# Weighing the Evidence: On Relationship Types in Microservice Extraction

Lisa J. Kirby
Univ. of British Columbia,
Vancouver, Canada
lisakirby@alumni.ubc.ca

Evelien Boerstra
Univ. of British Columbia,
Vancouver, Canada
evelien.boerstra@alumni.ubc.ca

Zachary J.C. Anderson
Univ. of British Columbia,
Vancouver, Canada
zacharyjcanderson@alumni.ubc.ca

Julia Rubin
Univ. of British Columbia,
Vancouver, Canada
mjulia@ece.ubc.ca

***Abstract*—The microservice-based architecture – a SOA-inspired principle of dividing systems into components that communicate with each other using language-agnostic APIs – has gained increased popularity in industry. Yet, migrating a monolithic application to microservices is a challenging task. A number of automated microservice extraction techniques have been proposed to help developers with the migration complexity. These techniques, at large, construct a graph-based representation of an application and cluster its elements into service candidates. The techniques vary by their decomposition goals and, subsequently, types of relationships between application elements that they consider – structural, semantic term similarity, and evolutionary – with each technique utilizing a fixed subset and weighting of these relationship types.**

**In this paper, we perform a multi-method exploratory study with 10 industrial practitioners to investigate (1) the applicability and usefulness of different relationships types during the microservice extraction process and (2) expectations practitioners have for tools utilizing such relationships. Our results show that practitioners often need a "what-if" analysis tool that simultaneously considers multiple relationship types during the extraction process and that there is no fixed way to weight these relationships. Our study also identifies organization- and application-specific considerations that lead practitioners to prefer certain relationship types over others, e.g., the age of the codebase and languages spoken in the organization. It outlines possible strategies to help developers during the extraction process, e.g., the ability to iteratively filter and customize relationships.**

## I. Introduction

The microservice-based architecture is an approach wherein a monolithic application is divided into a distributed system of autonomous, independently deployable services. Adopting this approach has advantages over using a traditional monolithic architecture, including stronger modularity at the boundaries of each microservice, the ability to selectively scale out and/or deploy services within a system, and the option to develop a system with polyglot or multilingual technology stacks. To harness these advantages, many organizations are looking to migrate existing applications to microservice-based architectures; however, extracting code from large monolithic applications into *microservice candidates* is a challenging task [10], [42]. The extraction process can entail manually sorting hundreds, if not thousands of files, classes, and methods into groups – a process requiring substantial time, effort, and familiarity with the existing monolithic system.

A number of automated techniques have recently been proposed to assist developers with microservice extraction [4], [9], [14], [23], [31], [32], [35], [38], [44]. These techniques, at large, attempt to analyze and capture relationships between elements of a monolithic system and further use these relationships to group similar elements together.

The techniques vary by the types of relations they consider. *Structural relationships*, e.g., static or dynamic call dependencies [14], [23], [38], [44], aim to capture architectural similarity between application elements and group together elements that are likely to belong to the same architectural component. The main idea behind these techniques is that architectural components are good candidates to become microservices as they minimize inter-service calls and performance overhead. *Semantic relationships*, e.g., identifier similarity [4], [32], aim to capture lexical similarity between application elements and group together elements that use the same terminology. The idea here is that elements using similar terminology are likely to belong to the same business domain. Finally, *evolutionary relationships*, e.g., contributor similarity [14], [32], aim to capture the structure of the team working on an application and group together elements developed by the same team members. The idea behind these techniques is to produce microservices that can be developed by independent teams. Some of the techniques also utilize a combination of relationships, e.g., dynamic calls and class name similarities [23], assigning each relationship type a fixed weight during microservice extraction process.

The techniques are typically evaluated on benchmarks and metrics selected by the authors, demonstrating the usefulness of the technique for a particular decomposition criteria, e.g., independence of development. There is also a number of reports evaluating a technique on an industrial case study [9], [31], [35]. Yet, each such report contributes a new microservice extraction tool variant, with its own considered relationship types and weights. In practice, microservice extraction techniques do not see wide adoption yet [10], [11], [17].

Motivated to understand how automated approaches can better assist practitioners with the microservice extraction process, in this work, we investigate the following research questions:

**RQ1 (Usefulness):** What is the applicability and usefulness of different relationship types during the extraction process?

**RQ2 (Tool support):** What features would benefit automated extraction tools that utilize element-to-element relationships?

To answer these research questions, we conducted a multi-method empirical investigation, collecting opinions of 10 industrial practitioners experienced with microservice extraction. Specifically, to answer **RQ1**, we performed semi-structured interviews during which we evaluated the importance practitioners assign to different relationship types during the decomposition process. To answer **RQ2**, we conducted think-aloud sessions for which we implemented a generic, "vanilla" microservice extraction prototype capable of dealing with multiple types of relationships, applied it on a realistic case-study application, and used that setup to collect expectations practitioners have for tools utilizing such relationships. We choose this method as having a working prototype is shown to be an effective method for identifying requirements compared with simply asking the participants about their needs [7], [26].

Our results show that practitioners value a "what-if" analyses tool allowing them to investigate different types of relationships and decompositions based on these relationships. This is because they consider multiple relationship types during the extraction process and because there is no uniform way to pick and weight these relationships across all applications, organizations, and teams. Structural relationships can easily become polluted when there are many structural dependencies on elements that cross-cut concerns. Semantic name similarity might not work well in aged code bases, where names used at the beginning of development no longer have a clear meaning, especially as people contributing to the project change. Likewise, evolutionary data can be highly polluted in long-living projects: data accumulated over decades of development is likely unreliable due to transition of development between different version control systems and commit models, changes in design, and fluctuation of personnel.

Furthermore, our results provide several actionable suggestions for future microservice candidate extraction tools. For example, our participants indicated the need to iteratively filter and customize relationships during the extraction process, apply different levels of abstraction to different parts of the application, interactively work with the tool to "fix" some decisions and let the tool propose candidates that take these fixed decisions into account, evaluate multiple extraction strategies against each other, and more.

We believe the results of our study will inform and inspire researchers and tool builders to devise novel solutions that better address practitioners' needs. Our results can also benefit industrial practitioners who perform microservice extraction and are interested in learning from each other, borrowing successful ideas, and avoiding common mistakes. We make our prototype tool and the experimental data we used for the study publicly available, to facilitate future work in this area [25].

The remainder of the paper is structured as follows. Section II describes the microservice architectural principle and the types of relationships considered by the state-of-the-art microservice extraction approaches. Section III presents the design of our study, and Section IV discusses the results. We discuss the threats to the validity of our findings in Section V. Section VI outlines the related work, and Section VII concludes the paper.

## II. From Monolithic to Microservice-based Applications

### A. *Microservices*

The microservice-based architecture is a software development paradigm in which an application is arranged as a collection of loosely coupled services that communicate with each other using lightweight technologies such as HTTP/REST [27]. Microservice-based architectures are closely related to service-oriented architectures (SOA), which is a style of software design where services represent application components that communicate over a network [33].

