# Microservice Decomposition Techniques: An Independent Tool Comparison

Yingying Wang
Univ. of British Columbia, Canada
wyingying@ece.ubc.ca

Sarah Bornais
Univ. of British Columbia, Canada
sbornais@ece.ubc.ca

Julia Rubin
Univ. of British Columbia, 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 into microservices is a challenging task. A number of automated microservice decomposition techniques have been proposed in industry and academia to help developers with the migration complexity. Each of the techniques is usually evaluated on its own set of case study applications and evaluation criteria, making it difficult to compare the techniques to each other and assess the real progress in this field. To fill this gap, this paper performs an independent study comparing eight microservice decomposition tools that implement a wide range of different decomposition principles with each other on a set of four carefully selected benchmark applications. We evaluate the tools both quantitatively and qualitatively, and further interview developers behind two of the selected benchmark applications. Our analysis highlights strengths and weaknesses of existing approaches, and provides suggestions for future research, e.g., to provide differential treatment of application elements of different types, to customize the decomposition strategy and granularity per specific application, and more.

**ACM Reference Format:**
Yingying Wang, Sarah Bornais, and Julia Rubin. 2024. Microservice Decomposition Techniques: An Independent Tool Comparison. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695504

## 1 Introduction

Microservice-based architecture is a SOA-inspired principle of building complex systems as a composition of small, loosely coupled components that communicate with each other using lightweight technologies such as HTTP/REST [32]. This architectural principle has become increasingly popular in industry due to its advantages, such as greater development agility and improved scalability of deployed applications. With the increased popularity of microservice-based architectures, many companies invest in migrating their monolithic applications onto this more modern architecture.

The decomposition of a monolithic application into microservices requires substantial time and effort [17, 55, 57]. It entails sorting hundreds, if not thousands, of classes and methods into groups (potential microservices). A number of academic and industrial automated microservice decomposition techniques have been recently proposed to assist developers with this process [6, 8, 11, 16, 19, 20, 27–29, 33, 38–40, 45, 46, 48, 49, 58, 59]. These techniques, at large, capture relationships between elements of a monolithic application and further use these relationships to cluster similar elements together. The techniques vary by the type of elements and relationships they consider and the type of clustering they perform.

Each technique is typically evaluated on a set of case study applications, comparing it to a subset of other techniques using a subset of evaluation metrics chosen by the authors. Yet, there is no uniform comparison that performs a systematic selection of techniques, case studies, and metrics. Moreover, the evaluation often focuses on decomposition outputs, with little discussion given to the advantages and disadvantages of the underlying principles the evaluated techniques use.

Our paper aims to fill this gap, providing an independent systematic study that compares a diverse set of code-based microservice decomposition tools, both quantitatively and qualitatively. To this end, we first analyze the existing literature, identifying more than 60 papers describing microservice decomposition tools, which we further categorize based on the underlying principles these tools use. As running such a number of tools is impractical, we systematically narrow down our selection to eight tools, involving the developers of the evaluated tools in our study, to make sure the results we obtain are reliable. One of our selected tools, Mono2Micro [28], is a commercial offering by IBM [3]; the rest are developed in academia.

Next, we evaluate more than 20 monolithic applications, identifying those that satisfy the input requirements of all the selected tools, e.g., having a test suite with reasonably high coverage, using databases, etc. We also include a "challenge" open-source application that was not attempted by any of the tools in prior experiments. Overall, we use four case study applications in our analysis.

We further systematically select a set of quantitative metrics that align with tool decomposition goals, such as structural modularity, business use case separation, and database separation, to inform our analysis of decomposition results. We analyze the decomposition results produced by each tool both quantitatively and qualitatively, and further conduct two semi-structured interviews with the authors of the involved case study applications, to get their opinions about the decomposition.

When inspecting tool results, we interpret them w.r.t. the descriptions in the corresponding papers. As our team is not involved in the development of any of the tools, we believe our comparative analysis of the results is independent and unbiased. As a means to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ASE '24, October 27-November 1, 2024, Sacramento, CA, USA*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695504

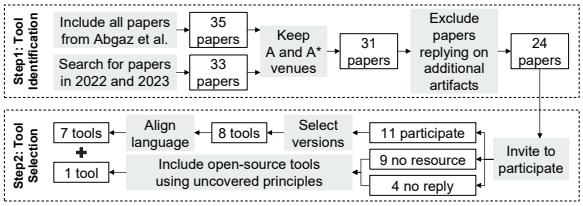


**Figure 1: Tool Selection Process.**

quality assurance, all authors of the paper discussed the results of each tool; we clarified with the tool authors results we could not adequately explain. In a few cases, this process led the authors to re-run their tool and send us updated results. Sections 2 and 3 provide more details about our selection of tools, case studies, metrics, and our evaluation methodology.

