\forall op_k \in m_i \\ \forall op_q \in m_j}} calls(op_k, op_q) \tag{13}$$

where $calls(op_k, op_q)$ is the frequency count of the number of method calls between the methods $op_k$ and $op_q$ (inter-microservice interactions).

**Comparison with existing approaches.** After defining methods to evaluate our approach and analyzing the quality of its results, we need to evaluate it when compared to other approaches that takes the objective of decomposing monolithic applications into microservices. For this reason, we selected three prior approaches, namely, *LIMBO* [48], *FoME* [12], and *MEM* [9]. For LIMBO, we used the default MoJo method to identify the total number of microservices with a threshold of 1, and the default threshold $\tau(\phi)$ as described in [48]. For MEM, which is based on class-to-class semantic coupling, we used a threshold the default threshold of 0.7 which has been empirically defined as a default value [9,12]. For FoME, we used a class-microservice dependency threshold of 0.6. We selected these three competing approaches as they use different decomposition techniques, and have been selected in recent comparative studies [12]. More details about these approaches are provided in Section 2.2.

### 4.2.2. Method analysis for RQ2

**Search algorithm selection.** IBEA is a Multi-Objective Evolutionary Algorithm (MOEA) that recently achieved better benchmark results compared to its counterparts. This research question aims to showcase the advantage of the use of IBEA in our approach instead of other MOEA algorithms. As such, to answer RQ2, we consider three Multi-Objective Evolutionary Algorithms (MOEA), namely IBEA [32], Non-dominated Sorting Genetic Algorithm II (NSGA-II) [33], and Strength Pareto Evolutionary Algorithm 2 (SPEA2) [34]. We selected these three MOEAs as they have widely been applied to solve many engineering problems [71].

**Performance metrics.** We compare the results of the selected algorithms according to a widely-used performance metric, namely *Hypervolume (HV)* [32–34,72,73] used in assessing MOEAs. $HV$ allows assessing how good is the optimal solutions within the Pareto fronts [73]. We use HV in our comparative study for RQ2 since it does not need a reference set of representing the Pareto front, which makes it suitable for many real-world optimization scenarios [74]. The HV result is sensitive to any improvement to a set with respect to Pareto dominance

and reflects the diversity, convergence, spread and cardinality of a solution set as pointed out by Li et al. [74,75].

**Statistical test method.** To compare the performance of these three algorithms, we run each of them 31 times in each system. Then, we use the non-parametric statistical test Wilcoxon in a pairwise fashion to detect performance differences between the compared algorithms [76]. We also assess the effect size using Cliff's delta ($d$) [77] to evaluate the difference magnitude. The Cliff's delta statistic classifies the magnitude of the obtained effect size value into four different levels (*i.e.*, *negligible, small, medium*, or *large*) [77]. We consider the median value of the 31 runs as performance score of each considered algorithm as recommended by Arcuri and Briand [76].

**Implementation and parameters setting.** We used the standard implementations of each of the used algorithms, IBEA, NSGA-II and SPEA provided by MOEA Framework,[5] a widely-used open source tool that allows developing and experimenting with multiobjective evolutionary algorithms (MOEAs). As for the parameter settings, we followed a trial-and-error method to select the hyper-parameters [21] to handle parameter tuning for search-based algorithms which is a common practice in search-based software engineering (SBSE) [21,76,78–80]. To ensure a fair comparison, we used the same parameter settings for the three algorithms as follows: population size = 100; maximum the number of generations = 2,000; crossover probability = 0.8; and mutation probability = 0.1.

#### 4.2.3. Method analysis for RQ3

MSExtractor is meant to be used in a lift-and-shift application modernization scenario to migrate existing monolithic applications. However, since the target microservices extraction can be subjective in some situations, we tried to compare the performance of MSExtractor to the greenfield scenario in which the application is designed from scratch according to the microservice architecture principles. In this way, we grantee that the target microservices are known in advance, which allows us to compare our extracted microservices with the ground truth.

**Subjects.** We conduct an experimental study on 3 open source projects that were built based on the microservices architecture using Java SpringBoot framework.[6] The projects, named Spring PetClinic,[7] Microservices Event Sourcing,[8] and Kanban Board demo,[9] are described in the Table 2. PetClinic is an example project created to showcase the functionalities of the Spring framework and has had multiple versions with different architectures. Spring PetClinic is the implementation based on the microservices architecture and serves as an ideal example project to study in our case. Microservices Event Sourcing is similarly an example project meant to demonstrate a Netflix-like architecture but compared to PetClinic is has a larger scale. We chose this project additionally since its text is not in English and it would be interesting to view our approach's handles this issue. Kanban Board demo is an application that enables users to collaboratively create and edit Kanban boards and tasks. We selected this project for similar reasons in addition to the higher number of microservices it has. These projects were selected in order to have a different size in terms of number of classes, as well as number of involved microservices.

**Evaluation protocol.** To answer RQ3, we restructured each microservice-based project into a monolithic architecture, *i.e.*, merge all classes from all microservices into a single application. Then, we evaluated to which extent MSExtractor was able to retrieve the original microservices, considered as the ground truth.

To evaluate our approach, we need to define measures that can compare two given sets of classes that represent the set of extracted microservices and the ground truth microservices. It is worth noting that the encoding of the clusters for any given decomposition solution can be different. For example, an individual A has the set [112233] while an individual B has the set [331122]. Although the distributions of classes for these individuals are virtually identical, their encoding is different. This issue becomes even more complicated when they are not identical. For example, individual A has the set [111222] while individual B has the set [221111]. In this case, it is ambiguous to identify which clusters correspond to another.