Microservices aim at shortening the development lifecycle while improving the quality, availability, and scalability of applications at runtime. Cutting one big application into small independent pieces reinforces the component abstraction and makes it easier for the system to maintain clear boundaries between components: APIs specified in the service contract are the only channel for accessing the service. Developers can focus on small parts of an application, without the need to reason about complex dependencies and large code bases [46]. Microservice-based applications also promote autonomous teams working on services that are organized around business capabilities and assume end-to-end responsibility for these capabilities, from development to production. Another major advantage of microservice-based architectures is independent deployment, which reduces the coordination effort needed to align on common application delivery cycles and also leads to independent scaling at runtime.

Migrating legacy applications from monolithic to microservice architectures is a challenging task. We now review the landscape of the microservice extraction techniques designed to help the developers with this endeavor.

### B. *Microservice Extraction Techniques*

To collect the set of main principles underlying existing microservice extraction techniques, we started from the recent systematic literature review on the topic conducted by Ponce et al. [36]. In January 2021, we performed a secondary search using the same methodology, to cover more recent papers and also performed a snowballing review on the identified papers. We focused on automated techniques for producing microservice candidates, omitting approaches that rely on additional artifacts often unavailable in practice and/or taking a considerable time to produce, e.g., entity-relationship diagrams and use case models [21], [28]. This search resulted in more than 10 automated techniques listed in Table I. We also list the number of citations for each corresponding paper according to Google Scholar as of January 2021.

At large, these techniques construct a graph representation of the monolithic application, wherein the nodes represent application components, e.g., packages, classes, methods, etc., and the edges between the nodes represent relationships between the components, as discussed next. Once the graph-based representation of the program is extracted, the main properties driving the decomposition are loose coupling (minimizing inter-service connection) and high cohesion (maximizing

TABLE I: Considered Microservice Extraction Techniques

| Paper | Year | #Cit. | Approach |
|---|---|---|---|
| Escobar et al., "Towards the Understanding and Evolution of Monolithic Applications as Microservices" [13] | 2016 | 40 | *Structural-static* (method calls, inheritance) |
| Baresi et al., "Microservices Identification Through Interface Analysis" [4] | 2017 | 46 | *Semantic* (similarity of terms) |
| Mazlami et al., "Extraction of Microservices from Monolithic Software Architectures" [32] | 2017 | 92 | *Semantic* (similarity of terms) OR *Evolutionary* (artifact co-changes) OR *Evolutionary* (contributors) |
| Eski and Buzluca, "An Automatic Extraction Approach: Transition to Microservices Architecture from Monolithic Application" [14] | 2018 | 8 | *Structural-static* (method calls, inheritance) AND *Evolutionary* (artifact co-changes) |
| Selmadji et al., "Re-architecting OO Software into Microservices A Quality-Centred Approach" [40] | 2018 | 5 | *Structural-static* (method calls, data dependencies) |
| Ren et al., "Migrating Web Applications from Monolithic Structure to Microservices Architecture" [38] | 2018 | 11 | *Structural-static* (method calls) AND *Structural-dynamic* (weighted method calls) |
| Abdullah et al., "Unsupervised Learning Approach for Web Application Auto-decomposition into Microservices" [3] | 2019 | 21 | *Structural-dynamic* (access logs, URI groups with similar resource requirements) |
| Taibi et al., "From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining" [44] | 2019 | 16 | *Structural-dynamic* (weighted method calls) |
| Jin et al., "Service Candidate Identification from Monolithic Systems based on Execution Traces" [23] | 2019 | 12 | *Structural-dynamic* (weighted method calls) AND *Semantic* (class name similarity) |
| Pigazzini et al., "Tool Support for the Migration to Microservice Architecture: An Industrial Case Study" [35] | 2019 | 2 | *Structural-static* (data dependencies, inheritance, package structure) AND *Semantic (term similarity)* |
| Carvalho et al., "On the Performance and Adoption of Search-Based Microservice Identification with toMicroservices" [9] | 2020 | 1 | *Structural-static* (method calls, data dependencies) AND *Structural-dynamic* (weighted method calls, data dependencies) |
| Matias et al., "Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis" [31] | 2020 | 0 | *Structural-static* (method calls, package structure) AND *Structural-dynamic* (weighted method call) |
| Kalia et al., "Mono2Micro: An AI-based Toolchain for Evolving Monolithic Enterprise Applications to a Microservice Architecture" [24] | 2020 | 0 | *Structural-static* (weighted method calls, data dependencies, inheritance) OR *Structural-dynamic* (weighted method calls) |

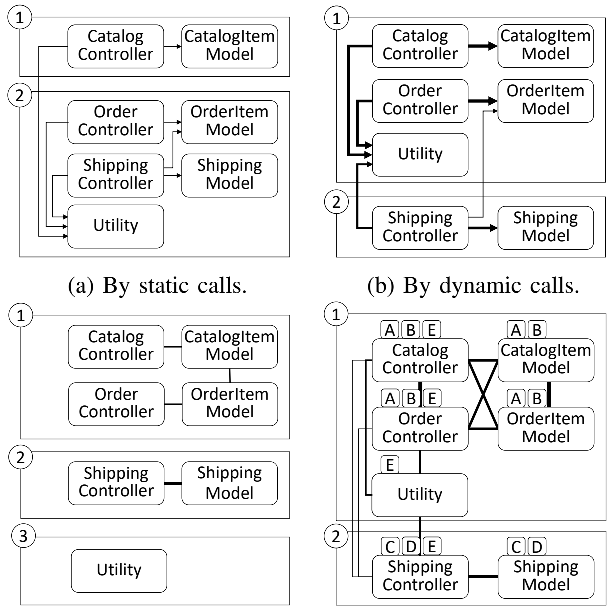
intra-service connections). For that purpose, the techniques utilize existing clustering algorithms, such as k-Means and Agglomerative Hierarchical Clustering [16].

The relationship types extracted by the techniques (see the last column of Table I) determine the nature of the decomposition and can broadly be divided into *structural*, *semantic*, and *evolutionary*. We discuss these relationships and exemplify them on a simple monolithic application in Figure 1. We also refer to this example later in the paper, as it is a "mini-version" of the case study app we used in our study.

The example application consists of seven classes, six of which correspond to three conceptual domains: catalog, which allows the user to browse items, order, for ordering from the catalog, and shipping, which manages the shipment of orders. Within each domain, a controller class communicates with its corresponding persistence layer class, e.g., *Shipping-Controller* and *ShippingModel*. The *ShippingController* class also communicates with the *OrderItemModel* class: when a shipment request is submitted, the controller retrieves and updates information about the *Orders* being shipped.

**Structural relationships** are most frequently used by the existing techniques. These relationships are obtained via static or dynamic code analysis. This type of relationship aims to group together architecturally-related elements, reducing inter-service communication and minimizing performance overhead. *Static structural relationships* include method calls, i.e., static call graphs. Some techniques also use data dependencies, inheritance, and package structure, e.g., [13], [14], [24]. *Dynamic structural relationships* include method calls, which are typically weighted based on the frequency of invocations in the dynamic execution traces [23], [24], [38], data dependencies between elements [9], and dependencies on shared resources [3].