Our evaluation results show that decomposition techniques could benefit from the ability to identify application classes of different types, such as business capability, persistence layer, and helper classes, and treat them appropriately. The techniques could also benefit from developing strategies to decide when to duplicate classes, when to split by methods, etc. Adaptively selecting proper decomposition principles (by business use cases, linguistic similarity, databases, etc.) for a given application type (or even its parts) is also needed. Finally, decomposition techniques often fail due to limitations of the underlying tools they use, especially around static analysis and framework support (e.g., Spring). Evaluating and adapting ideas for practical use is often overlooked.

**Contributions.** This paper makes the following contributions:
1. It provides a unified benchmark of case study applications and metrics that are applicable for evaluating a wide range of microservice decomposition tools.
2. It performs the first in-depth independent analysis that compares microservice decomposition tools to each other on the same set of case study applications and metrics.
3. It identifies the main strengths and weaknesses of each tool and provides suggestions for future research.
4. It makes our empirical evaluation setup and results publicly available to facilitate replicability, reproducibility, and future work in this area [56].

We hope that such a comprehensive comparison will allow the community to better understand the strengths and weaknesses of existing solutions, facilitate development of more advanced approaches, and provide grounds for a more systematic evaluation of the existing and new tools.

## 2 Tools, Case Studies, and Metrics

We start from describing the microservice extraction tools and our process of selecting tools for our study. We then discuss our case study and metric selection.

### 2.1 Microservice Extraction Tools

To identify relevant tools, we started from a systematic review by Abgaz et al. [7], which includes 35 papers describing automated microservice decomposition tools that were published before the end of 2021. To extend this set of tools with more recent ones, in January 2024, we repeated the search using the methodology by Abgaz et al., covering the years 2022 and 2023. This resulted in 33

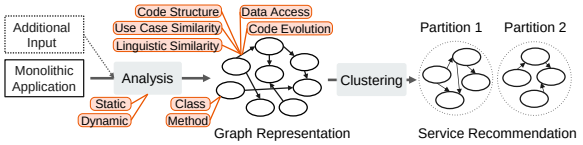


**Figure 2: Automated Microservice Decomposition.**

additional relevant tools. The exact search query that we used and our inclusion/exclusion criteria are available online [56].

As running almost 70 tools is impractical, we first selected only publications from A* and A venues according to the CORE ranking [1, 2] – an approach commonly used in the literature [18, 34, 47]. Two researchers then independently read each publication to extract the underlying principles each tool uses. We further excluded 7 tools that rely on additional artifacts that are often unavailable or require a considerable time to produce, e.g., entity-relationship and use case diagrams, as was also done in prior work [30]. This selection results in a set of 24 publications, as described in the upper part of Figure 1.

We observed that while each of the identified tools has unique characteristics, they largely follow the same general principle illustrated in Figure 2: they analyze a monolithic application statically and/or dynamically, to extract information deemed relevant to decompose the application. They encode this information in a graph whose nodes represent application entities, such as classes and methods, and edges represent relationships between the entities; they further cluster the graph into partitions, i.e., service recommendations, using existing or customized clustering algorithms.

The exact type of information the techniques extract depends on the heuristics that they use to partition the monolithic software and the goal they aim to achieve: high code modularity, encapsulation of business domains by services, etc. as discussed below:

**1. Code Modularity:** code structure captures control and data dependencies between code artifacts. This type of information groups together architecturally-related elements to increase code modularity and thus reduce inter-service communication and performance overhead associated with over-the-network inter-service calls. Code structure relationships can be extracted statically or dynamically. Static relationships include method calls [8, 11, 16, 19, 20, 27, 38, 45, 49, 51, 58, 59], class hierarchy [8, 20, 38, 46, 58, 59], package structure [46, 58], and variable dependencies [45, 58, 59]. The main dynamic code structure relationship type is dynamic method calls [11, 16, 27–29, 33, 51].

**2. Business Context Purity:** use case similarity relies on a predefined set of application business use cases, identifying code elements that are used to realize the same business use cases [16, 28, 29]. The idea behind these techniques is to support a single-responsibility principle, so that produced partitions contain elements belonging to the same (small) set of use cases.

In addition, linguistic similarity assumes that developers followed a certain naming convention, giving similar names to elements that belong to the same conceptual domain. Some of the techniques focus on class/method name similarity [27], while others compute broader term similarity, considering other terms within the class/method bodies, such as variable names, method names and parameters, comments, etc. [40, 46, 49, 51, 58]. Most of the techniques exclude stop words and language-specific technical terms.

**3. Database Transaction Purity:** data access considers dependencies between code elements and databases [8, 19, 20, 38, 45] and internal dependencies between database tables (via foreign keys) [48]. Data dependencies are obtained with the goal of grouping together elements that access the same database tables, thus minimizing distributed transactions within the decomposition.