To overcome this challenge, we first need to identify the corresponding ground truth microservice for each extracted microservice. It will be the one that has the most common classes with the extracted microservice. We introduce the function (14) given an extracted microservices $m_i$ and a different set of clusters $M$ (set of ground truth microservices), $Corr(m_i, M)$ selects the ground truth microservices with the largest number of common classes with the extracted microservice.

$$Corr(m_i, M) = \underset{m_j \in M}{argmax}\left(\frac{\left|m_i \cap m_j\right|}{\left|m_i\right|}\right) \tag{14}$$

We then calculate the precision and recall of MSExtractor as follows. In our case, precision is the mean of the percentage of the correctly identified classes out of the total number of identified classes for each extracted microservice. Recall is the mean of the percentage of the correctly identified classes out of the total number of classes in the corresponding microservice in $M_t$ for each extracted microservice.

$$precision = \frac{1}{|M|} \times \sum_{\forall m_i \in M} \frac{\left|m_i \cap Corr(m_i, M_t)\right|}{\left|m_i\right|} \tag{15}$$

$$recall = \frac{1}{|M|} \times \sum_{\forall m_i \in M} \frac{\left|m_i \cap Corr(m_i, M_t)\right|}{\left|Corr(m_i, M_t)\right|} \tag{16}$$

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

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

$$SR = \frac{1}{|M|} \times \sum_{\forall m_i \in M} matching(m_i, Corr(m_i, M_t)) \tag{17}$$

where *matching* is defined in Eq. (18).

$$matching(m_1, m_2) = \begin{cases} 1 & \text{if } \frac{|m_1 \cap m_2|}{|m_1|} \geq threshold \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

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

**Table 2**
Microservices-based projects metadata.

| System | Version | SLOC | # of classes | # of microservices |
|---|---|---|---|---|
| Spring PetClinic | 2.3.6 | 1889 | 43 | 7 |
| Microservices Event Sourcing | 2.8.0 | 4597 | 121 | 12 |
| Kanban Board demo | 0.1.0 | 4380 | 118 | 21 |

### 4.3. Results and discussion

This section dives into the results obtained from the studies of the three formulated research questions (i.e., RQ1, RQ2 and RQ3).

**Results for RQ1.** Table 3 presents the achieved results by each of our approach, *MSExtractor*, and the compared approaches, *FoME* [12], *MEM* [9], and *LIMBO* [48]. The metrics CHM and CHD reflect the cohesion of the extracted microservices, while OPN and IRN reflect the coupling. Both higher cohesion values and lower coupling values

[5] https://github.com/MOEAFramework/MOEAFramework.
[6] https://github.com/spring-projects/spring-boot.
[7] https://github.com/spring-petclinic/spring-petclinic-microservices.
[8] https://github.com/chaokunyang/microservices-event-sourcing.
[9] https://github.com/eventuate-examples/es-kanban-board.

**Table 3**
The achieved results by MSExtractor, FoME, MEM, and LIMBO.

| System | Metric | MSExtractor | FoME | MEM | LIMBO |
|---|---|---|---|---|---|
| Jpetstore | CHM | **0.4-0.5** | 0.7–0.8 | 0.5–0.6 | 0.5-0.6 |
| | CHD | **0.6-0.7** | **0.6-0.7** | **0.6-0.7** | **0.6-0.7** |
| | OPN | 26 | **22** | 39 | 68 |
| | IRN | **32** | 35 | 48 | 329 |
| SpringBlog | CHM | **0.5-0.6** | 0.7–0.8 | 0.6–0.7 | 0.6-0.7 |
| | CHD | **0.6-0.7** | 0.8–0.9 | 0.8–0.9 | 0.7-0.8 |
| | OPN | 10 | **7** | 21 | 147 |
| | IRN | **21** | 26 | 30 | 238 |
| Jforum | CHM | **0.6-0.7** | 0.7–0.8 | **0.6-0.7** | **0.6-0.7** |
| | CHD | **0.4-0.5** | 0.7–0.8 | 0.6–0.7 | 0.6-0.7 |
| | OPN | 36 | 70 | **11** | 94 |
| | IRN | **71** | 97 | 145 | 993 |
| Roller | CHM | **0.5-0.6** | 0.6–0.7 | 0.6–0.7 | 0.6-0.7 |
| | CHD | **0.6-0.7** | 0.8–0.9 | 0.8–0.9 | 0.7-0.8 |
| | OPN | 63 | **56** | 66 | 1062 |
| | IRN | **946** | 1441 | 2786 | 46964 |

**Table 4**
Statistical tests results of IBEA compared to NSGA-II and SPEA2.

| Project | Measure | IBEA vs NSGA-II | IBEA vs SPEA2 |
|---|---|---|---|
| JPetStore | *HV* | 0.042 vs 0.024 | 0.042 vs 0.035 |
| | *p-value* | $2.2 \times 10^{-16}$ | $2.2 \times 10^{-16}$ |
| | *Effect size* | 1 | 0.96 |
| | *Magnitude* | Large | Large |
| SpringBlog | HV | 0.68 vs 0.57 | 0.68 vs 0.6 |
| | *p-value* | $2.2 \times 10^{-16}$ | $1.64 \times 10^{-11}$ |
| | *Effect size* | 0.995 | 0.778 |
| | *Magnitude* | Large | Large |
| Jforum | *HV* | 0.63 vs 0.47 | 0.63 vs 0.54 |
| | *p*-value | $2.2 \times 10^{-16}$ | $6.25 \times 10^{-10}$ |
| | *Effect size* | 0.968 | 0.717 |
| | *Magnitude* | Large | Large |
| Roller | *HV* | 0.65 vs 0.52 | 0.65 vs 0.61 |
| | *p-value* | $4.76 \times 10^{-11}$ | 0.00129 |
| | *Effect size* | 0.762 | 0.372 |
| | *Magnitude* | Large | Medium |

indicate better performance than alternative approaches. The cohesion results are provided in the form of an interval, *e.g.*, $[0.5 - 0.6]$, instead of specific values since there are non significant differences between CHM values as well as between those of CHD, as suggested by Jin et al. [12]. We observe through Table 3 that our approach, MSExtractor, outperforms the three competing techniques for the four studied systems in the majority of metrics. In particular, for smaller systems such as `JPetStore` (24 classes), the achieved results on the four metrics are comparable. Indeed, this system represents a relatively smaller search space where deterministic approaches may achieve high performance. For larger systems, such as `Roller` and `JForum` (340 and 534, respectively), there is a clear superiority achieved by MSExtractor compared to the three compared approaches, in terms of CHM, CHD, as well as IRN.