For the example in Figure 1a, solid lines represent statically-obtained, non-weighted method call relationships between classes. Microservice candidates obtained by clustering based



(a) By static calls. (b) By dynamic calls.

(c) By class names (semantic). (d) By commits (evolutionary).

Fig. 1: Extraction Candidates for Different Relationship Types

on these relationships are denoted by numbered boxes enclosing the classes. Here, classes from the *Catalog* domain are combined into one service candidate as the *CatalogCon-troller* and *CatalogItemModel* classes call each other. Classes from the *Order* and *Shipping* domains are combined into a separate service candidate due to the structural dependency between classes within each of these domains and between the *ShippingController* and *OrderItemModel* classes. *Utility* was added to this service candidate as well, as its affinity with *OrderController* and *ShippingController* is stronger than with *CatalogController* only alone.

Figure 1b shows weighted dynamic method call relationships, as well as microservice candidates produced using these relationships. In this figure, the weight of a relationship is indicated by the thickness of the line. At runtime, all controller classes heavily rely on their respective model classes and call the generic *Utility* class. Overall, classes from the *Shipping* domain are called less frequently than *Catalog* and *Order* classes. This creates coupling between *Catalog* and *Order* domains, pulling them into service #1.

**Semantic relationships** are obtained via Information Retrieval (IR) techniques. Intuitively, this type of relationship aims to identify elements that belong to the same business domain by relying on naming conventions followed by developers. Some of the techniques focus on *class name similarity* only [23], i.e., the edge between the classes is weighted based on the similarity of terms in their class names. Other techniques compute broader *term similarity*, considering other terms within the class/method body, such as variable names, method names and parameters, comments, etc. [4], [32]. Most of the techniques exclude stop words, i.e., programming-language-specific and technical terms.

Figure 1c shows class name similarity relationships, computed by dividing the number of common words by the total number of words in both class names, while excluding *Controller* and *Model* terms, as done in prior work [23], [32]. In this example, classes from the *Order* and *Catalog* domains are combined into one service candidate due to the semantic coupling caused by the shared *Item* term, while classes from the *Shipping* domain, as well as the *Utility* class, are placed in their own service candidates.

**Evolutionary relationships** are obtained by mining code repositories. These relationships aim to produce microservice candidates that can be maintained and developed by autonomous teams, grouping together code contributed by the same developers or code frequently changed together. More specifically, *commit similarity* between two elements is calculated as the number of commits shared by these elements, normalized by the combined number of unique commits that involve either element [14], [32]. *Contributor similarity* is defined as the number of developers who have worked on both elements, normalized by the total number of unique developers that have worked on either element [32].

For the example in Figure 1d, boxed letters placed at the top of a class represent distinct commits that have changed that class, and the edges between classes represent weighted commit similarity relationships computed using this information. For example, *CatalogController*, *CatalogItemModel*, *OrderController*, and *OrderItemModel* were all changed together in both commit A and B. These classes are placed in service #1, together with the *Utility* class that has a stronger affinity with the *Catalog* and *Order* controllers than with the single *Shipping* controller.

While in this example each decomposition is performed by a single relationship type (with just the method calls for structural relationships), several approaches consider multiple relationships simultaneously [9], [14], [23], [31], [35], [38]. Yet, none of these approaches allows the user to control weights, i.e., the importance assigned to each type of relationship.

## III. Research Methodology and Study Design

To answer our research questions introduced in Section I, we collected input from 10 industrial practitioners experienced with microservice extraction. We divided the data collection process into two parts, which were performed during the same online session. To answer **RQ1**, we followed the semi-structured interviews methodology, investigating the practitioners' perspectives on the applicability and usefulness of different relationships types during the extraction process. For **RQ2**, we conducted think-aloud sessions with practitioners for which we implemented a generic microservice extraction prototype capable of dealing with multiple types of relationships. We used this prototype to collect expectations practitioners have for tools utilizing such relationships. We now describe our selection of participants, the design of both parts of the study, and our process of analyzing the results.

### *A. Subjects*

We recruited software developers experienced with microservices and microservice extraction approaches. To ensure that our data is valuable and reliable, our selection criteria were for participants to (1) have more than five years of software development experience and (2) more than one year of involvement with a microservice-based migration of substantial size and complexity.

For identifying the interviewees, we reached out to developers who actively participate in microservice-related events and meetup groups, and used the Reddit and Twitter web platforms to recruit developers that include microservice development in their core skills and hold active software development positions. Our selection criteria were purposely strict to include only participants with substantial experience decomposing monolithic systems into microservices. Yet, we were able to include in our study 10 who have such an experience, some of which also authored popular books on microservices (P4, P10) or worked with microservices even before the term was officially coined in 2011 [12] (P8).

Table II contains detailed information about our study practitioners, including their current position, years of general software development experience, and years of experience working specifically with microservice-based applications. These practitioners are from eight companies in Canada, Croatia, Germany, Norway, Australia, and the USA, with the sizes of the companies ranging from small to large, according to the corresponding EU classification [15]. For two of the companies, we interviewed two employees each, as these employees work in different areas/countries.

All our practitioners were involved in decomposing at least one monolithic application into microservices. Some, e.g., P1, have spent a number of years on decomposing a single monolithic system; others, e.g., P2, P4, P10, work primarily as architecture consultants, helping different companies decompose their monoliths into microservices. We believe such a range of experiences is highly valuable for our study.

Table II also lists the size and age of the largest monolithic application practitioner decomposed. We categorize monoliths with up to 50 thousands of lines of code (KLOC) as small, between 50 to 250 KLOC as medium, and over 250 KLOC as large. P1, P2, P4, and P7 all report experience with

TABLE II: Interview Participants.

| ID | Position | Development Experience [years] | Microservice Experience [years] | Largest Decomposed Monolith [KLOC] | Oldest Decomposed Monolith [years] | Company Size | Education | Country |
|---|---|---|---|---|---|---|---|---|
| P1 | Staff Software Developer | 5-10 | 5-10 | Large | 10-20 | Large | Bachelor | Canada |
| P2 | Architecture Consultant | 10-20 | 1-5 | Large | 20+ | Medium | Master's | Croatia |
| P3 | Director of Engineering | 20+ | 5-10 | Medium | 10-20 | Medium | Master's | Croatia |
| P4 | Fellow, Architecture Consultant | 20+ | 5-10 | Large | 5-10 | Medium | Master's | Germany |
| P5 | VP, Software Development | 20+ | 5-10 | N/A | N/A | Large | Bachelor | Canada |
| P6 | Architect | 10-20 | 5-10 | Small | 5-10 | Large | Bachelor | Canada |
| P7 | Executive Architect | 20+ | 1-5 | Large | 10-20 | Large | PhD | USA |
| P8 | Chief Technology Officer | 20+ | 10+ | Medium | 5-10 | Small | Bachelor | Canada |
| P9 | Architecture Consultant | 10-20 | 5-10 | Medium | 1-5 | Small | Bachelor | Norway |
| P10 | Chief Technology Officer | 20+ | 5-10 | Medium | 10-20 | Small | Bachelor | Australia |

decomposing a monolith with over one million lines of code. P5 could not disclose the size and age of the largest/oldest monoliths they have decomposed.

