A Multi-Criteria Strategy for Redesigning Legacy Features as Microservices: An Industrial Case Study

Wesley K. G. Assunção*†, Thelma Elita Colanzi*‡, Luiz Carvalho*, Juliana Alves Pereira*,
Alessandro Garcia*, Maria Julia de Lima§, Carlos Lucena*
*DI – Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil.
†PPGComp – Western Paraná State University, Cascavel, Brazil.
‡DIN – State University of Maringá, Maringá, Brazil.
§Tecgraf Institute – Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil.

***Abstract*—Microservices are small and autonomous services that communicate through lightweight protocols. Companies have often been adopting microservices to incrementally redesign legacy systems as part of a modernization process. Microservices promote better reuse and customization of existing features while increasing business capabilities, if appropriate design decisions are made. There are some partially-automated approaches supporting the re-design of legacy features into microservices. However, they fail in covering two key aspects: *(i)* provide an architectural design of the features being redesigned, and *(ii)* simultaneously support relevant criteria, *e.g.*, feature modularization and decrease of network communication overhead. Also, these two aspects tend to be poorly discussed along industrial case studies. To fulfill these gaps, we propose a redesign strategy to support the re-engineering of features legacy code as microservices. This strategy covers key possibly-conflicting criteria on microservice-based architectures. We employ search-based optimization to deal with such conflicting criteria. The output of the strategy is a set of redesign candidates of legacy features as microservices. We reflect upon the benefits and drawbacks of the proposed strategy through an industrial case study. In particular, we perform an in-depth analysis of the resulting microservice candidates, and a discussion about their potential for customization and reuse. The reflections/discussions are also supported by observations of developers involved in the process.**

***Index Terms*—microservice architecture, legacy systems, software evolution, search-based software engineering.**

I. Introduction

Despite being often outdated or obsolete, legacy systems represent a massive and long-term investment in organizations regardless of their business domain. Modernization processes have often required the redesign of legacy systems into microservice architectures [1, 2, 3, 4, 5]. A microservice is an autonomous small service that communicates through lightweight protocols [6, 7]. Microservices can modularize existing features [8], otherwise tangled in the legacy code, and make them available as new business capabilities through the network [6, 7]. To this end, maintainers have to decide on *which* and *how* existing features in the legacy system should be redesigned as microservices. The redesign process consists of both identifying and analyzing the existing structure of each legacy feature – often structurally degraded – in order to define a new design structure, *i.e.*, the resulting microservice-based redesign. However, redesigning legacy features into microservices is a quite complex activity for several reasons [9].

The redesign activity should guide: *(i)* the identification of features (and possibly subfeatures) that are likely to be aligned with the business capabilities, *(ii)* the proper modularization of such features into microservices, *(iii)* the simultaneous satisfaction of influencing criteria, such as cohesion and network overhead [10], *(iv)* the generation of alternative designs with different levels of microservice decomposition – alternatives are essential to support maintainers in choosing the one that fits best their needs, and *(v)* the redesign of features that are likely to be reusable and customizable.

There are some partially-automated approaches for supporting the redesign of existing systems into microservices [11, 12, 13, 14] (Section II). However, they are not feature-driven, *i.e.*, developers cannot select an initial set of features to be redesigned as microservices. These approaches also do not support the further decomposition of candidate features into subfeatures, which improve modularization and the alignment with new business capabilities. There are several approaches for feature modularization [15], but they often support only one or two criteria [16, 17], and do not master microservice-specific criteria, *e.g.*, network overhead. Moreover, the evaluation of such approaches does not cover the five guidance needs mentioned above and also neither consider industrial systems, in which access to actual developers is possible.

To address these gaps, we propose a semi-automated strategy for assisting software engineers along with microservice-based redesign of legacy features (Section IV). Here, redesign encompasses the identification of boundaries in the source code, where the legacy system should be split in microservices. Our strategy is based on: *(i)* both static and dynamic analyses of the legacy code, and *(ii)* search-based multi-criteria optimization to deal with four possibly-conflicting criteria to be satisfied while redesigning features as microservices. We reflect upon the benefits and drawbacks of the proposed strategy through an industrial case study (Section III). In particular, we perform an in-depth analysis of the resulting redesign candidates and discuss their potential for feature customization and reuse (Section VI). Such reflections are also based on observations of developers of the legacy system.

In summary, our contributions are as follows:

- A feature-driven strategy for microservice identification based on optimization of four criteria.
- A qualitative empirical study demonstrates that all interviewed developers agree that the generated candidate microservices are aligned with the business capabilities.
- Experimental evidence reveals that reuse and customization opportunities can be better achieved with our strategy. For example, developers observed three types of customization opportunities (interface customization, microservice specialization, and disabling of a microservice) and envisioned reuse opportunities of four microservices.

II. Related Work

Several studies report on empirical evidence about the benefits of migrating legacy systems to a microservice architecture [1, 2, 3, 4, 5, 18, 19]. In spite of known benefits, existing approaches are still at an early stage as they provide very limited support. They do not focus on the entire redesign process to achieve a microservice architecture. Besides, the evaluation of these approaches are often conducted with small toy-example applications. They are not based on case studies with access to the developers of industrial systems under migration. Finally, existing approaches are mostly based on only one or two redesign criteria and, as they are not feature-driven, it is not possible to select which features should be redesigned as microservices, hampering both feature modularization and alignment with new business capabilities. Existing approaches cannot be easily adapted to be feature-driven because it is necessary to modify either their input or redesign criteria.

Nunes *et al.* [20] and Henry & Ridene [9] identify microservices based on domain concepts derived with the use of clustering algorithms. However, microservice identification based only on domain boundaries of the legacy system often leads to an explosion of microservices [9]. Many of these microservices are not aligned with the business capabilities of the organization. Moreover, these approaches solely present an illustrative example from an online shop application. Similar to our work, some approaches [11, 12, 13, 14] use search-based algorithms, considering one or two criteria, limited to coupling and cohesion. However, cohesion in a module (*e.g.* a class) code differs from cohesion in features, *i.e.*, the feature code can be scattered and tangled across program modules. Thus, these approaches are not feature-driven and do not consider network overhead – a relevant criterion for microservice architectures.

Li *et al.* [21] propose a dataflow-driven approach to identify microservices. Their approach was also evaluated on an industrial case study, but they only consider coupling and cohesion metrics. Pigazzini *et al.* [22] and Megargel *et al.* [23] propose a similar approach. Interestingly, Megargel *et al.* [23] mentioned reuse as post-migration benefits as they could compose new products without creating new microservices. However, their process was evaluated only in an illustrative system. On the work of Maisto *et al.* [24], the identification of microservices is strictly based on coupling between classes and no evaluation is performed. Overall, these approaches rely only on static analysis of the legacy code and are not feature-driven.