We also observe from Table 3 that FoME tends to provide better results in terms of OPN in three out of the four systems. This superiority is justified by the fact that FoME excludes a relatively substantial number of classes that are not covered by the dynamic analysis scenarios. These excluded classes will be, in turn, excluded from the candidate microservices. Obviously, ignoring a number of classes may improve coupling, but would provide functionally incomplete microservice candidates. These classes are generally related to exceptions, *e.g.*, the classes `MailingException`, `FilePathException`, `BootstrapException` from the project *Roller*, or to other third-party or no-behavior classes, *e.g.*, `YoutubeLinkTransformer`, `MessageHelper`, and `SecurityConfig` from the project *Springblog*. Such exclusion of classes from microservices would result in an incomplete architecture and would require a manual inspection by developers performing the migration. Indeed, one of the reasons behind the performance of our approach as compared to the existing approaches can be related to the distinctions between Inner and Interface classes. That is, in practice a microservice is composed of two types of classes (1) the Inner classes only have internal connections to other classes of the same microservice, whereas (2) Interface classes define the functionality provided by the microservice to other microservices, defining the microservice interface. However, existing approaches do not consider such distinction when identifying candidate microservices leading to further manual inspection from the developer to adjust them.

Moreover, another interesting feature of our approach is that it takes the advantage of MOEAs to provide a variety of near-optimal solutions allowing the developer to explore different microservice candidate solutions. Fig. 5 depicts the main interface of MSExtractor allowing the developer to navigate and choose the decomposition solution that best matches its criteria.

**Results for RQ2.** Fig. 6 and Table 4 report the achieved hypervolume results by each of the algorithms, IBEA, NSGA-II and SPEA2 over 31 simulation runs for every testbed system. We observe that IBEA clearly outperforms both NSGA-II and SPEA2 in the four projects. The Wilcoxon statistical tests analysis, with a *p*-value below 0.05, indicates significant differences in favor of IBEA over SPEA2 and NSGA-II. Moreover, the Cliff's delta effect between IBEA and each of NSGA-II and SPEA2 is large for the four projects, except with SPEA2 on the Roller project which was medium (cf. Table 4). A possible explanation for the lower efficiency of NSGA-II's hypervolume could be related to its crowding operator's mechanism which tends to impair with 3 or more objectives, while the qualitative indicator function over Pareto approximations shows its advantages for IBEA. Thus, IBEA seems to be a better choice for our problem as a search method since it yields more solutions in the approximate Pareto frontier.

**Results for RQ3.** In reference to Table 5, we observe that the extracted microservices by MSExtractor manage to achieve high values of precision ranging from 0.73 to 0.91 for the three studied systems. As compared to the three other existing approaches, MSExtractor is clearly better which means that our is able to identify a large number of correct classes with regards to the microservices they belong to. In addition, in the case of the *SR* metric, when lowering the threshold defined in (18), we observe an increase in the successfully retrieved microservices which suggests the confidence level for their precision values to be high.

For the recall, MSExtractor achieved less high values than precision ranging from 0.61 to 0.74 for the three studied projects. The results indicate that our approach clearly outperforms the other considered approaches. To get a better understanding of the achieved results, we investigated the reasons behind the missed classes, and we found that MSExtractor was not able to handle the cases of almost identical classes that should not be grouped together. In fact, in the microservices architecture, we can observe redundant classes, that abstract the same concept within each microservice, in order to ensure proper communication and data transfer between the different microservices. These classes represent the same data type but each of them is adapted to the need of its corresponding microservice as well as the database type used by the microservice. Hence, we advocate that while automated microservices identification approaches are helpful, the role of the human developer is crucial to validate and calibrate the extracted microservices [9,10,42].

To provide a more qualitative sense, we showcase the example of the *Microservices Event Sourcing* project which is an online shopping platform that was built according to the microservice architecture. Fig. 7 draws the original microservices decomposition of this application. Most of the microservices implement a domain-specific service such as *account-service*, *catalog-service, inventory-service*, *order-service*, *shopping-cart-service* and *user-service*. These services are of different sizes

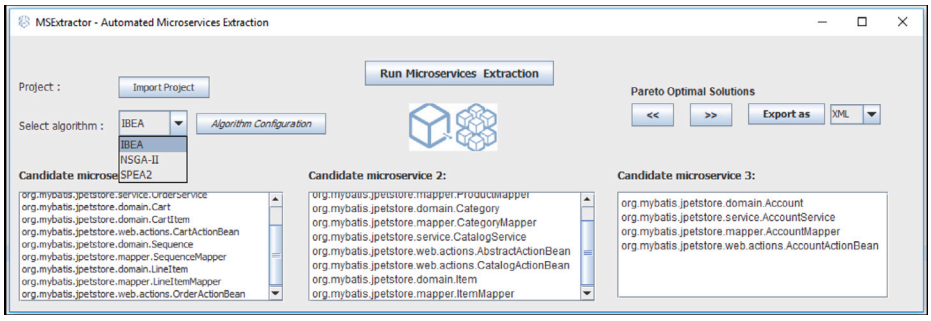



**Fig. 5.** A screenshot of MSExtractor User Interface.

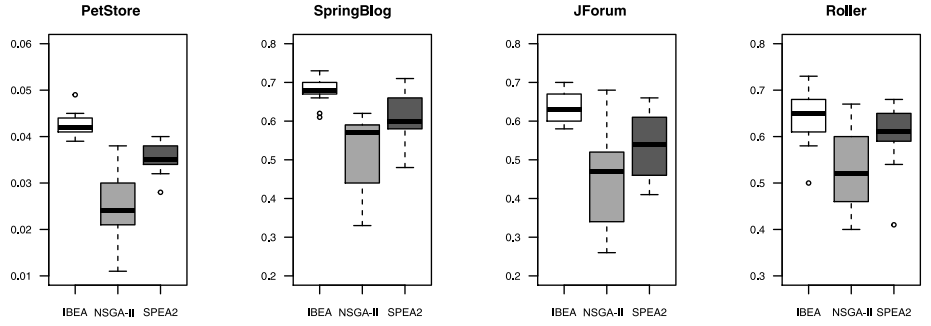



**Fig. 6.** Boxplots of the hypervolume results achieved by IBEA, NSGA-II and SPEA2.

**Table 5**
The precision, recall and success rate results achieved by MSExtractor, FoME, MEM, and LIMBO.

| System | Metric | MSExtractor | FoME | MEM | LIMBO |
|---|---|---|---|---|---|
| Spring PetClinic | precision | **0.91** | 0.78 | 0.53 | 0.43 |
| | recall | **0.68** | 0.61 | 0.53 | 0.39 |
| | SR@5 | **0.88** | 0.72 | 0.5 | 0.41 |
| | SR@7 | **0.88** | 0.61 | 0.42 | 0.33 |
| | SR@9 | **0.88** | 0.61 | 0.35 | 0.33 |
| Microservices Event Sourcing | precision | **0.73** | 0.58 | 0.49 | 0.56 |
| | recall | **0.74** | 0.6 | 0.56 | 0.55 |
| | SR@5 | **0.62** | 0.55 | 0.49 | 0.56 |
| | SR@7 | 0.54 | 0.49 | 0.4 | **0.56** |
| | SR@9 | **0.54** | 0.42 | 0.4 | 0.48 |
| Kanban Board demo | precision | **0.75** | 0.7 | 0.66 | 0.62 |
| | recall | 0.61 | 0.56 | 0.53 | **0.64** |
| | SR@5 | **0.82** | 0.69 | 0.5 | 0.62 |
| | SR@7 | **0.73** | 0.66 | 0.46 | 0.55 |
| | SR@9 | 0.5 | 0.52 | 0.46 | **0.51** |