## B. RQ1: Interview Study

We now describe the interview study we conducted to investigate the applicability and usefulness of different relationship types during the extraction process.

We performed semi-structured interviews with a set of open-ended questions. To identify an appropriate study protocol, we first conducted two pilot interviews with colleagues that are familiar with microservice-based development. We proceeded to the main study only as the pilot interviews ran smoothly; we discarded the data collected during the pilot study.

The interviews were conducted in English by at least two of the authors of this paper using telecommunication software, such as Zoom. All but one interviewee agreed to be recorded, to avoid misunderstandings and ease the analysis of the collected data. For the remaining interview, both researchers took notes during the meeting. The interviews took between 60 and 90 minutes. Furthermore, we collected quantitative data about the participants' background, experience, their project, etc. offline, which saved time during the interviews.

We began each interview by explaining the goal of our study: to identify how automated tools can better assist practitioners with microservice extraction. We then explained that automated tools rely on different types of relationships between entities, introduced the relationships, and discussed the intuition behind using these relationship, similar to the discussion in Section II-B. We structured the remaining part of the interview around four central questions:

**Q1.** Which relationship types do you find useful for microservice extraction and why?
**Q2.** If there is more than one useful relationship type, what is the relative importance of each relationship and why?
**Q3.** Are there any additional relationship types you find useful?
**Q4.** What characteristics of an application/organization could change how you use relationships?

We followed up with subsequent questions and in-depth discussions based on the interviewees' responses and encouraged participants to include examples from their microservice experience in their answers. A more detailed description of our interview protocol is available online [25].

## C. RQ2: Think-Aloud Study

Following the interview study, we showed participants a "vanilla" microservice extraction tool prototype that allows the user to browse different types of relationships and select the desired types and weightings of these relationships. We implemented this prototype to evaluate practitioners' opinions "in vivo", asking them to verbalize their thoughts as they performed a microservice decomposition task. Such a protocol, referred to as a think-aloud session [22], allows designers to gain insight into the cognitive processes involved with solving a problem and has been shown to be effective for eliciting user needs and requirements [7], [26].

We used a case study monolithic application to facilitate the discussion. We describe this application next and then briefly outline our prototype implementation (a working version is available online [25]) and our study protocol.

*1) PartsUnlimitedMRP Case Study:* We used PartsUnlimitedMRP [1], a Manufacturing Resource Planning (MRP) application developed and maintained by Microsoft, as our case study application. This application is similar to and has inspired our motivating example in Figure 1. We selected this application because it is a realistic, yet relatively compact example of a system that has both a monolithic and microservice-based version [2] available online. Having a microservice-based version of our case study allowed us to better understand ways in which the monolith can be decomposed.

We focus our analysis on the backend part of PartsUnlimitedMRP, which contains 54 classes conceptually corresponding to different functional domains of the system: catalog, dealer, order, quote, and shipment. The application is implemented in Spring and uses Spring controllers, models, and repository components to implement the presentation, business, and persistence layers, respectively. In addition to the Spring-based application elements, the system contains several classes that implement secondary functionalities (i.e., application and database configuration, error handling, utility/logging access code, etc.). These classes are used throughout all domains, like the *Utility* class in Figure 1.

The microservice-based version of PartsUnlimitedMRP contains five services that represent the application's five functional domains. The decomposition itself is performed at the class level, with each of the services containing its corresponding controller, model, and repository classes from the original monolithic application. As the produced microservice-

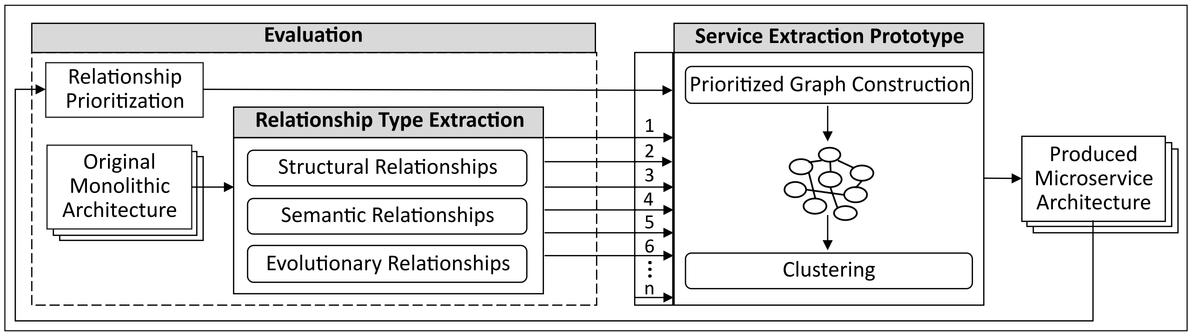

Evaluation
Relationship Prioritization
Original Monolithic Architecture
Relationship Type Extraction
Structural Relationships
Semantic Relationships
Evolutionary Relationships
1
2
3
4
5
6
n
Service Extraction Prototype
Prioritized Graph Construction
Clustering
Produced Microservice Architecture

Fig. 2: Overview of the microservice extraction prototype.

based version is a reasonably straightforward decomposition of the original monolithic application, we believe this is a feasible task, which makes PartsUnlimitedMRP a suitable case study for our analysis. Note that we did not introduce the microservices version of the application to the study participants as the objective of our study is to collect requirements for a microservice extraction tool, rather than to arrive at any particular microservice-based version of PartsUnlimitedMRP.

*2) Prototype Implementation:* The Graphical User Interface (GUI) of our prototype tool [25] contains six tabs, each corresponding to a relationship type described in Section II: static, dynamic, class name, term, commit, and contributor similarity. This presentation is similar to the one in Figure 1, where the nodes represent classes of an application and the edges represent a particular type of relationship. The user can move the nodes around and reorganize the view in the way most suitable for them. When the user switches between tabs, the nodes stay in their selected location and only the relationships between the nodes change. The GUI also includes an *edge weight slider* that can hide weaker relationships within the graph, which we introduced following a pilot study. A *clustering menu* allows the user to select the types and weights of relationships and, based on that, create a set of service candidates extracted according to the user's selected relationships/weights. The user can then inspect the results, adjust the relationship weights if needed, and repeat the extraction process.

The underlying implementation of the tool is shown in Figure 2 and is also available online [25]. It obtains as input any number and type of relationships extracted from a monolith, alongside the user's weighting of each relationship. For our prototype implementation, we relied on SciTool's Understand [39] to extract static structural relationships between classes, which included class-to-class calls, data member usages, object references, and inheritance dependencies. For dynamic structural relationships, we instrumented the monolithic version of PartsUnlimitedMRP with the Kieker execution monitoring framework [45], ran the functional test suites of the application, and further interacted with the user interface of each application to increase trace coverage.