Tizzei *et al.* [4] reported on a case study of migrating a legacy system to a microservice-based product line to minimize maintenance efforts, support scalability, independent deployment, and configurability. However, the migration was performed manually, not documented in detail, and based on experts' knowledge. Silva *et al.* [25] propose a feature-driven process, but it was also a manual and expert-dependent. Still, the evaluation was conducted on a small eShop system.

In summary, our work has a complementary nature to previous research by including: *(i)* a case study with an industrial legacy system; *(ii)* interview with developers to define the set of relevant criteria for generating microservices; *(iii)* the use of well-defined criteria from the field of microservices; and *(iv)* a qualitative analysis of the microservice candidates from the reuse and customization perspectives.

III. Case Study

Our case study relies on a legacy system developed and maintained at Tecgraf Institute, an institute responsible for providing software products to oil and gas industries. This legacy system is predominantly written in Java and has been maintained for more than 15 years. The system provides a shared environment for different users, *e.g.*, engineers and scientists, to maintain and execute complex algorithms and share their results. These results include algorithm executions, algorithm configurations, binaries, to cite some.

In its inception, the legacy system was designed to be a framework that allows customization to different customers. However, after many years of maintenance, its architecture degraded. Currently, the system exhibits limited feature modularization, *i.e.,* high tangling and scattering of features. Furthermore, it has scarce documentation, which consists of few diagrams of the old architecture and some documented APIs. Developers reported that its maintenance is very complex and time-consuming. There is a limitation to incorporate new technologies such as the use of cloud computing resources.

The legacy system originally was developed to be used in a private network, with on-premise installations and a limited number of users. However, about three years ago, the customers asked for new requirements such as making the system available on the internet, providing a web interface, and using a cloud infrastructure. Based on that, the institute decided to redesign the legacy system into microservices. In a first attempt, the developers tried to perform a manual analysis of the source code to identify possible features to be extracted into microservices by considering three main criteria: coupling, cohesion, and feature modularization. However, this manual analysis required a high effort from developers due to system complexity and size. To reduce effort, visual tools were used to represent classes and their relations. The developers also gave up this strategy as the system presents poor modularization, *i.e.*, large class and feature scattering, hindering the understanding of microservices boundaries.

Our study focus on three major features, which are related to the business capabilities of high value for clients. These features have at least two subfeatures, 180 classes and 1,038

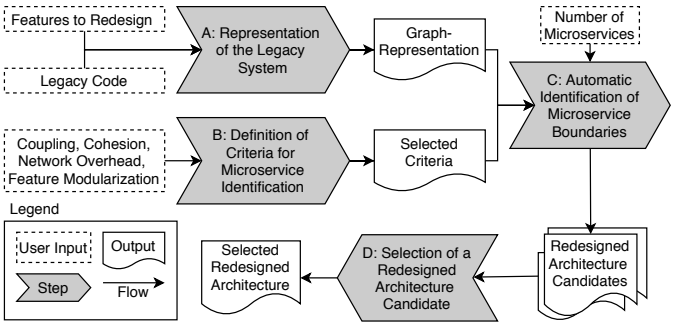


Fig. 1. Overview of the Proposed Redesign Strategy

methods associated with their implementation in the legacy code. Each feature provides the operations and data as follows:

Authentication: verifies the identity of the users. This feature includes the creation and validation of tokens, verification of login and password, update of password, and related simple information about the system's users. Source codes related to this feature are used extensively by almost the entire system for validations and information retrieval.

Algorithm: manages algorithms information, including parameters, binary, documents, and connection points with other algorithms. Also, this feature store algorithm's output.

Project: provides a collaborative environment among the system's users to share projects and their metadata.

IV. Proposed Redesign Strategy

In the context of our work, *redesign is the process of identifying and analyzing the structure of existing features of a legacy system in order to define a new microservice-based design of those features*. Redesigning features as microservices requires considering some aspects related to the structure of the legacy system and aspects of the new architectural style, namely the microservice architectural style [26]. The existing features may be further decomposed/modularized in subfeatures as microservices. In this context, we propose a strategy composed of four steps, presented in Figure 1.

The proposed redesign strategy requires as *input (i)* a list of features implementing business capabilities desired to be migrated to microservices; *(ii)* the source code of the legacy system; *(iii)* criteria available for the identification of microservices; and *(iv)* the number of desired microservices to be generated. The first two pieces of information are used in *Step A*, the third in *Step B*, and the last one in *Step C* of the strategy. The user in *Step A* may also specify a set of functional test cases related to the features taken into consideration. As our strategy is multi-criteria, *Step B* requires the selection of a subset of the four criteria to be used during the automatic identification of microservice candidates. The *output* of *Step D* is a redesign candidate that is an architecture where each legacy feature is represented by a microservice or group of microservices. The architecture also describes dependencies among microservices, which also represents network communication paths. The four steps of the redesign strategy are described next.

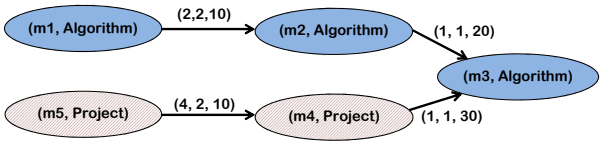


Fig. 2. Graph-based representation of the legacy (method name, feature label)

A. Representation of the Legacy System

This step is responsible for representing the legacy system in a way to enable the automatic identification of microservices. We adopt a generic representation that allows our strategy being applied in any legacy system. To allow a fine-grained analysis of the legacy system, we choose a representation at the method level. By using a method level representation, our strategy can reach a better feature modularization, since legacy systems often have very large files where many features are tangled to each other.

The generic representation adopts a graph-based structure. Each vertex represents a method of the legacy system mapped to the respective feature(s) it realizes. Each edge represents the relationship between methods, describing estimated communication, syntactic and dynamic calls between them. More specifically, the edges are a tuple $e = (static\_calls, dynamic\_calls, data\_traffic)$. The information can be collected from the legacy by using code analysis tools that explore both static and dynamic characteristics. The edge between two vertices v_i and v_j exists only when there is at least a syntactic call in the v_i method body to the method v_j. Figure 2 depicts an excerpt of the graph-based representation. The graph has five vertices, each representing a method in the legacy system under analysis. The vertices $m1$, $m2$ and $m3$ are part of the feature Algorithm, whereas the vertices $m4$ and $m5$ implements the feature Project. The edge $(2, 2, 10)$ describes the relationship between $m1$ and $m2$. The first 2 is the number of calls from $m1$ to $m2$ found in the source code. The second 2 is the number of calls observed when exercising the legacy. Finally, 10 is an estimation of the size of data exchanged between the methods. Section V-B presents details on how we construct the graph for our case study.

B. Definition of Criteria for Microservice Identification

To define the set of relevant criteria for microservice identification, we rely on findings from a previous survey and interviews with experienced practitioners [10, 27]. This study identifies the set of criteria practitioners have adopted to extract microservices. Their criteria are based on three different perspectives: *(i)* traditional criteria to evaluate the actual structure of the legacy system, namely *coupling* and *cohesion*; *(ii)* a microservice-based criterion that estimates the *network overhead* that will be created when dependent features are migrated to different microservices; and *(iii)* a criterion based on feature modularization to guide the redesign of customizable microservices.