**4. Team Independence:** code evolution captures the similarity between elements through the lens of the version control history. 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 [40]. More specifically, commit similarity groups together elements that are often changed together in the same commit, while contributor similarity groups together elements that are often updated by the same developer.

## 2.2 Tool Selection

When inspecting the 24 identified tools, we observed that many are not publicly available and many other require configuration and adaptations to run properly, e.g., to correctly configure as many as 11 different parameters [27] or adapt the class-labeling script for each considered case study [59]. Some tools only provide partial implementation and, thus, do not run out-of-the-box [20]. After experimenting with several tools for a few weeks each, we realized that such a process (a) cannot scale and (b) might bias the study if we configure tools incorrectly.

To evaluate this sheer number of tools in a fair and reliable manner, we decided to reach out to the tool authors and involve them in our study, while focusing our efforts on an unbiased selection of benchmark applications and in-depth analysis of results. Specifically, we first asked the tool authors whether they would agree to run their tools on a set of open-source case study projects, given that we will (a) ensure each project is appropriate for their tool in terms of the implementation language and framework, the required information types, test coverage, etc., (b) provide guidelines on how to compile and run the projects and their tests, and (c) offer any help and technical support handling the projects, as needed.

We received a positive confirmation from the authors of 11 tools from eight research groups. The authors of an additional nine publications replied to our inquiry but did not have the resources to participate in the study; the authors of four publications did not reply, even after a reminder. From the replies we received, two groups contributed different versions of the same tool: one group with two tools [33, 60] and another one – with three [49–51]. As the tools from each group were versions of each other, we left the authors the freedom to run their best-performing tool. Furthermore, as all but one participating tool worked for Java applications, to enable a meaningful comparison, we had to exclude the single tool that works for Python [39]. In fact, out of the 24 identified tools, 18 support Java, 3 are language-agnostic, 2 support PHP, and 1 – Python. Thus, using Java is a reasonable choice for our comparison.

Finally, after inspecting the obtained set of seven tools, we decided to include one additional publicly available tool which we ran ourselves [40], to make sure the selected tools cover all popular microservice decomposition principles. This is the only tool in our collection that relies on code evolution (and optimizes for team independence) when decomposing to microservices.

This methodology led us to include eight tools listed in Table 1 and described below. To the best of our knowledge, our study involves more tools than any other comparisons. Our tool selection also provides a diverse and representative set of microservice decomposition approaches that rely on static and dynamic analysis, work on class and method levels, and optimize for a variety of goals, such as code modularity, business context purity, team independence, and database transaction purity. One of these tools, Mono2Micro [28], is a commercial offering by IBM [3]; the rest are developed in academia. We describe these tools next.

**Mono2Micro** [28] relies on business use cases and structural code relationships to create class-level decompositions that optimize for business context purity and code modularity. Specifically, it analyzes business use case execution traces, where each use case trace can be obtained either by running a set of tests or by having the user manually trigger the application to execute a business use case. The tool performs hierarchical clustering to group together classes that are structurally similar and that contribute to the implementation of the same business use cases. Apart from monolith application artifacts and business use cases, the tool can be configured with a desired number of microservices to generate; this number is set to five by default. A recent commercial release of the tool [4] includes additional support for automated detection of utility classes based on a number of heuristics, such as the ratio of incoming/outgoing method calls, the ratio of methods/fields in a class, and the frequency of static class methods and fields.

**HyDec** [51] uses static and dynamic analysis to extract method calls, variable dependencies, and inheritance relationships. It further extracts linguistic similarity between classes, considering all terms defined within classes, such as class, field, and method names. Combining this information, HyDec performs hierarchical clustering to produce class-level decompositions that optimize code modularity and business context purity. Different from Mono2Micro, HyDec does not require the tests to correspond to business use cases but uses linguistic similarity to identify elements that belong to the same business domain.

**Data-Centric** [48] looks at relationships between classes and database tables. It statically analyzes the database schema of a monolithic application to extract cross-table data accesses (foreign key relationships) and linguistic similarities among database tables. Based on this information, it partitions the database tables of the monolithic application and further creates a class-level microservice decomposition based on transitive structural relationships between application classes and database tables. This decomposition optimizes for code modularity and database transaction purity.

**Log2MS** [33] analyzes execution logs to label classes of the monolithic application according to the following order of priorities: (a) controller classes that serve as externally accessed application endpoints; (b) data-related classes that map to or interact with database tables; and (c) other classes that process business logic. It then assigns each controller class to a separate partition and further groups business-logic-related classes into partitions with which they have the highest affinity, based on dynamic method call relationships. Finally, the tool adds data-related classes into these partitions. Similar to HyDec, Log2MS does not assume tests correspond to business use cases and does not require additional input, besides the application artifacts.