in terms of the number of classes varying from 8 classes for the *catalog service* to 29 classes for the order service. The other microservices provide technical requirements related to the underlying platform and the framework used to implement the microservices architecture such as services registry and discovery, configuration management and the distributed logging. For this type of infrastructure and configuration microservices, MSExtractor was not able to retrieve the correct target microservices for some classes such as the classes *DiscoveryServiceApplication* in *discovery-service*, *HystrixDashboardAppilcation* in *hystrix-service* and *ConfigServiceApplication* in *config-service*. As we can see in Fig. 7, these microservices *discovery-service*, *hystrix-service*, and *config-service* are only composed of a single class. However, MSExtractor is designed to avoid mining single class microservices in order to prevent solutions consisting of single element clusters which theoretically are the most granular and cohesive decomposition but practically less useful decomposition from a domain-driven designed perspective. In general, a naïve decomposition of a legacy application would be to create microservices each containing a single class. This decomposition would have the optimal granularity and cohesiveness but it is clearly not practical since we will be ignoring the overhead added by the required communications between these microservices. MSExtractor is designed to avoid this type of decomposition. The same can be said about the other extreme decomposition which consists of a single microservice. This decomposition has the optimal coupling score but obviously does not achieve the primary objective of decomposing the application. This shows both extremes of the objective metrics we used and how they balance each other. However, the microservices discovery-service, hystrix-service, and config-service have only a single class since each of them represents a very specific and singular technical aspect in the microservices implementation. Generating these technical microservices is out of the scope of our paper. An improvement in the future to this evaluation process would be to eliminate technical classes from the input and utilize only domain classes.

In the case of the microservices `account-service`, `order-service`, `shopping-cart-service`, `user-service`, and `inventory-service` that have to interact and exchange information with each other, we can find a duplicated class called `BaseEntity` which represents a base API that other Entity classes have to implement. Due to the overlapping nature of these classes, their semantic similarity is very high which leads MSExtractor to group

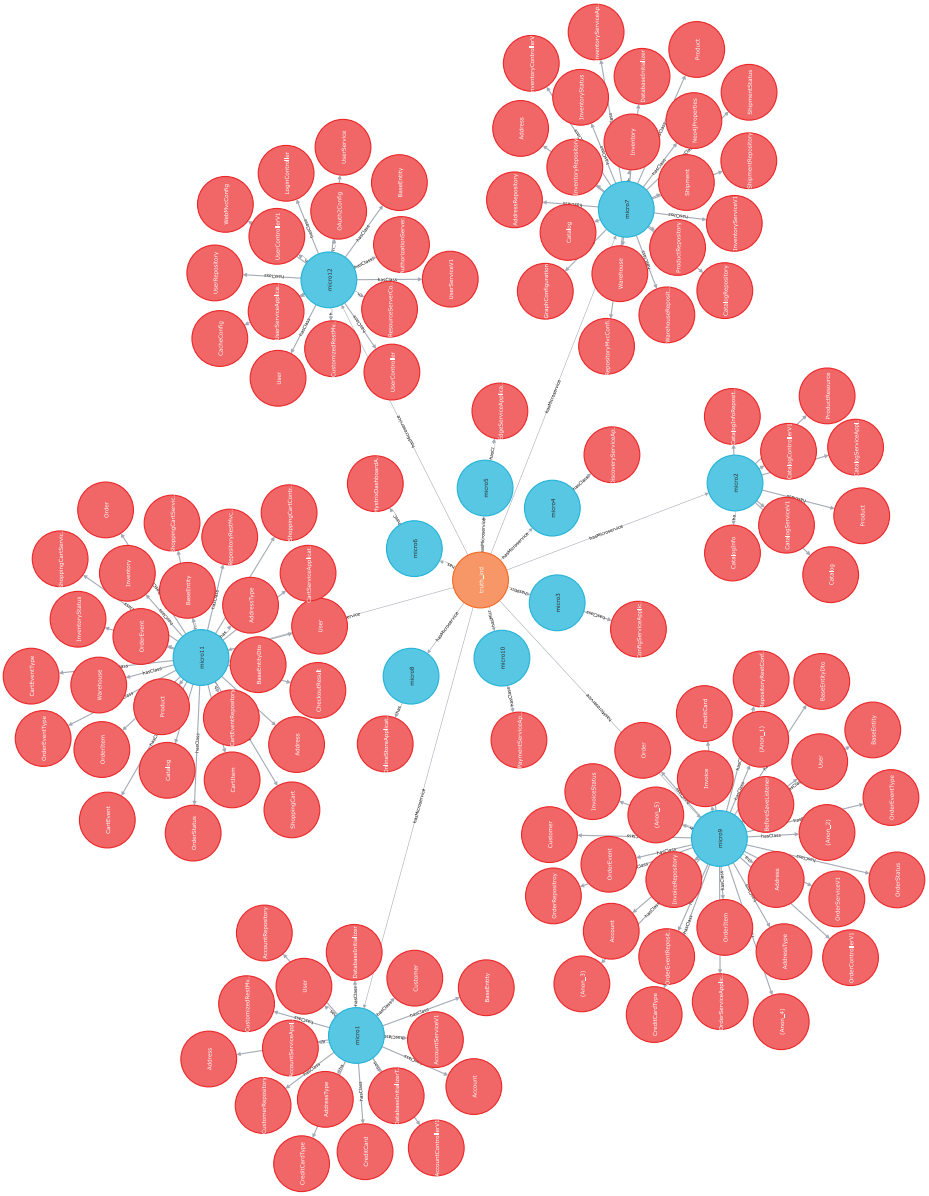


**Fig. 7.** Event Sourcing microservices decomposition.

them in a single utility microservice composed of the four duplicated `BaseEntity` classes as well as two data transfer object `BaseEntityDto` classes. A similar phenomenon was observed for other classes of the `account-service` that we show in Fig. 8. For instance, multiple classes that extend `BaseEntity` such as `Account`, `Address`, `CreditCard`, `Order`, `Customer`, and `User` were grouped with semantically identical or very similar classes.