Next, we present in detail each of these criteria that composes our evaluation model for the redesign of features as

microservices. For that, let us consider MS_c as a microservice candidate and RCA a redesigned architecture candidate.

1. Coupling: this criterion measures the coupling between microservices. The function $\delta(MS_c)$, presented in Equation 1, computes the number of static calls from methods within a MS_c to the other microservices in RCA or parts of the legacy system. The coupling of MS_c is the sum of the number of static calls, computed by sc, from the methods in v_i (that belong to MS_c) to methods in v_j (that does not belong to MS_c). The total coupling of RCA is the sum of the values of coupling associated with every MS_c in a RCA as described in Equation 2. The lower the coupling better.

$$\delta(MS_c) = \sum^{v_i \in MS_c \wedge v_j \notin MS_c} sc(v_i, v_j) \quad (1)$$

$$Coupling(RCA) = \sum^{\forall MS_c \in RCA} \delta(MS_c) \quad (2)$$

2. Cohesion: this criterion represents how strongly related are the methods within a microservice candidate. Cohesion is computed by dividing the number of the static calls between methods within the microservice boundary (the set of methods assigned to MS_c) by all possible existing static calls. The auxiliary boolean function ce (Equation 3) verifies the existence of at least one static call. Then, the cohesion of MS_c is computed by $C(MS_c)$. This function divides the number of static calls by the number of all possible dependencies between methods of a candidate microservice, where $|MS_c|$ is the cardinality of MS_c. The denominator is the combination two-by-two of all methods within a MS_c. The total cohesion of RCA is the sum of the cohesion associated with every MS_c, according to Equation 5. The higher the cohesion better.

$$ce(v_i, v_j) = \begin{cases} 1, if sc(vi, vj) > 0 \\ 0, \ otherwise \end{cases} \quad (3)$$

$$C(MS_c) = \frac{\sum^{\forall v_i \in MS_c \wedge v_j \in MS_c} ce(vi, vj)}{\frac{|MS_c|(|MS_c|-1)}{2}} \quad (4)$$

$$Cohesion(RCA) = \sum^{\forall MS_c \in RCA} C(MS_c) \quad (5)$$

3. Network Overhead: Some non-functional requirements associated with a feature may be negatively affected by the network communication overhead when such a feature is modularized as a microservice. To minimize this issue, we created a heuristic that uses dynamic information to estimate the network overhead. The heuristic uses the size of the objects and primitive types in the parameter list between methods, collected during the execution of the legacy system. In addition, the heuristic considers the overhead caused by the protocol adopted for the future migrated microservices. For example, the HTTP protocol adds a header to each call and, therefore, the size of this header is considered in our heuristic.

The network overhead computation of a method (a vertex) is presented in Equation 6, where the function $P(v_j)$ returns the list of parameters used in the execution of the method v_j. The function $sizeOf(p, m)$ is the size of the p^{th} parameter in the m^{th} call from v_i to v_j. The function dt in Equation 7 computes the data traffic, where dc function is the total of calls from method v_i to method v_j in execution time. The network overhead of MS_c (Equation 8) is the sum of all data traffic within their methods, and the network overhead of RCA (Equation 9) is defined as the sum of the network overhead for every MS_c. The lower the network overhead better.

$$overhead(v_i, v_j, m) = \sum^{\forall p \in P(v_j)} sizeOf(p, m) \quad (6)$$

$$dt((v_i, v_j)) = max_{m=1}^{m=dc(v_i, v_j)}(overhead(v_i, v_j, m)) \quad (7)$$

$$O(MS_c) = \sum^{\forall v_i \in MS_c \wedge \forall v_j \notin MS_c} dt((v_i, v_j)) \quad (8)$$

$$Overhead(RCA) = \sum^{\forall MS_c \in RCA} O(MS_k) \quad (9)$$

4. Feature Modularization: we propose this criterion to optimize the responsibility of microservice candidates. The notion of *predominant feature* was created to indicate the occurrence of the feature that most occurs in the vertices (methods) associated with MS_c. This notion of predominant feature is used to minimize the number of distinct features per microservice. The goal is to reduce the tangling of features and improve microservice structure with a single responsibility, one of the best practices of microservice design [6]. Equation 10 defines pf that computes the number of predominant features of MS_c, where F_{MS_c} contains the number of occurrences of each feature K in MS_c. The feature modularization of MS_c is computed as the maximum number of the features' occurrences divided by the sum of all features' occurrences (Equation 11). The feature modularization of RCA, defined in Equation 12, is the sum of feature modularization for every MS_c plus the number of distinct predominant features in the $|FRCA|$ divided by the number of microservice $|RCA|$. $|FRCA|$ is the set of distinct features in the RCA. This division aims to avoid a separation of the same feature by different microservices candidates. A degree of feature modularization should be as high as possible.

$$pf(MS_c) = max_{\ \forall k \in F_{MS_c}}\{k\} \quad (10)$$

$$f(MS_c) = \frac{pf(MS_c)}{\sum^{\forall k \in F_{MS_c}}\{k\}} \quad (11)$$

$$F(RCA) = \sum_{\forall MS_c \in RCA} f(MS_c) + \frac{|FRCA|}{|RCA|} \quad (12)$$

C. Automatic Identification of Microservice Boundaries

With the focus on understanding the benefits of existing criteria to identify microservices, we conducted a preliminary experiment [28]. In this evaluation, we concluded that the criteria of coupling, cohesion, feature modularization, and network overhead are relevant, interdependent, and conflicting.

Due to the interdependence and conflicting nature of the criteria presented in Section IV-B, this step relies on a many-objective search-based approach, adopting the Non-dominated Sorting Genetic Algorithm III (NSGA-III) [29]. NSGA-III is based on a genetic algorithm, which is a method for solving optimization problems that reflect the process of natural selection [30]. This algorithm repeatedly selects the fittest individuals for reproduction in order to produce offspring of the next generation. Over successive generations, the individuals evolve toward a set of optimal solutions. Next, we provide details about the search-based approach of our strategy.

Representation of Solutions: the graph-based representation described in Section IV-A is used in this step. A redesigned architecture candidate is composed of groups of vertex in the graph that represent microservices (Figure3). The maximum number of microservices and the range of vertex (methods) allocated in each microservice is provided as input by the software engineer (see Figure 1). The population size of solutions for NSGA-III is set to 50 individuals.

Fitness functions: we used the evaluation model described in Section IV-B to assess each individual/solution during the evolutionary process of NSGA-III. Each criterion was represented as an objective in the evaluation model. Thus, the redesign of features as microservices was defined as a four-objective optimization problem. For the evolutionary process (Step C), all the objectives were designed for minimization. Despite cohesion and feature modularization being defined in the evaluation model as maximization, they were inverted during the fitness computation. The maximum number of fitness evaluation (stopping criterion) was set to 30,000.