For class name similarity, we collected the names of Java classes within the application and, again, relied on SciTool's Understand to tokenize each class of the application into a set of terms. We then used TF-IDF [37] to determine a term vector for each class and used the cosine similarity between term vectors to represent the weight of the relationship between classes. We extracted data for both commit similarity and contributor similarity by mining Git version control logs. All referenced data, including the scripts used to collect and process the data, is publicly available to ensure reproducibility [25].

Given the set of relationships and their weights provided by the user, the tool constructs a single graph that represents the prioritized relationship types. To this end, we normalized structural relationships to be in the range [0,1] (semantic and evolutionary relationships are already normalized) and then applied the user's prioritization to the relationship types by multiplying the weights of each relationship type by its corresponding prioritization. Finally, we used *TurboMQ* clustering, a hill-climbing algorithm proposed by BUNCH [34] to cluster the elements of the graph into microservice candidates. We chose this implementation as it is very efficient in terms of its runtime, which is critical for collecting user feedback on extraction results during an interactive session and also because BUNCH performed well in earlier comparative studies [19]. As a search-based approach, Bunch may yield different results in different runs. To mitigate non-determinism for our experiments, we configured BUNCH to run with a very large population of initial clusters, which can be extensively explored due to the size of our case study.

*3) Methodology:* We began each session by presenting the purpose and high-level design of PartsUnlimitedMRP to the participants and asked them to imagine they need to decompose this application into microservices. We then showed participants the graphical user interface (GUI) of our prototype tool. In the GUI, the classes of PartsUnlimitedMRP were grouped horizontally by technical layer, e.g., controllers, repositories, and models, and vertically by domains, e.g., orders, catalog, shipping. To help the practitioners better understand PartsUnlimitedMRP within the time constraints of the interview, we used Zoom's Annotate features to highlight and explain the purpose of each technical layer and domain within the app. We then identified and explained the role of four utility-based classes used to hold common string operations, handle request errors, switch between databases, and perform basic database read and write operations.

We asked participants to complete three tasks, encouraging them to verbalize their thoughts and to explain their actions as they worked through each task.

**T1.** Use the GUI to familiarize yourself with the relationship types of PartsUnlimitedMRP.
**T2.** Distribute 100 points across the relationship types according to their importance in a microservice extraction of PartsUnlimitedMRP.
**T3.** Use the clustering menu to produce a microservice extraction of PartsUnlimitedMRP.
We answered any questions participants asked about the implementation of PartsUnlimitedMRP. We also encouraged them to repeat T2 and T3, if they indicated that they were unhappy with the microservice candidates generated by their weightings. All verbalizations and interactions with the GUI were recorded for review.

### D. Data Analysis

We used open coding – a qualitative data analysis technique borrowed from grounded theory – to identify main ideas expressed by our study participants [41]. To this end, we used an automated tool to transcribe the interview recordings, and we cross-checked the generated transcriptions against the interview recordings to identify and correct any transcription mistakes. Two of the paper authors then independently read the transcriptions and interview notes to identify and name concepts – key ideas contained in data – for relationship-usefulness and tool-expectation categories, separately. When looking for concepts, the authors searched for the best phrase that describes what we believe is the idea expressed by the participant. Then, all the authors met to discuss the identified concepts, refine concept labeling, and merge related concepts, if needed. The results of our study are described next.

## IV. Results

### A. RQ1: Applicability and Usefulness of Relationship Types

We heard a range of replies about relationships practitioners find useful for automated microservice extraction. Interestingly, only two of the participants expressed a strong preference for one particular relationship type: P3 for evolutionary, to produce microservices that can be developed and maintained by independent teams, and P7 for structural, to prioritize architecturally-cohesive services that minimize refactoring effort. Other participants found a combination of relationships useful for incorporating benefits of multiple strategies.

At the same time, all participants repeatedly commented that there is no one-size-fits-all strategy that can work across all applications. They emphasized the importance of a "what-if" analysis tool that can help examine and experiment with different relationships and weights to take application-, organization-, and domain-specific considerations into account. We now outline such considerations provided by the participants.

*1) Structural Relationships:* The programming language of a monolithic application affects its structural properties, which influences the decomposition strategy. For example, the value of static and dynamic data can vary depending on the type system of a programming language: «(P5) In a statically typed language, looking at the code would be interesting. If it is a dynamically typed language, then you would want to put the full [weighting] on dynamic [analysis]». The nature of function calls between elements also influences the value of structural relationships. For example, Ruby relies on implicit method calls, which complicates the identification and comprehension of structural relationships: «(P1) When [we] were looking at the dynamic analysis and then looking at the code, we could not even see how the two were related». Likewise, dependency injections – a technique in which an object receives other objects that it depends on – lead to strong structural relationships between these classes, which is not necessarily valuable for the decomposition purposes: «(P1) Those classes should not be tightly coupled, but [that] was not reflected in the [structural] analysis at all.»

Several participants, e.g., P1-P5, P8-P10, deprioritized structural relationships because they believed that these relationships become entangled and lose their meaning as a monolith ages and expands: «(P4) The structure of the source code that exists often leaves something to be desired. Most often, the architecture became messed up over time. [...] So if I take the existing structure and use an algorithm that aims to preserve it, I'd be pretty surprised if that brings any benefit.» In addition, some participants, e.g., P4, P9, P10, noted that structural relationships do not provide as much business and organization value as other relationship types: «(P10) doing what's easy to refactor, isn't necessarily going to deliver value to the business. Usually, the business doesn't care about the structure of the code as much.»

Participants mentioned the difficulty of acquiring reliable dynamic data: «(P2) Because you would have to have a really comprehensive test suite that encompasses many different functionalities. [...] Large monoliths [I have worked with] often have incomplete functional test suites.» When the dynamic data is incomplete, participants recommended complementing it with other types of relationships: «(P7) In those types of cases, I may use some semantic relationships to see if I can derive relationships between classes that can be verified later.»

At the same time, two of the participants, P6 and P7, considered structural relationships to be the most reliable way of performing decompositions, viewing microservice extraction primarily as a refactoring effort: «(P7) When I go for actual monolith code refactoring/rearchitecting, [using semantic relationships] might not produce good clusters because there could be relationships between classes that break the code and structural dependencies between classes. I would instead focus on structural relationships and the partitions they produce that can be realized with minimal code tweaking.»

Several participants, e.g., P7-P10, wanted to see database structure and access patterns as part of the considered structural relationships: «(P9) You look at the database first and what is accessing it, then you work backwards [into the code], using transactional integrity as a guide.» They noted that database decomposition itself could be a difficult task, as it has to be done carefully to minimize transactions that span multiple services/databases. Yet, considering the database structure and how it affects the structure of the code could be of value.