**Table 1: Participating Tools.**

| Tool | Granularity | Analysis | Optimization Goal | Relationship Type | Additional Input |
|---|---|---|---|---|---|
| Mono2Micro [28] | Class | Dynamic | Code Modularity, Business Context Purity | Code Structure (method calls), Use Case Similarity | business use cases, number of microservices |
| HyDec [51] | Class | Static and Dynamic | Code Modularity, Business Context Purity | Code Structure (method calls, variable dependencies, inheritance), Linguistic Similarity | - |
| Data-Centric [48] | Class | Static | Code Modularity, Database Transaction Purity | Code Structure (method calls), Data Access | database schema |
| Log2MS [33] | Class | Dynamic | Code Modularity | Code Structure (method calls) | - |
| MEM [40] | Class | Static | Business Context Purity, Team Independence | Linguistic Similarity, Code Evolutionary (commit, contributor) | number of microservices |
| toMicroservices [11] | Method | Static and Dynamic | Code Modularity, Business Context Purity | Code Structure (method calls), Use Case Similarity | business use cases, number of microservices |
| CARGO [45] | Method | Static | Code Modularity, Database Transaction Purity | Code Structure (method calls, variable dependencies), Data Access | number of microservices |
| MOSAIC [23] | Method | Static | Code Modularity | Code Structure (method calls, variable dependencies, inheritance) | - |

**MEM** [40] mines the version history of the monolithic application to extract the frequency of a file being modified in the same commit or by the same contributor. Similar to HyDec, it also extracts linguistic similarity between classes, considering all terms defined within classes. Using one or multiple types of the collected information, MEM constructs a weighted graph of the monolithic application, with the extracted information encoded as edge weights and clusters this graph to create microservice candidates. The tool optimizes for team independence, lifecycle independence, and business context purity. In our analysis, we focused on the first two cases: the frequency of files being modified in the same commit (denoted by MEM-CMT) and the frequency of files being modified by the same contributor (denoted by MEM-CNTR).

**toMicroservices** [11] performs a method-level decomposition, optimizing for code modularity and business context purity. The tool employs static and dynamic analyses to collect structural information from code and use case execution traces. It treats decomposition as a multi-objective optimization problem, considering four criteria: coupling, cohesion, inter-partition communication overhead (all of which correlate with code modularity), and feature modularity (which correlates with business context purity). In our analysis, we focused on the output decomposition with the highest feature modularity, to compare the results with those of Mono2Micro, which optimizes for the same goal. Like several other tools, toMicroservices relies on the user to specify the desired number of microservices to produce.

**CARGO** [45] relies on static analysis and performs a method-level decomposition to optimize for code modularity and database transaction purity. Unlike Data-Centric, it does not rely on a database schema. Instead, it performs context-sensitive static analysis to extract call and data dependencies between methods and data access relationships between methods and database tables. It then assigns a unique label to all methods of each class and utilizes a label propagation algorithm to assign the same label to (a) methods that share transaction edges with the same database table and (b) methods that are closely related based on method call and variable dependency relationships. The tool also requires as an additional input the desired number of microservices and uses it as a stopping condition for the label propagation.

**MOSAIC** [23] assumes a layered architecture of monolithic applications and statically analyzes the application to identify entity, logic-layer, and persistence-layer classes (using annotations in code). It focuses the decomposition on these classes only, disregarding the remaining non-business-logic-related classes. The tool optimizes for code modularity and performs a hybrid class- and method-level decomposition. It constructs a graph with two types of nodes – a class node for each entity class and a method node for each method in the logic- and persistence-layer classes; it then runs a community detection algorithm to first partition entity classes and then to determine which methods should be grouped together with each entity class. A unique feature of MOSAIC is its ability to detect classes and methods that should be duplicated to multiple partitions: it identifies entity classes that are extended by other entity classes but do not have their corresponding persistence-layer classes and duplicates them into the partitions of the extending classes. It also identifies methods of the persistence layer that do not have outgoing calls and are only called by methods from the same class; it duplicates them into partitions of the calling method.

## 2.3 Case Study Selection

We started by collecting a list of all monolithic applications used as case studies by at least one of the 24 identified microservice extraction tools. This resulted in 62 applications. Focusing on those that have (a) publicly-available Java source code and (b) documentation and setup instructions written in English resulted in 22 applications. Out of the 22, only eight have the corresponding microservice decomposition produced manually by developers. As we intended to use this manually-produced decomposition to facilitate our analysis of results, we proceed to the next step with these eight applications.

To satisfy the requirements of tools that rely on dynamic analysis, we built each of the applications and executed its corresponding functional test suites. We only included those applications that have at least 60% statement-level coverage – a common industrial guideline [10, 13]. To satisfy the requirement of the tools that rely on business use cases for decomposition purposes, we selected case studies with at least three functional tests (approximated these as high-level use cases). To satisfy the requirements of the tools that use database relationships, we selected case studies with at least three database tables. Finally, to satisfy the requirements of the tool that relies on data from version histories, we selected case studies with at least 100 commits and at least three contributors.