Genetic Operators[1]: for the *selection operator*, we adopt the binary tournament as a strategy to select individuals (solutions) to apply the genetic modifications. Thus, a set of individuals is randomly selected from the population, from which the individual with the best fitness is chosen to undergo mutation [32]. The *mutation operator* consists of moving methods from one microservice candidate to another one in an individual of the population [13, 14]. In a simplified form, this operator can be seen as an analogy of the move method refactoring [33]. The fraction of methods to be moved is configurable. The methods and microservice candidates are randomly selected. The mutation rate of NSGA-III is set to 0.4.

D. Selection of a Redesigned Architecture Candidate

The output generated in the previous step is a set of solutions (redesigned architecture candidates), each one with different trade-off among the criterion values. Each solution is a group of vertices that represents a microservice candidate, which implement feature(s). In this case, the software engineers will need to choose which solution fits better their needs. For example, they can select the solution that most reduces network overhead, or most increases feature modularization. In case they want a solution that represents the best trade-off among the objectives, we propose the use of the indicator named Euclidean Distance to the Ideal Solution (ED). ED is used to find the closest solution to the best theoretical objectives, *i.e.*, an ideal solution [34]. Since NSGA-III was implemented to deal with all objectives as a minimization, an ideal solution has a value equal to 0 for all objectives.

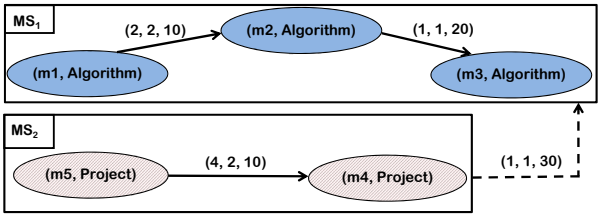


Fig. 3. Microservice candidates for the excerpt of Figure 2

To illustrate a possible output, we recall the example of input presented in Figure 2. Suppose that considering the ED indicator we chose the microservice architecture presented in Figure 3. In this example, $m1$, $m2$, and $m3$ are part of the microservice MS_1 , and $m4$ and $m5$ are included in the microservice MS_2. In this case, communication between MS_1 and MS_2 will become the communication between $m3$ and $m4$ by the network. This simplified solution in Figure 3 presents an adequate solution under the point of view of the criterion of feature modularization. This can be observed since each microservice candidate has a single feature, which represents a functional requirement in the system under analysis. The MS_1 modularizes only the feature Algorithm, while the MS_2 modularizes only the feature Project. Moreover, the solution presents a higher cohesion value and lower coupling value in the identification of two microservices to the representation graph. However, this solution has not the optimal value for the network overhead criterion.

This simple example illustrates the impossibility of finding an optimal solution to the four criteria as happens for the vast majority of systems under analysis. In this order, our strategy generates a Pareto set (common in multi-criteria optimization) with several solutions. In the graph representation introduced in Figure 2, the solution where the methods m_1 and m_2 are in one microservice candidate and the additional methods are in another microservice candidate contains the lower values to network overhead even if at the expense of the other criteria. This solution and that one presented in Figure 3 forms a potential Pareto set generated by our strategy.

V. Evaluation Setup

This section defines the empirical study conducted to evaluate different characteristics of our strategy, relying on an industrial case study described in Section III.

[1]It is difficult to guarantee the accuracy and consistency of architectural designs after the crossover operator application [31]. To avoid this problem, we rely exclusively on the use of the selection and mutation operators.

A. Research Questions

Our work addresses the following four research questions:

RQ1. Are the modularization of features as microservices aligned with the business capabilities of the case study? It is important to verify whether the redesign candidates modularize the legacy domain's features as expected. To this end, we ask developers to identify the predominant feature in each microservice candidate. They check whether they represent business capabilities (see Section IV-A).

RQ2. What are the most relevant criteria for the identification of microservice candidates in the case study? This question aims at analyzing the criteria most used by practitioners to evaluate the redesign candidates (see Section IV-B). To support our analysis, we explore the expertise of developers during the selection of a redesigned architecture.

RQ3. How do the inputs to our search-based approach impact the redesign of features? Here we discuss how the inputs used by our search-based approach impact the solutions generated in Step C (see Section IV-C).

RQ4. Which customization and reuse opportunities were identified for the microservice candidates? In this question we evaluate whether the developers identify customization and reuse opportunities for microservices of the redesign candidates. Reuse is a benefit expected from the adoption microservices. Customization is required to meet user needs.

B. Data Extraction for the Legacy System Representation

We developed an extractor to collect data and construct the graph of the legacy system in our case study. We describe below how the extractor works. However, for other legacy systems implemented in different programming language, the data can be extracted differently. In our study, the input was the source code, the list of features to redesign with corresponding entry points to their source code, and the functional test cases related to these features. These pieces of information were provided by an expert in the legacy system.

Feature label in the vertices: To map the features to their source code, we adopted an automatic strategy based on execution traces. This strategy executes functional test cases, to label the vertices with the names of the features they implement. This strategy uses code elements as entry points to associate features with execution traces. Each entry point defines one of the possible entries to the feature boundaries. The boundaries consist of a set of methods that directly interact with methods of other features, *i.e.*, delimit the feature scope.

Basically, an entry point is a relationship between a regular expression and a feature. Each regular expression is compared with patterns in the names of packages, classes, or methods in the execution trace. The regular expression can provide a match with methods of the execution trace by some name patterns. In the match, the method in the execution trace is labeled in graph vertex with the related feature. Each method in the execution trace that is not an entry point is labeled with the feature of the last entry point (lower depth number).

For example, Listing 1 presents a simplified trace execution whereas Listing 2 shows entry points to Algorithm and Project. In the analysis, the method `Algorithm.getAdminIds` is matched with the regular expression associated with the feature Algorithm, then the method is an entry point and labeled with this feature. The next methods with higher depth are also labeled with the feature Algorithm up to the point the method `ProjectService.getAllProjects` is reached. This method matches with the regular expression of the feature Project. Thus, `ProjectService.getAllProjects` is labeled with the feature Project as well as `ProjectInfoService.getInfo` as it has a higher depth. Finally, the method `Algorithm.algorithmsToVector` is labeled with the feature Algorithm because of the lower depth as an entry point is Algorithm and not Project.