*2) Semantic Relationships:* Several participants, e.g., P4, P5, and P9, prioritized class name similarities because «(P5) [...] a lot of people put significant effort into naming their classes.» However, while in newer applications a significant amount of time and effort is put into class naming, as a code base ages, class names tend to deviate from the original intention: «(P1) We have a lot of cases where, because our application is 14 years old now, the [class] names just do not apply anymore [...]. The older the classes and the more changes the code base has had, the less I would trust class names. [...] It takes a lot of effort to refactor class names and developers usually care more about code they are adding and changing than they care about code that is already there.»

The language spoken within an organization or team also influences the naming of classes and terms within classes. In these cases, semantic data can be unreliable for microservice extraction: «(P2) Because English is not our first language, [...] language ambiguity plays a role. For example, in Croatian, there is a word called predmet that translates in English to object, subject, document, etc. So when we translate this to English, we can end up with ObjectController, Document-Controller, or SubjectController.» Such situations increase inconsistencies, especially in globally-distributed organizations, where the development is carried out in multiple countries. The usefulness of semantic data is also affected by the team size and development practices they employ: «(P7) The more developers you have, the more things will have potential differences in how things are named and how things are written.»

Participants repeatedly noted that using semantic relationships can introduce false coupling between domains when terms are re-used across bounded contexts: «(P1) Something that exists in domain-driven design is the idea of bounded contexts, so that a name would have a meaning in one context, but not in the overall context of the app. [...] In one context we might have *Customer*, and in another context we might have *Customer*, but those classes might be completely different customers that actually have no relationship.»

*3) Evolutionary Relationships:* To our surprise, even though the majority of automated microservice extraction techniques rely on structural and semantic relationships, several participants, e.g., P1-P3, P4, P5, P10, gave strong preference to evolutionary data: «(P10) Evolutionary relationships are the most useful»; «(P3) The primary criteria we look at when grouping classes together into microservices is how often they change together, at once, and who is changing them.» Also, whilst structural and semantic relationships tend to become less reliable over time, commits/co-change relationships become stronger over time and «(P2) capture business requirements from the beginning [of development] to the very end»;

However, like with semantic data, older code means that evolutionary data might be polluted, especially if development transitioned from one code management system to another: «(P2) I do not always have a good version control history, in one 20-year-old code base, most of the code is pushed in an "initial commit" and in that case, I would put more weighting on structural similarity.»

The size of the team can also impact whether contributor similarity is useful for microservice extraction: «(P2) I could see contributor coupling as being useful when you have a very large team and different members of the team are working on different parts of the application. As we try to split monoliths into microservices, we would also split our large team into smaller teams.» For an organization with a small number of developers, contributor similarity becomes less meaningful: «(P4) Separating by contributor similarity might not be worthwhile because developers often work on different parts of the code simultaneously.»

Our study participants, e.g., P2 and P9, also identified additional evolutionary data that could benefit the automated extraction techniques. Specifically, they considered project management and issue tracking information, such as Jira tickets and GitHub issues, valuable to both strengthen the commit data and to estimate the effort required to maintain certain portion of the code, which could inform decomposition effort: «(P9) If you look at the time that elapsed between creating a ticket and the ticket being closed, as well as the time it took to go from development to quality assurance and from quality assurance to production, and then relate this to the amount of code that was changed, then you can see the amount of friction it took to get that feature out into production.»

*4) Summary:* To answer RQ1, our study showed that while each type of relationships has its value and intended use, there is a number of scenarios in which each particular relationship type needs to be re-considered and customized. Programming languages used, age of the code base, size and structure of the team, and even the language spoken by the team members can affect the quality of microservice candidate proposed by an automated extraction technique. Next, we discussed some tool extensions proposed by our study participants.

### *B. RQ2: Tool Support*

*1) Relationship Visualization:* All our participants believed that microservice extraction starts from understanding the monolithic systems and, thus, visualizing different relationship types in itself provides a high value to developers: «(P2) This tool allows you to visualize what's going on in the application»; «(P4) It is a good way to reduce the information and make it digestible.» They expressed difficulties trusting fully-automated tools and believed that a tool that "explains", via visualization, the reasons for the proposed decomposition would be beneficial: «(P10) A tool that actually did the decomposition probably would not be something I would trust for maybe a long time, but if there was a tool that I could give an existing application and it could give me proposed decompositions, [...] I think that would be really useful. [...] Anything that can give me extra information about the situation here is going to be valuable. »

At the same time, some participants noted that with bigger applications, visualizing all elements and relationships might become cumbersome. They wished to customize the level of abstraction at which elements and relationships are presented. Our prototype's *edge weight slider* that can hide weaker relationships between classes was a step in this direction; it was appreciated by practitioners as it allowed them to easily identify the strongest relationships between classes: «(P1) We tried similar visualizations for our app [...] but we did not have an edge weight filter. That would have been very useful.»

One of the participants suggested that the visualization of service candidates could be further improved by highlighting inter-candidate relationships: «(P1) It would be really interesting to then only display the relationships that go between clusters after extracting microservices, [..] to identify elements that cross-cut domains and question why they exist.»

*2) Decomposition Granularity:* Besides the visualization aspect, three participants, P2, P8, and P9, stated that they would rather increase the abstraction level at which microservice extraction is performed. Unlike most existing techniques that perform decomposition at class or even method level, these participants stated that they often perform microservice extraction at the package and/or namespace level: «(P8) If you take everything to a higher level of abstraction, for example with packages instead of classes, then I think the division between potential services becomes even clearer [...] I don't usually look lower than the package level during decompositions.»

This observation implies that developers should be able to configure the abstraction level of entities within relationship graphs. That is, tools can be extended to allow the developer to define the notion of "components" they want to manipulate. Such an extension would also increase the applicability of extraction techniques, at least those based on semantic and evolutionary relationships, across different programming languages: «(P9) A class is a [Java-specific] way of structuring a system – Golang, Python, and Javascript have different ways of modularizing things [...] In a nutshell, units of code that can be versioned together should be kept together as a component.»

*3) Interactive Processes:* An interesting insight was that several participants wanted to "work with the tool" rather than having the tool produce candidates automatically: «(P1) It would be interesting if I could influence the clusterings so that they make sense to me.» For example, they wanted to fix certain decisions by drawing a circle around a group of elements, indicating these elements should stay together. The tool would then proceed with decomposition while respecting this choice: «(P10) [It would be valuable] if you could just draw where the microservices are going to be, but then have the model somehow simulate that and, using all the connections it has got in the graph, tell you how feasible it is for everything to be connected in that way.»

*4) Decomposition Customization:* Participants wanted to customize relationship data based on their specific needs. For structural relationships, they wished to weight different criteria individually, e.g., data dependency or inheritance. For semantic relationship, practitioners mentioned that beyond custom dictionaries, the relationships are missing domain-specific concepts, e.g., bounded contexts: «(P9) The definition [of semantic relationships] is missing the concept of bounded contexts – what "rock" means depends on whether you're in the mountains or in a concert. Context means everything.» Developers also wanted to manually weight different parts of class/variable names: «(P1) [For us] the first part of the name would be some kind of namespacing and defining the context, and then the second part would have to be weighted differently for it to work. In this case, the parts should have different weightings depending on their position in the name.» For evolutionary relationships, commits need to be filtered by the type of the change they make: «(P1) It would be ideal if we could filter commits by feature, bug, and/or project.»