This process narrowed down our selection to three case study applications: JPetStore [44], Spring-PetClinic [54], and PartsUnlimitedMRP [41]. These applications, together with their properties, are listed in the first three rows of Table 2. Interestingly, only two out of the eight tools we selected for our study used these applications for their evaluation: MEM used Spring-PetClinic and MOSAIC used both Spring-PetClinic and JPetStore.

**Table 2: Case Studies.**

| Case Study | Monolith | | | | | | | | | Microservice | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Version | # Cls. | # Meth. | SLOC | Test Cvg. | # Use Cases | # DB Tbl. | # Cmit. | # Contrib. | # Svc. | # Cls. | # Meth. |
| **JPetStore** | 4a07a02 | 24 | 298 | 1409 | 64% | 3 | 13 | 1334 | 21 | 3 | 26 | 304 |
| **Spring-PetClinic** | 1079767 | 23 | 90 | 752 | 94% | 4 | 7 | 837 | 101 | 3 | 31 | 112 |
| **PartsUnlimitedMRP** | a83586b | 53 | 459 | 4407 | 65% | 5 | 5 | 791 | 44 | 5 | 73 | 525 |
| **7ep-demo** | 7fdc2b0 | 47 | 266 | 2326 | 93% | 4 | 4 | 1047 | 4 | 4 | 70 | 378 |

To further challenge the tools with new and realistic case studies, in January 2023, we searched for open-source monolithic Java applications in GitHub. We started by identifying the 100 most starred GitHub repositories which contain Java web applications that have at least one commit in the last two years (GitHub Search Query: "web application" OR "web app", language:Java, pushed:>2021-01-01, archived:False, orderby:stars). We then excluded repositories not using English for instructions and documentation, frameworks and tutorials for building web applications, projects that are already implemented as microservices, projects with limited business use cases and test coverage, and projects without databases. This left us with three projects.

As these projects do not have an existing reference microservice-based version available, we reached out to their owners, asking whether they are willing to review and provide feedback on the decompositions of their web apps to microservices produced by automated tools. We received a positive reply from the 7ep-demo owner and included this project in our study. The details about the 7ep-demo application are given in the last row of Table 2; the full list of all repositories we considered is available in our online appendix [56]. Overall, our four selected case studies span multiple application domains. We describe them in more detail below.

**JPetStore** [44] is an online pet store application, which consists of 24 classes and 298 methods implementing account, catalog, and order functionality of the store. It has 13 database tables and 10 functional test cases. We excluded five tests that check the user interface and do not exercise any business logic. We mapped each of the remaining five tests to one of the three application business use cases according to the JPetStore documentation. We relied on these functional tests to produce per-business-use-case information for Mono2Micro and toMicroservices. Overall, these five functional tests provide a statement-level coverage of 64%.

The reference microservices-based version of this application was developed in prior work by researchers and three experienced software engineers [59]. It includes three partitions, *Account Service*, *Catalog Service*, and *Order Service*, which contain all classes of the monolithic application. The microservice-based version also includes two additional classes with six methods; these classes are, in fact, duplicates of one abstract class from the monolithic version that was added to all of the partitions.

**Spring-PetClinic** [54] is a pet clinic management system built in Spring, which implements owner, pet, vet, and visit management use cases of the store. It consists of 23 classes with 90 methods, 13 database tables, and 4 functional tests that correspond to the 4 business use cases of the application. The functional tests provide a statement-level coverage of 94%.

The Spring-PetClinic team also developed a microservice-based version of this application [53], which contains three services incorporating the application business logic: *Customers/Pets*, *Vets*, *Visit*, and four additional services that implement microservice-related infrastructure, such as API Gateway and Discovery services. We only focused on the business-logic-related services for our reference microservice-based version. As the owner and pet management use cases are highly overlapping, they were merged into one partition in the microservice-based version. Moreover, as with the JPetStore application, four classes were duplicated into two additional partitions, resulting in a version with 31 classes and 112 methods.

**PartsUnlimitedMRP** [41] is a Manufacturing Resource Planning application developed and maintained by Microsoft. We focus our analysis on the backend part of PartsUnlimitedMRP. It contains 53 classes with 459 methods, which conceptually correspond to five functional domains: catalog, dealer, order, quote, and shipment. The application has five NoSQL documents and five functional tests, each corresponding to one functional domain of the system. Its functional tests provide a statement-level coverage of 65%.

The microservice-based version of PartsUnlimitedMRP was also developed by Microsoft [42]. It contains five services that represent the application's five business domains, and two additional infrastructure services, which we excluded from our analysis, like in the case of Spring-PetClinic. Our resulting reference microservice-based version consists of 73 classes and 525 methods; 20 of these classes (with 66 methods) are duplicates of 7 classes in the monolithic version (exceptions and utilities).