Listing 1. Execution trace example

```
Name:Algorithm.getAdminIds#Depth:12
Name:Algorithm.getAllAlgorithms#Depth:13
Name:Algorithm.loadLocalAlgorithmCache#Depth:14
Name:Algorithm.getPermission#Depth:13
Name:AlgorithmPermission.getAllPermissionIds#Depth:14
Name:AlgorithmPermission.getPermissionIds#Depth:15
Name:ProjectService.getAllProjects#Depth:16
Name:ProjectInfoService.getInfo#Depth:17
Name:Algorithm.algorithmsToVector#Depth:14
```

Listing 2. Entry points of the features Algorithm and Project

```
Algorithm<Algorithms.getAdminIds>
Project<ProjectService.*>
```

Relationship information in the edges: In addition to labeled vertices, edges also contain information required by the optimization performed by the genetic algorithm. Static calls between methods were extracted from the system under analysis by the tool depfinder [35]. Regarding the dynamic information, we created an extractor that injects code in each method of the program under analysis before its compilation. The injected code is responsible to count how many times has a method been called during the execution. Moreover, the injected code also computes the size of the parameters' data provided to each method associated with each call. This computation considers the sum of all primitive types in each parameter, *e.g.*, the primitive types stored in an object passed as parameter. In summary, the size of each primitive type is summed based on the size provided by the virtual machine.

C. Experiment Configuration

To evaluate the proposed strategy, we relied on a legacy system of Tecgraf Institute as input for Step A. The list of features was: Authentication, Algorithm, and Project (see Section III). An author of this paper is the project manager of the legacy system, which made its source code and test cases available for the evaluation. Another author is an expert on the case study and provided the entry points for each feature.

For Step B, we adopt the evaluation model with four criteria (see Section IV-B). For Step C, we configured three different experiments to further investigate how the features were redesigned in the generated solutions, as follows:

- *Exp–5MS*: number of desired microservices = 5, number of methods for each microservice between 6% and 32%.
- *Exp–7MS*: number of desired microservices = 7, number of methods for each microservice between 4% and 24%.
- *Exp–10MS*: number of desired microservices = 10, number of methods by microservice between 3% and 16%.

To have significant information for Step D, we executed 30 independent runs of each experiment. After the computation of all solutions, we considered the indicator ED to select one redesign architecture candidate for each experiment, which was the basis for the qualitative evaluation presented next.

D. Qualitative Evaluation with Developers

In the qualitative study we interviewed eight developers of the legacy system and inquired their opinion about some generated solutions. During the interview, each developer assessed the microservices of each solution regarding their main features. They are mostly experienced developers, with a median time experience in software development of 12.5 years. Regarding their experience with the legacy system, we have both experienced developers, with experience of 8, 13 and 20 years, and recent developers, with experience between 0.5 and 3 years. Seven participants are developers of the legacy system and one is a team leader. All data collected were analyzed by three of the authors. Details are available in [36].

VI. Results

The part of the legacy system given as input for the redesign strategy has a total of 180 classes, where 76 of them (42%) have pieces of code that realize more than one feature. Yet, the implementations of Algorithm, Project and Authentication are diffused in 100, 85 and 65 classes, respectively. These numbers reinforce how difficult is a manual identification of microservices taking into account the four criteria (Section IV-B).

The generated solutions have different compromises among the four criteria. Figure 4 presents the fitness of each solution for the set of non-dominated solutions obtained by Exp–10MS considering the 30 independent runs. In this figure, we can observe that coupling and cohesion are conflicting measures (yellow and black lines). These two measures seem to increase/decrease inversely proportional. On the other hand, coupling resembles network overhead in half of the cases. This is an expected result: when there is more coupling, more is the communication among microservices, and vice-versa. But we can also observe that this relation is not proportional. It seems that depending on the solution, it can have very low coupling and still lead to high overhead (red lines).

An interesting observation is that structural cohesion is not strictly related to feature modularization. Some studies on migrating legacy systems to microservices mention that structural cohesion is a guiding criterion for this process [6, 10]. However, we can observe that cohesion in classes code does not lead to cohesion in features, *i.e.*, feature modularization. As far as exclusive adherence to the criterion of cohesion is concerned, we might violate the principle of single responsibility. This means that we may have cohesive microservices, but

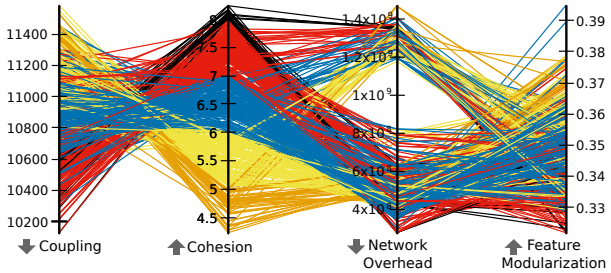


Fig. 4. Non-dominated solutions for experiment *Exp–10MS*

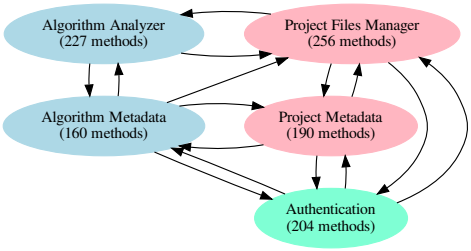
that leads to tangled features, which would further harm the migration and future maintenance and evolution. In addition, tangled features makes DevOps adoption complex, one of the most appealing benefits of migrating to microservices [37].

To qualitatively analyze the redesign candidates found by each experiment, we picked up the solution with the lowest ED. They represent the solutions that have the best trade-off among the objectives, following the recommendation of Step D (see Section IV-D). Figures 5(a), 5(b) and 5(c) depict the graphs that represent these solutions. We analyzed the methods that constitute each microservice candidate and assigned a name for it. The name is associated with the predominant feature (or subfeature) realized by the microservice. These names were validated by the developers.

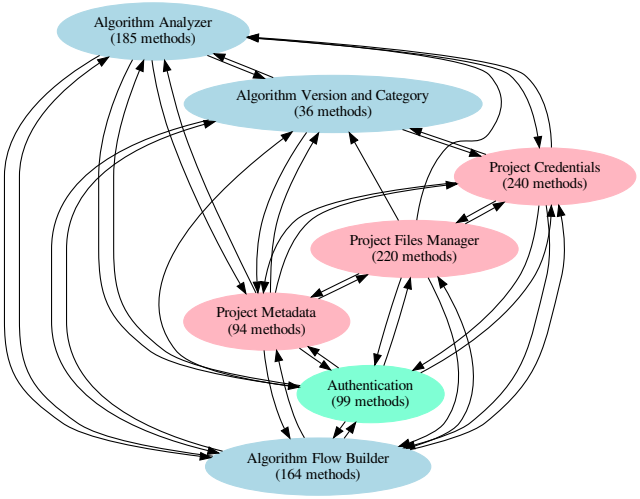
The features are modularized in different ways in each solution. The solution of Exp–5MS (Figure 5(a)) has coarse-grained microservices as the feature Algorithm was modularized in two microservices: *Algorithm Analyzer* and *Algorithm Metadata*. The same applies to the feature Project, resulting in *Project Files Manager* and *Project Metadata*. The last microservice realizes the feature Authentication. On the other hand, solutions of Exp–7MS and Exp–10MS have fine-grained microservices. Taking into account the solution of Exp–7MS, the feature Algorithm was modularized in *Algorithm Analyzer*, *Algorithm Flow Builder*, and *Algorithm Version and Category*. In this solution, there are three microservices to deal with Project, namely *Project Metadata*, *Project Credentials*, and *Project Files Manager*. The latter is a microservice for *Authentication*. In comparison with the solution of Exp–7MS, the solution of Exp–10MS has one more microservice for Algorithm (*Algorithm Parameter Validator*), an additional microservice of Authentication (*User*), and an exclusive microservice for logging (*Server Log*) that in the other redesign candidates was scattered in other microservices. We observe that the microservices related to Algorithm and Project of Exp–5MS and Exp–10MS are more related to each other than in Exp–7MS. The former solutions have lower coupling between microservices that realize different features.