The above issues showcase the need of duplicate classes and purely technical classes in a microservice-based application which is not directly addressed by the microservices decomposition task. When developing a monolithic application with an Object-Oriented design, it is crucial to reduce redundancy as much as possible which is why interfaces and abstract classes are often used to group together shared functionalities or code. However, the classes that inherit or implement them do not necessarily serve the same use case. As such, when decomposing a legacy application, these shared classes can be duplicated for each microservice that requires them. In addition, some microservices will have to exchange data between them which may require having a model class in each microservices that represents the shared datatype. This represents another example of classes that will need to be duplicated. These examples lies beyond the scope of our current research which focuses on decomposing the legacy application and recommending a microservice architecture to developers. However, to achieve a functioning microservice-based application, human experts will be required to validate, customize and modify these decompositions in order to finalize the task.

## 5. Discussion

In the following, we discuss several decisions and choices made for our approach. Most approaches that handle the problem of decomposing legacy applications can be split into two categories which are dynamic analysis approaches and static analysis approaches. Dynamic analysis is based on analyzing and parsing the execution traces of the said applications while static analysis is based on analyzing the source code of the legacy software. Since execution traces show the list of classes/methods that are used in conjunction and since they are usually tied to a specific use case defined by the user, the results achieved by dynamic analysis approaches are more precise. However, this precision

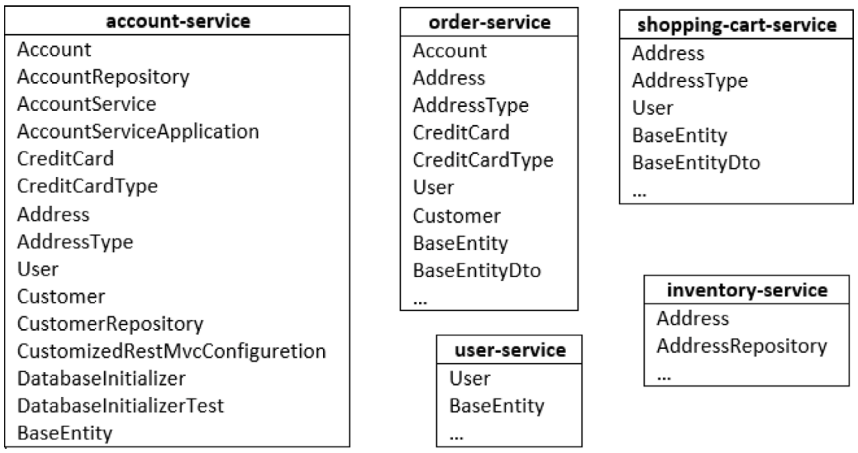



**Fig. 8.** Account-service classes.

comes at a very high cost because there are two major perquisites. First, there needs to be enough class/method coverage within the execution traces in order to have enough data for the decomposition. Second, the execution data needs to be properly labeled in order to associate them to use cases which will define the base of the decomposition. However, guaranteeing both of these conditions is hard, especially for legacy applications where one of the driving reasons for automating the decomposition is the cost associated with obtaining developers that are familiar with the software. On the other hand, the source code is almost always available when trying to decompose the software application. As such, a solution based on static analysis can be used in most settings and can easily be generalized while still including all of the components of the input application. Nonetheless, an ideal solution will be to incorporate both approaches. This solution should be able to generalize but provide better and more precise results when execution data is available. This type of solution will be tackled in future work.

A legacy application can be decomposed at the method level rather than at the class level to properly handle methods that can be misplaced or too big classes. However, we believe a good approach is to start with code refactoring before microservices extraction at the class level. Refactoring can be in the method level such as fixing god classes (extract class, move method, move attribute, etc.), feature envy (move method, extract method), renaming code identifiers, cleaning up duplicate code, and other bad design and implementation practices.

Microservices are often small in size by design and contain only a few classes that achieve a specific functionality. Hence, when extracting microservices from a monolithic application, one of the objectives of the final decomposition is to avoid extremely large or extremely small microservices.

If we only used the cohesion as the only objective function, we would have microservices containing each exactly 1 class as that would correspond to the maximum cohesion. Similarly, if we only used the coupling as the objective function, its optimal value is achieved when there is only a single microservice with all classes included. Theoretically, when we combine these objectives, the resulting decompositions should have a compromise between these metrics and as such would be more balanced. However, this was not the case in practice since most decompositions contained either large or very small microservices. For this reason, we decided to introduce this metric as an additional objective function that controls the mean size of the microservices in the decomposition. In addition, it provides an option to the user to control how strict the granularity of the target decomposition should be using a threshold value.

The objective of our approach is to extract microservices from a monolithic application by grouping together the classes that are the most semantically and structurally dependent. Using the objective functions, we defined, the final generations in the approach will mostly have individuals that respect this target as much as possible, and thanks to the nature of evolutionary algorithms, weaker candidates which have large coupling values and/or high granularity and cohesion values will be dropped in the initial generations of the algorithm. Nonetheless, by nature, decomposing a monolithic application will result in microservices that may break the dependencies since some of their dependencies will be tied to other microservices. The objective of this research is to recommend potential microservices. After which, a refactoring process will be needed to achieve a functioning microservice-based application, human experts will be required to validate, customize and modify these decompositions. This step is out of the scope of our approach and will be tackled in future work.

Given the difference in the architectural patterns used for the monolithic and microservices applications, a portion of the source code will include implementations of these patterns and as such would become obsolete when migrating to the newer architecture. For example, in the JPetStore project, there are multiple classes that implement the Model–View–Controller pattern which will have to be changed or removed after the decomposition phase. On the other hand, new classes in the microservices architecture will need to be added to implement a RESTful API in order to expose the functionalities of the microservices. Our objective behind the distinction between inner and interface classes is to provide a starting point for this task and facilitate it since we identify the interface classes that will have to be refactored. In our future work, we are interested in empirically quantifying the benefits provided by this distinction.