**7ep-demo** [5] is a web application developed for demonstration purposes. It contains four main functionalities: authentication, library management, mathematics, and database management. It consists of 47 classes with 266 methods, 4 database tables, and 12 functional tests that correspond to the four business use cases of the application. The tests provide a statement-level coverage of 93%.

7ep-demo does not have an existing reference microservice-based version available. The project owner we engaged with instead preferred to decompose the application by business use cases, which results in four partitions: *Authentication*, *LibraryManagement*, *Mathematics* and *DBManagement*. The owner also decided to duplicate 10 classes with a total of 42 methods (nine utility classes and one application bootstrap class); one additional class and the interface it implements were split into three different partitions. The resulting reference microservice-based version consists of 70 classes with 378 methods.

Target number of produced partitions. Four of the tools in our study, Mono2Micro, MEM, CARGO, and toMicroservices, require the desired number of partitions as an input. We asked the tool owners to produce decompositions with the same number of partitions as the reference microservice-based version, namely, 3, 3, 5, and 4 services for JPetStore, Spring-PetClinic, PartsUnlimitedMRP, and 7ep-demo, respectively.

## 2.4 Metrics Selection

We use two types of metrics to inform and complement our qualitative analysis of decompositions produced by the evaluated tools: metrics that (a) evaluate microservice quality w.r.t. their design principles, and (b) compare a decomposition produced by a tool to the reference decomposition produced manually by developers.

**Microservice Design Principles.** We started from the ten microservice design principles for evaluating microservice-based architectures collected by Engel et al. [22]. We excluded scalability

and performance, as we cannot assess these properties for the non-runnable decompositions produced by the tools. We also excluded maintainability, as we cannot reliably assess how the team maintaining the produced decompositions will deem it.

We mapped the remaining seven principles (Size; Cyclic Dependencies; Structural Modularity; Network Complexity; Business Context Modularity; Domain Independence; Team, Lifecycle, and Technology Independence) to a set of quantitative metrics used in the literature. Specifically, when possible, we relied on the relevant metrics used in the evaluations of the tools included in our study. However, we noticed that the tool authors occasionally use different calculations for even seemingly identical metrics, e.g., Business Context Modularity in [28, 45]. We thus unified and cleaned up the definitions, when necessary.

Partition Size measures the minimum, maximum, mean, and median number of elements in all partitions of a decomposition. The rationale behind these statistical metrics is to assess whether the produced microservice candidates are reasonably small and their sizes are evenly distributed, e.g., that a tool does not produce one excessively large and a few tiny partitions.

Cyclic (In-)Dependence (*CiD*) corresponds to the principle stating that microservice networks should be free of cyclic dependencies. We say two partitions $p$ and $q$ have a cyclic dependency iff there is at least one call from a method in $p$ to a method in $q$ and there is at least one call from a method in $q$ to a method in $p$. Then, this metric measures the fraction of partition pairs that do not have such cyclic dependency.

Code Modularity (*CMod*) measures the coupling and cohesion of the produced decomposition w.r.t. structural relationships between partitions. Low partition coupling and high cohesion correspond to more efficient, scalable, and maintainable software. As such, all tools in our study optimize for better code modularity. We calculate *CMod* using Turbo Modularization Quality (*TurboMQ*) [43], which combines coupling and cohesion into a single metric. It is defined over a graph whose nodes are partition elements and edges are relationships between these elements. For method-level decompositions, we calculate the metric over a static method-level call graph, where there exists a directed edge between methods $M_i$ and $M_j$ iff $M_i$ calls $M_j$; each edge in this graph has a weight of 1. For class-level decompositions, we use a static class-level call graph, where there is a directed edge with the weight $w_C$ between classes $C_i$ and $C_j$ iff the total number of calls between methods of $C_i$ and $C_j$ is $w_C$. We denote by $\mu_i$ the sum of intra-partition edge weights for a partition $i$ and by $\varepsilon_{ij}$ – the sum of inter-partition edge weights from partition $i$ to $j$. Then, the Cluster Factor for a partition $i$, $CF_i$ is $\frac{2\mu_i}{2\mu_i+\sum_j \varepsilon_{ij}+\varepsilon_{ji}}$ and the normalized *TurboMQ* is defined as the average of all Cluster Factors for all partitions in a decomposition. The value of *CMod* varies between 0% and 100%, with higher values indicating more modular decompositions. We use this metric to assess both Structural Modularity and Network Complexity properties, as network complexity is directly proportional to the coupling between partitions.

Business Context Purity (*BCP*) [28, 29, 45] measures how business use cases are distributed across partitions, which is one of the optimization goals for three tools in our study: Mono2Micro, HyDec, and toMicroservices. These works define business use case distribution in terms of the number of business use cases each partition handles. A high *BCP* thus means that each partition handles a small number of business use cases, which represents adherence to the Single Responsibility Principle [37].