A. RQ1. Microservices Aligned with Business Capabilities

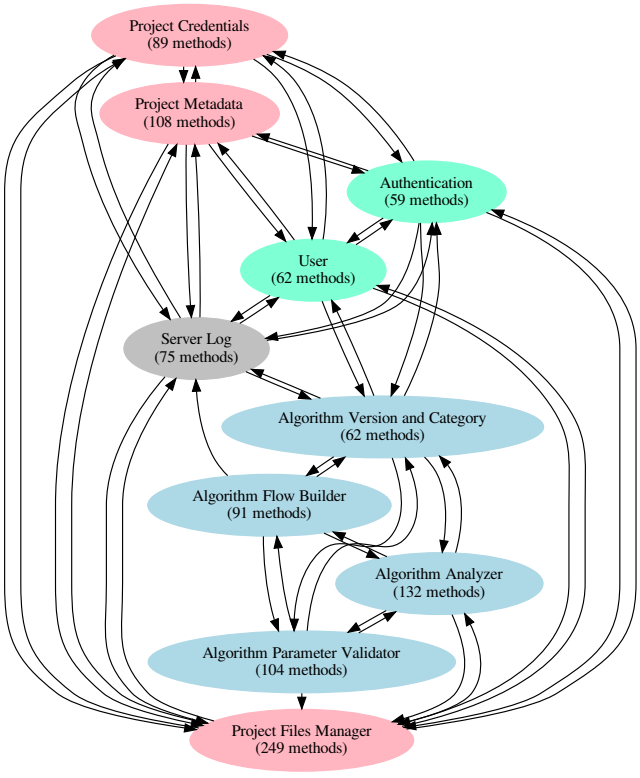
We asked eight developers to check whether they recognize the microservices that represent business capabilities. To this end, they inspected the source code of the microservices of the redesign candidates. Interestingly, regardless of the developers' experience in the legacy system, all participants could identify



(a) Solution of *Exp–5MS*



(b) Solution of *Exp–7MS*



(c) Solution of *Exp–10MS*

Fig. 5. Selected solutions for each experiment. The edge values for syntactic calls, dynamic calls, and data traffic were omitted to improve readability.

the predominant features. For the sake of illustration, one participant (P6) argued: *"It is a microservice of Algorithm that writes both the algorithm and its corresponding files."*

Another participant (P7) observed that *"one microservice is highly-cohesive in the Project logic despite addressing two functionalities of Project: Project File Manager and Project Metadata"*. This statement also showed that the participants recognized new features (including subfeatures), which were not previously informed as known by them. Another statement, by participant P4, clearly addresses this fact: *"It is possible to identify that this microservice focuses on Algorithm. However, there are several methods associated with Project Credentials. This microservice could be smaller, extracting the logic related to the Project Credentials subfeature and delegating it to another microservice."* As expected, the participants with more experience in the legacy system (P4, P5, P6, P8) recognized more new features in general than recent developers. Some subfeatures identified by the participants include: *Project Credentials*, *Project Metadata*, *Project Files Manager*, *Algorithm Flow Builder*, and *Algorithm Parameter Validator*.

Taking into account the solutions presented in Figure 5, we could notice that Exp–7MS and Exp–10MS have microservices realizing some new subfeatures identified by the developers. The activity of evaluating the generated solutions allows the participant to reason about the features and their subfeatures; and they agreed with the predominant feature modularized in each microservice. This means that our automatic strategy based on execution traces is properly mapping the features to their respective legacy methods. The only exception is related to the microservices whose granularity is different from the developer's preference (see Section VI-B).

> ***RQ1:*** *The features and subfeatures modularized in the microservice architecture solutions are aligned with business capabilities as they are related to (i) the features informed as input to the strategy, and (ii) the subfeatures recognized by developers during the evaluation.*

B. RQ2. Criteria Relevance for the Microservice Identification

We also analyzed which criteria the developers have considered during the evaluation of the solutions obtained by our strategy. Most developers have considered all four criteria. Feature modularization, coupling and cohesion were mentioned by all the developers. Network overhead was considered by three developers. For instance, when asked to identify the feature realized by a specific microservice, P8 stated *"This microservice is related to Project. It is highly-cohesive and has low communication overhead."* Other participant (P7) also mentioned cohesion and overhead: *"This microservice is cohesive regarding Algorithm; however, it includes Algorithm Flow Builder that makes this microservice large and affects the communication overhead."* Interestingly, this corroborates the results of an existing survey that suggests that at least four criteria are often considered simultaneously as useful to support the identification of microservices [10].

In addition to the criteria investigated in previous studies, other criteria were considered by the developers, albeit much less frequently. They are related to team size and developer knowledge, or quality requirements, such as performance, security, maintainability and interface complexity. Team size and business were considered by two developers whereas the other ones were considered once.

> ***RQ2:*** *The set of criteria adopted in our strategy are often considered as relevant and useful by the developers of the legacy system.*

C. RQ3. Impact of the criteria on the Redesign of Features

This RQ addresses the impact of the criteria used during the evolutionary process, *i.e.*, the Step C of the proposed strategy. Therefore, we analyze the fitness value of each selected solution (see Table I). The values of coupling and cohesion indicate conflict between them as one can notice in the solution of Exp–5MS. This solution is highly cohesive and loosely coupled, whereas the solution of Exp–10MS has low cohesion and high coupling. These values are due to the different levels of microservice granularity as previously discussed. The solution of Exp–5MS might be the ideal solution in terms of coupling and cohesion. However, given its coarse-grained microservices, the value of network overhead is extremely high when compared to other solutions.

Another dimension of the evaluation model is the degree of feature modularization into microservices. The solution of Exp–10MS has the lowest value of feature modularization as well as the lowest values of cohesion. In this solution, 60% of the microservice candidates realize at most two features. Also, 12% of the methods have pieces of code realizing more than one feature, recalling that a method can be mapped to more than one feature. The solution of Exp–7MS has 24% of methods realizing more than one feature and six microservice candidates deal with three features. The solution of Exp–5MS has only 9% of its methods realizing more than one feature. For instance, the microservice candidates associated with Algorithm also deal with another feature. The microservice candidates associated with Project also have methods realizing Algorithm and Authentication. These details impact on the value of the fitness function Feature Modularization.