Similarly, as suggested by our findings in RQ1, the applicability and usefulness of different relationship types is highly dependent on the application scenario. This implies that tools need the ability to allow the developer to select and prioritize different types of relationships according to their preferences: «(P1) The weightings are useful.»

*5) Examples and Recommendations:* Some of the participants, i.e., P9 and P10, wanted tools to offer multiple decomposition alternatives, allowing the user to browse and select the desired one. P10 envisioned an iterative genetic algorithm-inspired process, where the developer selects the most fitting alternative, further customizes and fine-tunes it, and lets the tool propose improved variants based on that feedback: «(P10) I would like to have [the tool] consider a million [relationship type] configurations, run them through a simulation to discard some of them, and then present me with a small number of options [displayed] in a grid. Then I choose the options that map best to what I hope would be good configurations, [...] and then fine-tune the weights.»

Participants also wished tools would analyze the structure of the application and provide recommendations about the feasibility and necessity of migration: «(P6) It would be good to look for indicators of when you should not [transition to microservices].» «(P3) If changes are made across all classes by very few people, then it makes the most sense to keep the application as one service that one team handles.»

Moreover, tools could assist with exploring whether there is a configuration of relationships that results in a highly-modular decomposition, or check the modularity of relationship types prioritized by the developer and suggest changes that lead to better modularity. Finally, tools could generate metrics capturing the quality of the decomposition suggested by the developer: «(P10) You can come up with a bunch of potential configurations for microservices and if nothing else that would be a useful thing to be able to take to management: these are the types of structures that I am considering, you know? Or you could take it to the development team and you could ask the development team, is this a good structure? Can you think of why this would not work? So it would just be really cool as a modeling and communication tool.»

*6) Summary:* To answer RQ2, our study identified several features practitioners want to see in microservice extraction tools, such as the ability to visualize and investigate relationships and possible microservice candidates obtained by utilizing these relationships, to customize elements and relationships considered by the extraction tool, to perform "what-if" analyses of possible decompositions, and to interactively work with the tool by "fixing" some decisions and letting the tool work with this setup. Notably, all study participants found this direction productive and beneficial for their needs, and we obtained comments, such as: «(P3) It makes sense for architects to use a tool like this to figure out how to [cluster]»; «(P2) If this tool becomes open source, I would be interested in taking a look at it and running it on my own applications.»; «(P5) I always imagined having a tool like this – I would love to try it out on my code.»

## V. Threats to Validity

For **external validity**, our results may be affected by the selection of our interview study participants. We attempted to mitigate this threat by reaching out to experienced software engineers from companies of different sizes and geographic locations. For the think-aloud protocol, the participants could be influenced by the case study application we experimented with. We attempted to mitigate this threat by selecting a representative third-party application, which was already successfully decomposed into microservices. We also made sure to emphasize during the interviews that we use the application as an example and are interested in the practitioners' opinions based on their broader experience. We believe that our selection of experienced software engineers with substantial microservice extraction experience helped to mitigate both threats.

For **internal validity**, we might have misinterpreted participants' answers or misidentified concepts. To mitigate this threat, we made sure at least two authors of this paper attended each interview and recorded all but one interview for further detailed analysis. Moreover, our data analysis was performed independently by two authors of the paper and all divergences were discussed and resolved by all the authors.

## VI. Related work

We discuss the related work along two main dimensions: migration to microservices and broader studies on architectural recovery techniques.

*1) Migration to Microservices:* Existing automated microservice extraction approaches are extensively discussed in Section II. A number of studies have also recently surveyed industrial practitioners, collecting challenges related to the migration from monolithic to microservice-based architectures [11], [43]. They observed that finding a proper service granularity and setting up an initial infrastructure for microservices are some of the migration challenges that developers face. Carvalho et al. [10] interviewed 15 microservice migration specialists, investigating the usefulness of criteria for microservice extraction, such as coupling, cohesion, communication overhead, reuse potential, database schema, and more. The results suggest that practitioners often need to consider multiple criteria simultaneously as well as their trade-offs to support their decisions. The participants also stated that existing tools are insufficient to support their microservice extraction decisions, which is also confirmed in additional studies [11], [17], [20], [47] Yet, unlike our work, these studies do not provide exact reasons for the lack of adoption of the tools and do not detail suggestions for support practitioners need to employ tools in their work environment.

Gysel et al. [21] also investigated service decomposition based on a number of coupling criteria distilled from the literature. Yet, this work focuses on design artifacts, such as use cases and Entity-Relationship Models, while our work considers development artifacts. Fritzsch et al. [18] proposed a decision guide recommending a certain tool based on developers' decomposition goal and available data. Unlike us, the study did not focus on factors affecting the availability of that data and did not provide suggestions for improving the tools.

*2) Architectural Recovery:* Microservice extraction is closely related to the field of architectural recovery, as both activities aim to extract architectural information from existing systems by grouping software entities into clusters. Abreu and Goulao [6], Bavota et al. [5], as well as Candela et al. [8] investigated how class coupling, as captured by structural, dynamic, semantic, and logical coupling measures, aligns with developers' perception of coupling. Similarly, Lutellier et al. [29], [30] explored the impact of relationships on architectural recovery approaches, focusing on structural relationships alone. Garcia et al. [19] performed a comprehensive analysis of software architecture recovery technique showing a relatively low accuracy for most of the analyzed approaches.

The authors of these works observed that none of the decomposition principles exclusively dominate the modularity of the studied systems. Our study confirms this finding. Yet, our focus is specifically on the microservice extraction process, which requires additional considerations when compared with architectural decomposition in general, e.g., the need to maintain independence of teams and their deployment schedules. To the best of our knowledge, our work is the first to investigate the impact and usefulness of a wide range of code relationships on the microservice extraction process in the context of practitioners' development practices.

## VII. Conclusion

In this paper, we investigated the usefulness of relationships extracted from a monolithic application for the microservice extraction process. We also collected expectations practitioners have for tools utilizing such relationships. Our study involved 10 practitioners with substantial hands-on microservice extraction experience. Our results show that practitioners often need a "what-if" analysis tool that simultaneously considers multiple relationship types. This is because the relevance of some relationship types is influenced by the choice of programming language, the age of the code base, language(s) spoken by the team, and more. We also extract suggestions for improving current extraction tools, e.g., with the ability to iteratively select, filter, and customize relationships, as well as to fix some decisions during the extraction process.

**Acknowledgments:** The work has been partially supported by IBM CAS Canada.

## REFERENCES

[1] "PartsUnlimitedMRP," https://github.com/microsoft/PartsUnlimitedMRP.

[2] "PartsUnlimitedMRP Microservices," https://github.com/microsoft/partsunlimitedMRPmicro.

[3] M. Abdullah, W. Iqbal, and A. Erradi, "Unsupervised Learning Approach for Web Application Auto-Decomposition into Microservices," *Journal of Systems and Software (JSS)*, vol. 151, pp. 243–257, 2019.