Our approach uses the source code in order to extract the dependencies between classes. The dependencies were in the form of relationships that we called the structural similarities or shared terms that we called the semantic similarity. However, this method of comparing the classes is heuristic in nature and is meant to replace the actual use case relationships between the classes which are rarely explicitly available. For these reasons, there is an inherent risk to grouping the classes that are similar based on these views but are not related otherwise.

## 6. Threats to validity

**Internal threat to validity** concerns our ability that might have influenced our results. Our approach relies on randomly generating microservice decomposition solutions, and their evolution is stochastic. Therefore, there is a random variation in findings. To mitigate this threat, we gathered our experiments from 31 runs and the findings were statistically analyzed and report the median value. The comparison between evolutionary algorithms was only performed using the hypervolume. Being one of the widely used performance indicator, as it was strictly compliant with the Pareto dominance [72] does not shadow

the need for comparing with more indicators to gain better balanced insights of algorithms evolution, in terms of convergence and diversity.

**Threats to construct validity** could be related to the performance measures. We believe that there is a little bias towards the used evaluation metrics. To conduct a more robust evaluation, we used different measures such as cohesion (in RQ1) hypervolume (in RQ2) and correctness, precision and recall (in RQ3) to further evaluate our approach with different experimental scenarios. Other evaluation metrics can be also considered with other evaluation scenarios.

**Threats to external validity** link factors preventing the generalization of our findings. Our experiments were conducted on a limited set of four legacy web applications. More projects are eventually needed to generalize the results of the study. However, the projects under test were diverse enough in nature and architecture, which was a challenge to the performance of our evolutionary algorithm. Another important threat is the limited qualitative analysis that was only performed by the authors. Better feedback on the quality of the proposed solution can be issued by software architects and experts who are practicing the extraction on a regular basis. To mitigate this issue, surveying them and pitching our solutions makes the qualitative part of our future investigations. Moreover, while we considered three existing approaches as baselines to compare the performance of our approach, we cannot generalize our findings. As part of our future work, we are interested in comparing our approach with other state-of-the-art approaches for a more robust evaluation.

Research in the field of automated microservices extraction is still in its infancy, thus researchers and practitioners may suggest different strategies for the migration process towards a microservices architecture [43,81], However, most of the suggested strategies share common rationale with different steps (i) identification of microservices boundaries, (ii) execution of the modernization or the migration, and (iii) handling technical aspects with regards to the frameworks and infrastructure.

## 7. Conclusions and future work

In this paper, we proposed MSExtractor, a novel approach that tackles the microservices extraction problem and formulates it as a multi-objective combinatorial optimization problem. Specifically, MSExtractor employs the Indicator-Based Evolutionary Algorithm (IBEA) to drive a search process towards an optimal decomposition of a given legacy application while considering structural and semantic dependencies in the source code. Our evaluation demonstrates that MSExtractor is able to extract cohesive and loosely coupled services with higher performance than three other considered methods. Moreover, our results show that IBEA has the best performance, in comparison with state-of-the-art multi-objective search algorithms, NSGA-II and SPEA2. As we only focused on the identification of microservices boundaries, we plan in our future work to investigate the other steps of the migration process towards containerization and pre-deployment configuration of our candidate microservices. We also plan to evaluate our approach from developers and software architects perspective while using more testbed systems. We also plan to consider non-functional criteria that are essential in the context of microservices architecture, including the scalability and availability of the system.

## Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2022.106996.

## Data availability

Data will be made available on request.

## Acknowledgments

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

## References

[1] S. Sharma, R. Rajesh, D. Gonzalez, Microservices: Building Scalable Software, Packt Publishing Ltd, 2017.
[2] S. Newman, Building Microservices: Designingfine-Grained Systems, O'Reilly Media.
[3] J. Fritzsch, J. Bogner, S. Wagner, A. Zimmermann, Microservices migration in industry: intentions, strategies, and challenges, in: 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2019, pp. 481–490.
[4] D. Taibi, V. Lenarduzzi, C. Pahl, Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation, IEEE Cloud Comput. 4 (5) (2017) 22–32.
[5] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S.T. Larsen, S. Dustdar, Microservices: Migration of a mission critical system, IEEE Trans. Serv. Comput. 14 (5) (2018) 1464–1477.
[6] F.H. Vera-Rivera, C. Gaona, H. Astudillo, Defining and measuring microservice granularity—a literature overview, PeerJ Comput. Sci. 7 (2021) e695.
[7] A. Balalaie, A. Heydarnoori, P. Jamshidi, Migrating to cloud-native architectures using microservices: an experience report, in: European Conference on Service-Oriented and Cloud Computing, Springer, 2015, pp. 201–215.
[8] P. Di Francesco, P. Lago, I. Malavolta, Migrating towards microservice architectures: an industrial survey, in: 2018 IEEE International Conference on Software Architecture, ICSA, IEEE, 2018, pp. 29–2909.
[9] G. Mazlami, J. Cito, P. Leitner, Extraction of microservices from monolithic software architectures, in: 2017 IEEE Inter.Conf. on Web Services, ICWS.
[10] C. Richardson, Microservices: Decomposing applications for deployability and scalability, InfoQ 25 (2014) 15–16, URL https://www.infoq.com/articles/microservices-intro.
[11] I. Saidani, A. Ouni, M.W. Mkaouer, A. Saied, Towards automated microservices extraction using muti-objective evolutionary search, in: International Conference on Service-Oriented Computing, Springer, 2019, pp. 58–63.
[12] W. Jin, T. Liu, Q. Zheng, D. Cui, Y. Cai, Functionality-oriented microservice extraction based on execution trace clustering, in: 2018 IEEE International Conference on Web Services, ICWS, IEEE, 2018, pp. 211–218.
[13] P. Di Francesco, I. Malavolta, P. Lago, Research on architecting microservices: trends, focus, and potential for industrial adoption, in: 2017 IEEE International Conference on Software Architecture, ICSA, IEEE, 2017, pp. 21–30.
[14] L. Carvalho, A. Garcia, T.E. Colanzi, W.K. Assunção, J.A. Pereira, B. Fonseca, M. Ribeiro, M.J. de Lima, C. Lucena, On the performance and adoption of search-based microservice identification with tomicroservices, in: IEEE International Conference on Software Maintenance and Evolution, ICSME, 2020, pp. 569–580.
[15] W.K. Assunção, T.E. Colanzi, L. Carvalho, J.A. Pereira, A. Garcia, M.J. de Lima, C. Lucena, A multi-criteria strategy for redesigning legacy features as microservices: an industrial case study, in: IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, 2021, pp. 377–387.
[16] L. Baresi, M. Garriga, A. De Renzis, Microservices identification through interface analysis, in: Europ. Conf.on Service-Oriented and Cloud Computing.
[17] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann, Service cutter: a systematic approach to service decomposition, in: European Conference on Service-Oriented and Cloud Computing, 2016, pp. 185–200.
[18] K. Bastani, Using graph analysis to decompose monoliths into microservices with neo4j, 2015, URL https://www.kennybastani.com/2015/05/graph-analysis-microservice-neo4j.html, this third source is not interesting.
[19] R.G. Kula, A. Ouni, D.M. German, K. Inoue, An empirical study on the impact of refactoring activities on evolving client-used APIs, Inf. Softw. Technol. 93 (2018) 186–199.
[20] S. Adjoyan, A.-D. Seriai, A. Shatnawi, Service identification based on quality metrics object-oriented legacy system migration towards soa, in: Software Engineering and Knowledge Engineering, SEKE, 2014, pp. 1–6.
[21] M. Harman, S.A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Comput. Surv. 45.
[22] K. Praditwong, M. Harman, X. Yao, Software module clustering as a multi-objective search problem, IEEE Trans. Softw. Eng.
[23] M.A. Saied, E. Raelijohn, E. Batot, M. Famelis, H. Sahraoui, Towards assisting developers in API usage by automated recovery of complex temporal patterns, Inf. Softw. Technol. 119 (2020) 106213.
[24] M. Paixao, M. Harman, Y. Zhang, Y. Yu, An empirical study of cohesion and coupling: balancing optimization and disruption, IEEE Trans. Evol. Comput. 22 (3) (2018) 394–414.
[25] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, A. Ouni, Many-objective software remodularization using NSGA-III, ACM Trans. Softw. Eng. Methodol.
[26] N. Almarimi, A. Ouni, S. Bouktif, M.W. Mkaouer, R.G. Kula, M.A. Saied, Web service API recommendation for automated mashup creation using multi-objective evolutionary search, Appl. Soft Comput. 85 (2019) 105830.