An interesting observation is the optimal modularization of the microservice entitled *Algorithm Flow Builder*, which deals only with this subfeature. The counterpart of this microservice in the solution of Exp–7MS deals with 3 features and around 24% of its methods realize more than one feature. We also noticed that Authentication, which is a naturally crosscutting feature, is a highly-coupled microservice (Figure 5(b)). Despite the difficulty of modularizing crosscutting features, our strategy found alternatives to redesign the feature Authentication. We observed that the microservices whose predominant feature is Authentication have the greatest values of feature tangling: 55% of the methods realize more than one feature.

TABLE I
FITNESS OF THE SELECTED REDESIGN CANDIDATES

Solution	Coupling	Cohesion	Network Overhead	Feature Modularization
Exp–5MS	5789.0	5.85	8.53	0.27
Exp–7MS	8397.0	5.34	1.25	0.24
Exp–10MS	8730.0	1.82	1.46	0.15

Another point of view that impacts the redesign of features is the granularity level of microservices, which is defined by the number of desired microservices given as input to our strategy. Considering the three selected solutions, one can argue that the solution of Exp–10MS is the best, as it has the lowest values for cohesion and feature modularization (Table I). However, the comprehension of what is a small microservice varies according to the developer or organization preferences, as we could observe when interviewing the developers of our case study. The redesign candidates obtained by the different experiments pointed out that our strategy is flexible. They can be easily adapted to fit a desired granularity level of preference. The developer only needs to inform the number of microservices to be extracted from the legacy code. Then, the features will be incorporated in the redesign candidates accordingly, *i.e.*, in a coarse or fine granularity.

The divergent preferences regarding the granularity of microservices can be illustrated by quotes of some participants. One developer (P8), who clearly prefer fine-grained microservices, argued about the microservice *Authentication* of Exp–5MS: *"The cohesion of this microservice is very low. It has methods to realize distinct functionalities: Algorithm, Project and User. It performs a lot of non-related operations"*. On the other hand, participant P6 recognized that the predominant feature of a microservice is *Project Files Manager* but stated *"I would add elements of Project Credentials to make the microservice more autonomous and reduce the communication overhead"*. For other participants some microservice candidates have the expected granularity: *"This microservice has methods related to the same objective what allows greater reuse. It is possible to identify that the main functionality is the management of project files" (P4)*.

Another important point to be considered is that the method names might not represent their behavior. One developer (P1) argued: *"It is complicated to trust that the method does what its name indicates. In legacy code, as ours, certain identifiers were defined 20 years ago and they were never renamed even after several modifications"*. This fact strengthens that solely static techniques for feature location (avoided in our strategy) indeed suffer due to the bad quality of the method names in the legacy code as well as endorse our option for a feature-to-code mapping combining dynamic and static analyses.

> ***RQ3:*** *The obtained redesign candidates composed by microservices with different granularity levels show that our strategy is flexible and able to generate solutions according to the developers' needs and preferences. The results evidence that the solutions generated by our strategy allow restructuring features to be smoothly migrated to a microservice architecture.*

D. RQ4. Customization and Reuse Opportunities

Customization opportunities were revealed during the evaluation, as shown by the team leader: *"one possible customization for this microservice is to allow using centralized or decentralized files for each project"*. Other opportunities mentioned by developers involve the microservices *Algorithm Flow Builder* and *Project Metadata*. The developers pointed different types of customization: *(i)* customizing the interface of a microservice, *(ii)* changing a microservice by specialization, and *(iii)* completely disabling a microservice.

Regarding the solution of Exp–5MS, four microservices contain variability. *Algorithm Analyzer* and *Project Metadata* may be customized at the interface level. *Project Files Manager* can be specialized according to the product to be generated, and *Project Metadata* can be completely disabled. *Algorithm Analyzer* is affected by variability on its interface since the subfeature responsible for enabling the builder of algorithm flows, *e.g.*, build a scientific workflow, can be disabled in some products. Thus, endpoints responsible to build an algorithm flow can be disabled to these products. *Project Files Manager* is completely responsible for enabling the feature of a project file.

However, products can implement the file management in different ways. For example, a client deriving a product can implement a distributed file system, use a file system of a cloud service, or a simple and centralized file system (default option). Hence, the microservice can be specialized by each product. *Project Metadata* contains information about the users of each project. In this order, different information can be stored and provided in the interface of this microservice. Thus, this microservice presents variability in its REST interface, allowing endpoints be customized or the metadata in the interface response contain variability. In addition, the complete concept of the project (a shared environment to researchers) in a product might be completely disabled.

Three microservices can be customized in the solutions of Exp–7MS and Exp–10MS. The subfeature responsible for building an algorithm flow was modularized in the microservice *Algorithm Flow Builder*. This microservice can be completely disabled in products that do not need this subfeature. The microservices *Project Files Manager* and *Project Metadata* contain the same type of customization of the solution generated by Exp–5MS. In summary, the observed variabilities lead to microservice customization that impacts the microservice interfaces, the microservice specialization or even the microservice existence.

Developers of the legacy maintain several software products for the domain of the oil and gas industry (see Section III). According to these developers, there is potential for reusing the *User, Authentication, Server Log, Project Files Manager* and *Project Credentials* microservices across the existing software products. One participant argued: *"The microservice that realizes management of project files (*Project Files Manager*) is highly reusable by different tenants"*.

RQ4: *We observed three types of customization that can be explored in redesigned microservices: interface customization, microservice specialization, and disabling a microservice. Yet, developers identified opportunities of customization and envisioned reuse opportunities of four microservices from the solutions obtained by our strategy.*

VII. Threats to Validity

An internal threat to the validity of this study can be related to execution traces bias, since all execution traces are generated using given test cases. To mitigate this threat, an expert in the target system conducted all the feature mapping activities. We believe that the test cases cover the features of interest. Another threat is related to the evolutionary algorithm chosen. We used the state-of-the-art many-objective evolutionary algorithm based on NSGA III [29] that has shown high accuracy to solve many-objective problems.

Regarding external validity, the first threat is related to the case study and the discussion of the results. Despite using only one system, it is a real-world legacy system with more than 15 years of existence. We believe we could provide robust and reliable results that can be useful in practice. The second threat is the effect of the participants' background on the results that might lead to different opinions about the solutions. We tried to minimize this threat by involving experienced developers who were knowledgeable with the legacy system and microservice architecture. Another threat is the lack of a meaningful comparison baseline. We mitigated this threat in our previous study [38], where NSGA-III applied in our strategy outperformed a baseline approach.

VIII. Concluding Remarks

In this paper, we introduce a strategy to redesign features as microservices. The strategy uses search-based software engineering techniques to quantitatively assess redesign candidates of legacy features as microservices in terms of four criteria: coupling, cohesion, network overhead, and feature modularization. An industrial case study has shown that the proposed strategy achieves promising results and calls for further investigation of its use on other legacy systems.