[4] L. Baresi, M. Garriga, and A. De Renzis, "Microservices Identification Through Interface Analysis," in *European Concerence on Service-Oriented and Cloud Computing (ESOCC)*, 2017, pp. 19–33.

[5] G. Bavota, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "An Empirical Study on the Developers' Perception of Software Coupling," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2013, pp. 692–701.

[6] F. Brito e Abreu and M. Goulao, "Coupling and Cohesion as Modularization Drivers: Are We Being Over-Persuaded?" in *European Conference on Software Maintenance and Reengineering (CSMR)*, 2001, pp. 47–57.

[7] B. Camburn, V. Viswanathan, J. Linsey, D. Anderson, D. Jensen, R. Crawford, K. Otto, and K. Wood, "Design Prototyping Methods: State of the Art in Strategies, Techniques, and Guidelines," *Design Science*, 2017.

[8] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using Cohesion and Coupling for Software Remodularization: Is It Enough?" *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, 2016.

[9] L. Carvalho, A. Garcia, T. E. Colanzi, W. K. G. Assunção, J. A. Pereira, B. Fonseca, M. Ribeiro, M. J. de Lima, and 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.

[10] L. Carvalho, A. Garcia, W. K. G. Assunção, R. de Mello, and M. Julia de Lima, "Analysis of the Criteria Adopted in Industry to Extract Microservices," in *Workshops of the IEEE/ACM International Conference on Software Engineering*, 2019, pp. 22–29.

[11] P. Di Francesco, P. Lago, and I. Malavolta, "Migrating Towards Microservice Architectures: an Industrial Survey," in *Proceedings of IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 29–38.

[12] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Cham: Springer International Publishing, 2017, pp. 195–216.

[13] 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)*, 2016, pp. 1–11.

[14] S. Eski and F. Buzluca, "An Automatic Extraction Approach: Transition to Microservices Architecture from Monolithic Application," in *Workshops of the International Conference on Agile Software Development*, 2018, pp. 1–6.

[15] European Commission, "Internal Market, Industry, Entrepreneurship and SMEs – SME Definition," https://ec.europa.eu/growth/smes/sme-definition_en.

[16] B. S. Everitt, S. Landau, and M. Leese, *Cluster Analysis*, 4th ed. Wiley, 2009.

[17] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices Migration in Industry: Intentions, Strategies, and Challenges," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 481–490.

[18] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From Monolith to Microservices: A Classification of Refactoring Approaches," in *DEVOPS Workshop*, 2018, pp. 128–141.

[19] J. Garcia, I. Ivkovic, and N. Medvidovic, "A Comparative Analysis of Software Architecture Recovery Techniques," in *ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 486–496.

[20] J. Ghofrani and A. Bozorgmehr, "Migration to Microservices: Barriers and Solutions," in *Applied Informatics*, 2019, pp. 269–281.

[21] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service Cutter: A Systematic Approach to Service Decomposition," in *European Conference on Service-Oriented and Cloud Computing (ESOCC)*, 2016, pp. 185–200.

[22] M. W. Jaspers, T. Steen, C. van den Bos, and M. Geenen, "The Think Aloud Method: A Guide to User Interface Design," *International Journal of Medical Informatics*, vol. 73, no. 11, pp. 781–795, 2004.

[23] 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 (TSE)*, 2019.

[24] A. K. Kalia, J. Xiao, C. Lin, S. Sinha, J. J. Rofrano, M. Vukovic, and D. Banerjee, "Mono2Micro: An AI-based Toolchain for Evolving Monolithic Enterprise Applications to a Microservice Architecture," in *Tool Demos of ESEC/FSE*, 2020, pp. 1606–1610.

[25] L. J. Kirby, E. Boerstra, Z. J. C. Anderson, and J. Rubin, "Supplementary Materials," https://doi.org/10.17605/OSF.IO/Y82G9.

[26] C. Lewis and J. Rieman, "Task-Centered User Interface Design: A Practical Introduction," *University of Colorado, Boulder, Department of Computer Science*, 1993.

[27] J. Lewis and M. Fowler, "Microservices: a Definition of This New Architectural Term," https://www.martinfowler.com/articles/microservices.html, 2014.

[28] 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 (JSS)*, 2019.

[29] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Comparing Software Architecture Recovery Techniques Using Accurate Dependencies," in *ACM/IEEE International Conference on Software Engineering (ICSE), Industrial Track*, 2015, pp. 69–78.

[30] ——, "Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques," *IEEE Transactions on Software Engineering (TSE)*, vol. 44, pp. 159–181, 2017.

[31] T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira, and A. Restivo, "Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis," in *Software Architecture*, 2020, pp. 315–332.

[32] G. Mazlami, J. Cito, and P. Leitner, "Extraction of Microservices from Monolithic Software Architectures," in *International Conference on Web Services (ICWS)*, pp. 524–531.

[33] Microsoft, "Chapter 1: Service Oriented Architecture (SOA)," https://web.archive.org/web/20160206132542/https://msdn.microsoft.com/en-us/library/bb833022.aspx, 2016.

[34] B. S. Mitchell and S. Mancoridis, "On the Automatic Modularization of Software Systems using the Bunch Tool," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 3, pp. 193–208, 2006.

[35] I. Pigazzini, F. A. Fontana, and A. Maggioni, "Tool Support for the Migration to Microservice Architecture: An Industrial Case Study," in *Software Architecture*, 2019, pp. 247–263.

[36] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from Monolithic Architecture to Microservices: A Rapid Review," in *International Conference of the Chilean Computer Science Society*, 2019, pp. 1–7.

[37] J. Ramos, "Using TF-IDF to Determine Word Relevance in Document Queries," in *Instructional Conference on Machine Learning*, 2003.

[38] Z. Ren, W. Wang, G. Wu, C. Gao, W. Chen, J. Wei, and T. Huang, "Migrating Web Applications from Monolithic Structure to Microservices Architecture," in *Asia-Pacific Symp. on Internetware*, 2018, pp. 1–10.

[39] Scientific Toolworks, "Understand," https://scitools.com/.

[40] A. Selmadji, A. Seriai, H. L. Bouziane, C. Dony, and R. O. Mahamane, "Re-architecting OO Software into Microservices," in *European Concerence on Service-Oriented and Cloud Computing (ESOCC)*, 2018, pp. 65–73.

[41] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage, 1998.

[42] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[43] 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.

[44] D. Taibi and K. Systä, "From Monolithic Systems to Microservices: a Decomposition Framework Based on Process Mining," in *International Conference on Cloud Computing and Services Science (CLOSER)*, 2019.

[45] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis," in *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2012, pp. 247–248.

[46] Y. Wang, H. Kadyala, and J. Rubin, *Journal of Empirical Software Engineering (EMSE)*, 2021.

[47] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang, "Microservice Architecture in Reality: An Industrial Inquiry," in *IEEE International Conference on Software Architecture (ICSA)*, 2019, pp. 51–60.