Calculating *BCP* relies on Shannon entropy [52], which assesses the degree of randomness in a set of samples. Here, samples represent methods in a partition and *BCP* measures the probability of these methods to implement a business use case. Specifically, for each partition, the probability that a method in the partition implements a given business use case $b$ is defined as $P(b) = n_b/n$, where $n_b$ is the number of times a method in the partition appears in the execution log of business use case $b$ and $n$ is the total number of times a method in the partition appears in the execution log of any of the business use cases. The business use case entropy for a partition, $H(i)$, is then the entropy over the probability distribution $P(b)$ for all business use cases in an application: $-\sum_b P(b) \times log(P(b))$. Further, *BCP* is defined as $1 - H$, where $H$ is the average $H(i)$ for all partitions in a given decomposition. *BCP* values can range from 0 to 100, where a higher *BCP* value indicates that each partition "implements" a smaller number of business use cases. *BCP* is calculated the same way for both method-level and class-level decompositions.

Domain Independence (*DI*) measures how many partitions handle each business use case, which represents how well decompositions adhere to the by-business-domain decomposition principle behind microservice-based architectures [32]. Its calculations also rely on the entropy of a set of samples. However, in this case, the samples are methods in an execution log of a given business use case and we calculate the probability that a use case is implemented in a partition $p$, $P'(p) = n_p/n$, where $n_p$ is the number of methods in a partition $p$ that implement a use case and $n$ is the total number of partitions that implement the use case.

Database Transaction Purity (*DTP*) [45] measures how database accesses are distributed across partitions, which is one of the optimization goals for two tools in our study: Data-Centric and CARGO. Having each partition interact with fewer databases is yet another principle behind microservice-based architectures [32]. Similar to *BCP*, *DTP* is defined based on the average entropy of partitions that access a given table across all database tables in a given decomposition. For each database table, the probability that a database transaction originates from a partition $p$ is $P(p) = n_p/n$, where $n_p$ is the number of times partition $p$ accesses the table, either directly or indirectly, and $n$ is the total number of database transactions that access the given table. That is, the metric aims to penalize situations where multiple partitions access the same database and we used it as one way to assess the Technology Independence principle. *DTP* is calculated the same way for both method-level and class-level decompositions.

Team – Contributors (*TC*) measures how frequently elements historically modified by the same developer are assigned to the same partition, which is one of the optimization goals of MEM. We use *TC* to assess the Team Independence principle, as a team working on a produced microservice will be more independent if all elements the team members need to modify are included in the same service. We calculate *TC* using *TurboMQ*, defined over a graph whose nodes are partition elements; there exists an edge between two elements iff these elements are modified by the same developer,

where the weight of this edge represents the number of developers who modified both entities. The value of *TC* varies between 0% and 100%, with higher values indicating elements frequently modified by the same developers are indeed placed in the same partition.

Lifecycle – Commits (*LC*) measures how frequently elements historically modified in the same commit are assigned to the same partition, which is also yet another optimization goal of MEM. We use *LC* to assess the Lifecycle Independence principle, as entities that are committed together are more likely to go through the same development lifecycle and, thus, should be grouped in the same service. Like *TC*, we calculate *LC* using *TurboMQ* over a graph whose nodes are partition elements and a weighted edge between two elements indicates the number of commits in which both elements are modified.

**Comparison of produced and reference decompositions.** We used two popular metrics commonly used in the literature on architectural refactoring and microservice decomposition for measuring the distance between two architectures of the same software system: *MoJoFM* [61] and Cluster-to-Cluster Coverage ($c2c_{cvg}$) [24, 35, 36].

*MoJoFM* measures the distance between two architectures of the same software system. In our study, it takes as input the automatically produced and the reference decompositions, *P* and *R*, and quantifies the number of *Move* and *Join* operations needed to transform *P* into *R*. The *Move* operation moves an entity from a partition to an existing or a newly created one. The *Join* operation joins two partitions into one, reducing the total number of partitions. *MoJoFM* scores range from 0% to 100%, wherein a higher value represents a higher similarity between two decompositions. As the metric assigns the same weight to both operations, it tends to prefer more fine-grained decompositions, where small partitions of the produced architecture can be merged into a larger partition of the reference architecture.

Cluster-to-cluster Coverage ($c2c_{cvg}$) is a complementary metric that measures the degree of overlap between the produced and reference decompositions, *P* and *R*. It first calculates the similarity between each partition in *P* and each partition in *R* as a fraction of common elements in *P* and *R* over the size of the larger of these two partitions: $\frac{|p_i \cap r_j|}{max(|p_i|,|r_j|)} \times 100\%$. $c2c_{cvg}$ is then defined as the fraction of partitions in *P* that are at least $th_{cvg}$-similar to at least one partition in *R*, where $th_{cvg}$ is a threshold that can be set to a certain percentage – typically, 10% (some overlap), 33% (moderate overlap), and 50% (high overlap) [36]. For example, a high $c2c_{cvg}$ 50% value implies that there is a large fraction of partitions in the produced decomposition that are at least 50% similar to at least one partition in the reference decomposition.