The main findings of our case study include discovering that (i) the features were modularized as microservices aligned with the business capabilities of the legacy system; (ii) the developers of the legacy system consider mainly the criteria included in our strategy; and (iii) reuse and customization opportunities arouse from the redesign of features as microservices. In this sense, a future direction is to propose an explicit criterion related to variability to support configurability.

Acknowledgment

This work was funded by CNPq (grants 434969/2018-4, 408356/2018-9, 428994/2018-0, 312149/2016-6), FAPERJ (grants 200773/2019, 010002285/2019, PDR-10 Fellowship 202073/2020), CAPES/Procad (grant 175956), CAPES/Proex and FAPPR (grants 51435, 51152). The authors thank Tecgraf Institute of PUC-Rio for its support in the industrial case study.

REFERENCES

[1] S. Fowler, *Production-Ready Microservices*. O'Reilly Media, 2016.

[2] C. Watson, S. Emmons, and B. Gregg. (2015) A microscope on microservices. [Online]. Available: http://techblog.netflix.com/2015/02/a-microscope-on-microservices.html

[3] W. Luz, E. Agilar, M. C. de Oliveira, C. E. R. de Melo, G. Pinto, and R. Bonifácio, "An experience report on the adoption of microservices in three brazilian government institutions," in *XXXII Brazilian Symposium on Software Engineering*. ACM, 2018, pp. 32–41.

[4] L. P. Tizzei, M. Nery, V. C. V. B. Segura, and R. F. G. Cerqueira, "Using microservices and software product line engineering to support reuse of evolving multi-tenant saas," in *21st International Systems and Software Product Line Conference (SPLC)*. ACM, 2017, pp. 205–214.

[5] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From monolithic to microservices: An experience report from the banking domain," *IEEE Software*, vol. 35, no. 3, pp. 50–55, 2018.

[6] S. Newman, *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015.

[7] J. Lewis and M. Fowler. (2014) Microservices. [Online]. Available: https://martinfowler.com/articles/microservices.html

[8] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-021, 1990. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231

[9] A. Henry and Y. Ridene, *Migrating to Microservices*. Cham: Springer International Publishing, 2020, pp. 45–72.

[10] L. Carvalho, A. Garcia, W. K. G. Assunção, R. de Mello, and M. J. de Lima, "Analysis of the criteria adopted in industry to extract microservices," in *Proceedings of the Joint 7th International Workshop on Conducting Empirical Studies in Industry and 6th International Workshop on Software Engineering Research and Industrial Practice*, ser. CESSER-IP '19. IEEE Press, 2019, pp. 22–29.

[11] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 524–531.

[12] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, and R. Casallas, "Towards the understanding and evolution of monolithic applications as microservices," in *XLII Latin American Computing Conference (CLEI)*. IEEE, 2016, pp. 1–11.

[13] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[14] W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai, "Functionality-oriented microservice extraction based on execution trace clustering," in *International Conference on Web Services (ICWS)*. IEEE, 2018, pp. 211–218.

[15] A. Isazadeh, H. Izadkhah, and I. Elgedawy, *Source code modularization: theory and techniques*. Springer, 2017.

[16] F. Brito e Abreu and M. Goulao, "Coupling and cohesion as modularization drivers: are we being over-persuaded?" in *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, 2001, pp. 47–57.

[17] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization: Is it enough?" *ACM Trans.s on Software Engineering and Methodology*, vol. 25, no. 3, Jun. 2016.

[18] J. Gouigoux and D. Tamzalit, "From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture," in *International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 62–65.

[19] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[20] L. Nunes, N. Santos, and A. Rito Silva, "From a monolith to a microservices architecture: An approach based on transactional contexts," in *Software Architecture*, T. Bures, L. Duchien, and P. Inverardi, Eds. Cham: Springer International Publishing, 2019, pp. 37–52.

[21] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, and Z. Shan, "A dataflow-driven approach to identifying microservices from monolithic applications," *Journal of Systems and Software*, vol. 157, p. 110380, 2019.

[22] I. Pigazzini, F. Arcelli Fontana, and A. Maggioni, "Tool support for the migration to microservice architecture: An industrial case study," in *Software Architecture*, T. Bures, L. Duchien, and P. Inverardi, Eds. Cham: Springer International Publishing, 2019, pp. 247–263.

[23] A. Megargel, V. Shankararaman, and D. K. Walker, *Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example*. Cham: Springer International Publishing, 2020, pp. 85–108. [Online]. Available: https://doi.org/10.1007/978-3-030-33624-0_4

[24] S. A. Maisto, B. Di Martino, and S. Nacchia, "From monolith to cloud architecture using semi-automated microservices modernization," in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, L. Barolli, P. Hellinckx, and J. Natwichai, Eds. Cham: Springer International Publishing, 2020, pp. 638–647.

[25] H. H. da Silva, G. F. d. Carneiro, and M. P. Monteiro, "Towards a roadmap for the migration of legacy software systems to a microservice based architecture," in *9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*. SciTePress, 2019, pp. 37–47.

[26] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: a systematic mapping study," in *8th International Conference on Cloud Computing and Services Science*. SciTePress, 2018.

[27] L. Carvalho, A. Garcia, W. K. G. Assunção, R. Bonifácio, L. P. Tizzei, and T. E. Colanzi, "Extraction of configurable and reusable microservices from legacy systems: An exploratory study," in *23rd International Systems and Software Product Line Conference*, ser. SPLC'19. ACM, 2019, pp. 26–31.

[28] L. Carvalho, A. Garcia, T. E. Colanzi, W. K. G. Assunção, M. J. Lima, B. Fonseca, M. Ribeiro, and C. Lucena, "Search-based many-criteria identification of microservices from legacy systems," in *22th Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '20. ACM, 2020, pp. 305–306.

[29] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2014.

[30] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998.

[31] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *9th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2007, pp. 1106–1113.

[32] D. Goldberg, K. Deb, and J. Clark, "Genetic algorithms, noise, and the sizing of populations," *Complex Systems*, vol. 6, pp. 333–362, 1992.

[33] M. Fowler, *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[34] J. Cochrane and M. Zeleny, *Multiple Criteria Decision Making*. University of South Carolina Press, Columbia, 1973.

[35] G. Vlahavas, "depfinder: a tool that finds dependencies of slackware packages," http://depfinder.sourceforge.net/, accessed: 2020-10-14.

[36] W. K. G. Assunção, T. E. Colanzi, L. Carvalho, J. A. Pereira, A. Garcia, M. J. de Lima, and C. Lucena, "Supplementary material," https://wesleyklewerton.github.io/publications/SANER2021_complementary_material.pdf, accessed: 2021-01-05.

[37] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[38] L. Carvalho, A. Garcia, T. E. Colanzi, W. K. G. Assunção, J. A. Pereira, B. Fonseca, M. Ribeiro, M. J. Lima, and C. Lucena, "On the performance and adoption of search-based microservice identification with tomicroservices," in *36th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 569–580.