[27] S. Huppe, M.A. Saied, H. Sahraoui, Mining complex temporal api usage patterns: an evolutionary approach, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C, IEEE, 2017, pp. 274–276.

[28] M. Gallais-Jimenez, H.A. Nguyen, M.A. Saied, T.N. Nguyen, H. Sahraoui, API misuse detection an immune system inspired approach, 2020, arXiv preprint arXiv:2012.14078.

[29] G. Ruhe, Optimization in Software Engineering: A pragmatic approach, in: Contemporary Empirical Methods in Software Engineering, Springer, 2020, pp. 235–261.

[30] A. Ouni, Search based software engineering: challenges, opportunities and recent applications, in: Genetic and Evolutionary Computation Conference, GECCO, 2020, pp. 1114–1146.

[31] M.A. Saied, H. Sahraoui, E. Batot, M. Famelis, P.-O. Talbot, Towards the automated recovery of complex temporal api-usage patterns, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2018, pp. 1435–1442.

[32] E. Zitzler, S. Künzli, Indicator-based selection in multiobjective search, in: International Conference on Parallel Problem Solving from Nature, Springer.

[33] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. (2002).

[34] E. Zitzler, M. Laumanns, L. Thiele, SPEA2: Improving the Strength Pareto Evolutionary Algorithm, TIK-Report 103, Eidgenössische Technische Hochschule Zürich (ETH), Institut für Technische, 2001.

[35] M. Fowler, K. Beck, J. Brant, W. Opdyke, d. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[36] A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables devops: Migration to a cloud-native architecture, IEEE Softw. 33 (3).

[37] L.A. Vayghan, M.A. Saied, M. Toeroe, F. Khendek, A Kubernetes controller for managing the availability of elastic microservice based stateful applications, J. Syst. Softw. 175 (2021) 110924.

[38] L.A. Vayghan, M.A. Saied, M. Toeroe, F. Khendek, Microservice based architecture: Towards high-availability for stateful applications with kubernetes, in: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security, QRS, IEEE, 2019, pp. 176–185.

[39] L.A. Vayghan, M.A. Saied, M. Toeroe, F. Khendek, Kubernetes as an availability manager for microservice applications, 2019, arXiv preprint arXiv:1901.04946.

[40] L.A. Vayghan, M.A. Saied, M. Toeroe, F. Khendek, Deploying microservice based applications with kubernetes: Experiments and lessons learned, in: 2018 IEEE 11th International Conference on Cloud Computing, CLOUD, IEEE, 2018, pp. 970–973.

[41] N. Dragoni, S. Giallorenzo, A.L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: yesterday, today, and tomorrow, in: Present and Ulterior Software Engineering, Springer, 2017, pp. 195–216.

[42] S. Hassan, N. Ali, R. Bahsoon, Microservice ambients: An architectural meta-modelling approach for microservice granularity, in: IEEE International Conference on Software Architecture, ICSA, 2017, pp. 1–10.

[43] R. Rajesh, Spring Microservices: Build Scalable Microservices with Spring, Docker, and Mesos, Packt Publishing Ltd, 2016.

[44] O. Al-Debagy, P. Martinek, A microservice decomposition method through using distributed representation of source code, Scalable Comput.: Pract. Exp. 22 (1) (2021) 39–52.

[45] A.K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, D. Banerjee, Mono2Micro: a practical and effective tool for decomposing monolithic java applications to microservices, 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (2021).

[46] U. Desai, S. Bandyopadhyay, S. Tamilselvam, Graph neural network to dilute outliers for refactoring monolith application, 2021, arXiv:2102.03827.

[47] M. Brito, J. Cunha, J.a. Saraiva, Identification of microservices from monolithic applications through topic modelling, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, 2021, pp. 1409–1418.

[48] P. Andritsos, V. Tzerpos, Information-theoretic software clustering, IEEE Trans. Softw. Eng. 31 (2) (2005) 150–165.

[49] A. Shatnawi, H. Shatnawi, M.A. Saied, Z.A. Shara, H. Sahraoui, A. Seriai, Identifying software components from object-oriented APIs based on dynamic analysis, in: Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 189–199.

[50] F. Freitas, A. Ferreira, J. Cunha, Refactoring java monoliths into executable microservice-based applications, in: 25th Brazilian Symposium on Programming Languages, 2021, pp. 100–107.

[51] M. Harman, B.F. Jones, Search-based software engineering, Inf. Softw. Technol. 43 (14) (2001) 833–839.

[52] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493.

[53] M.A. Saied, O. Benomar, H. Abdeen, H. Sahraoui, Mining multi-level api usage patterns, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER, IEEE, 2015, pp. 23–32.

[54] M.A. Saied, A. Ouni, H. Sahraoui, R.G. Kula, K. Inoue, D. Lo, Improving reusability of software libraries through usage pattern mining, J. Syst. Softw. 145 (2018) 164–179.

[55] A hierarchical DBSCAN method for extracting microservices from monolithic applications, in: The International Conference on Evaluation and Assessment in Software Engineering 2022, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450396134, 2022, pp. 201–210, http://dx.doi.org/10.1145/3530019.3530040.

[56] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2004.

[57] J. Lewis, M. Fowler, Microservices: a definition of this new architectural term, 2014, MartinFowler.Com 25.

[58] M.A. Saied, H. Sahraoui, B. Dufour, An observational study on api usage constraints and their documentation, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER, IEEE, 2015, pp. 33–42.

[59] A. Ouni, M. Kessentini, H. Sahraoui, M.S. Hamdi, Search-based refactoring: Towards semantics preservation, in: IEEE International Conference on Software Maintenance, ICSM, 2012, pp. 347–356.

[60] M.A. Saied, H. Abdeen, O. Benomar, H. Sahraoui, Could we infer unordered api usage patterns only using the library source code? in: 2015 IEEE 23rd International Conference on Program Comprehension, IEEE, 2015, pp. 71–81.

[61] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, K. Deb, Multi-criteria code refactoring using search-based software engineering: An industrial case study, ACM Trans. Softw. Eng. Methodol. 25.

[62] M.A. Saied, O. Benomar, H. Sahraoui, Visualization based API usage patterns refining, in: 2015 IEEE 3rd Working Conference on Software Visualization, VISSOFT, IEEE, 2015, pp. 155–159.

[63] M.A. Saied, H. Abdeen, O. Benomar, H. Sahraoui, Could we infer API usage patterns only using the library source code, in: 23rd International Conference on Program Comprehension, ICPC, 2015, Available at http://www-etud.iro.umontreal.ca/benomaro/publi/cwiaupulsc.pdf.

[64] N. Anquetil, T.C. Lethbridge, Recovering software architecture from the names of source files, J. Softw. Maint.: Res. Pract. 11 (3) (1999) 201–221.

[65] E. Merlo, I. McAdam, R. De Mori, Source code informal information analysis using connectionist models, in: IJCAI, 1993.

[66] M.A. Saied, H. Sahraoui, A cooperative approach for combining client-based and library-based api usage pattern mining, in: 2016 IEEE 24th International Conference on Program Comprehension, ICPC, IEEE, 2016, pp. 1–10.

[67] A. Corazza, S. Di Martino, V. Maggio, G. Scanniello, Investigating the use of lexical information for software system clustering, in: 2011 15th European Conference on Software Maintenance and Reengineering, IEEE, 2011, pp. 35–44.

[68] E.R. Hruschka, R.J. Campello, A. Freitas, A.C. De Carvalho, et al., A survey of evolutionary algorithms for clustering, IEEE Trans. Syst. Man Cybern. C 39 (2) (2009) 133–155.

[69] D. Athanasopoulos, A.V. Zarras, G. Miskos, V. Issarny, P. Vassiliadis, Cohesion-driven decomposition of service interfaces without access to source code, IEEE Trans. Serv. Comput. 8 (4) (2015) 550–562.

[70] A. Ouni, H. Wang, M. Kessentini, S. Bouktif, K. Inoue, A hybrid approach for improving the design quality of web service interfaces, ACM Trans. Internet Technol. 19 (1) (2018) 4.

[71] M. Harman, S. Mansouri, Y. Zhang, Search based software engineering: A comprehensive analysis and review of trends techniques and applications, Tech. Rep. TR-09-03, King's College, London, UK, 2009.

[72] E. Zitzler, L. Thiele, M. Laumanns, C.M. Fonseca, V.G. Da Fonseca, Performance assessment of multiobjective optimizers: An analysis and review, IEEE Trans. Evol. Comput. 7 (2) (2003) 117–132.

[73] S. Wang, S. Ali, T. Yue, Y. Li, M. Liaaen, A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering, in: 38th International Conference on Software Engineering, 2016, pp. 631–642.

[74] M. Li, X. Yao, Quality evaluation of solution sets in multiobjective optimisation: A survey, ACM Comput. Surv. 52 (2) (2019) 1–38.

[75] M. Li, T. Chen, X. Yao, A critical review of: "a practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering": essay on quality indicator selection for SBSE, in: The 40th International Conference on Software Engineering: New Ideas and Emerging Results, 2018, pp. 17–20.

[76] A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in: Software Engineering (ICSE), 2011 33rd International Conference on, 2011, pp. 1–10.

[77] N. Cliff, Ordinal Methods for Behavioral Data Analysis, Psychology Press, 2014.

[78] I. Saidani, A. Ouni, W. Mkaouer, Detecting skipped commits in continuous integration using multi-objective evolutionary search, IEEE Trans. Softw. Eng. (2021).

[79] A. Ouni, M. Kessentini, K. Inoue, M.O. Cinnéide, Search-based web service antipatterns detection, IEEE Trans. Serv. Comput. 10 (4) (2015) 603–617.

[80] O. Benomar, H. Abdeen, H. Sahraoui, P. Poulin, M.A. Saied, Detection of software evolution phases based on development activities, in: 2015 IEEE 23rd International Conference on Program Comprehension, IEEE, 2015, pp. 15–24.

[81] D. Wolfart, W.K. Assunção, I.F. da Silva, D.C. Domingos, E. Schmeing, G.L.D. Villaca, D.d.N. Paza, Modernizing legacy systems with microservices: a roadmap, in: Evaluation and Assessment in Software Engineering, 2021, pp. 149–159.