## 3 Evaluation Methodology

We evaluated the decomposition results proposed by the tools both qualitatively and quantitatively. For the quantitative evaluation, we used the metrics defined in Section 2.4.

As there could be multiple valid ways to decompose an application, we also qualitatively analyzed all decompositions to extract the strengths and weaknesses of each. Specifically, we conducted two types of qualitative evaluation. First, two authors of this paper independently analyzed the decomposition results produced by each tool for each of the case studies. Then, all authors of the paper met to discuss all decompositions. When analyzing the decompositions, we aimed to map the properties of the produced result to those of the analyzed tools. The goals of our analysis were to (a) perform a "sanity check", interpreting the results of the tools given our understanding of the mechanisms behind their implementations; (b) compare the results produced by different tools with each other; and (c) compare the results with the manual decomposition of the application provided by developers. We shared our observations with the tool authors, which, in a few cases, led to re-running the tools with an updated configuration setup and small bug fixes. We incorporated the fixes for further analysis.

To gain further insights into how practitioners view decompositions provided by the tools and collect additional practical considerations about their usefulness, we reached out to project owners and top contributors of our case study applications, asking whether they would be willing to review and provide feedback on the decompositions of their project produced by automated tools. We received positive replies from the developers of two applications: Spring-PetClinic and 7ep-demo. These developers are senior software engineers working in industry. One has 13 years of software development experience and 9 years of experience in microservices and cloud-based software. Another has 45 years of software development experience, 12 years of experience in microservices and cloud-based software, and is one of the founders of Spring Cloud, Spring Boot, and Spring Batch.

To maximize productivity and use the developers' time efficiently, we started the interaction by providing a pre-recorded video describing the purpose of our study and the principles behind existing microservice extraction techniques. We also shared the interview consent form and other documents for collecting the qualifications and demographic information offline.

After this initial communication, we conducted a semi-structured interview with each developer individually. Each interview lasted one and a half hours. We started the interview by briefly reviewing the background information about microservice decomposition tools. The remainder of the interview was driven by the following high-level questions: (1) What does the developer consider as a good decomposition; (2) How would they decompose their corresponding application; (3) What is their opinion on each of the decompositions produced by the tools; and (4) Whether and how they changed their mind about the properties of a good decomposition and the desired decomposition of their application after reviewing the decompositions proposed by the tools.

The interviews were recorded (with the consent of the developers), which gave us the ability to reliably transcribe the collected information. We also followed up with the developers by email after the interview, when we believed clarifications were needed. We shared the summary of the interviews and a draft of this paper with the developers, for their final approval. This study protocol was approved by the ethics board in our institution.

## 4 Results

We now discuss the performance of each tool on the four considered case studies. We then summarize the lessons learned and implications of our findings in Sections 5.

Table 3 shows the summary of the metrics we calculated to inform our tool evaluation. We first present the metrics for each

**Table 3: Evaluation Results for the Decomposition Metrics.**

| Case Study | Tool | #Partitions | Partition Size | | | | # Obs. Ents. (%) | *CiD* | *CMod* | *BCP* | *DI* | *DTP* | *TC* | *LC* | *MoJoFM* | $c2c_{cvg}$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Min | Mean | Med | Max | | | | | | | | | | 10% | 33% | 50% |
| **JPetStore** | *Reference (class-level)* | 3 | 5 | 8.7 | 10 | 11 | 24 (100%) | 66.7 | 71.4 | 50.5 | 45.4 | 85 | 33.1 | 33.1 | - | - | - | - |
| | Mono2Micro | 3* | 4 | 5.7 | 5 | 8 | 17 (70.8%) | 0 | 68 | 33.6 | 16.5 | 57 | **33.1** | **33.1** | 57.1 | **100** | 75 | 50 |
| | HyDec | 8 | 2 | 2.8 | 2 | 5 | 22 (91.7%) | **96.4** | 32 | **51.5** | 10.2 | 59.1 | 12.1 | 12.1 | 57.1 | **100** | 33.3 | 11.1 |
| | Data-Centric | 3 | 5 | 8 | 6 | 13 | 24 (100%) | 66.7 | **70.8** | 49.8 | 30.4 | 79.7 | **33.1** | **33.1** | 76.2 | **100** | **100** | **100** |
| | Log2MS | 3 | 4 | 8 | 10 | 10 | 24 (100%) | 66.7 | 69.9 | 48.9 | **43.6** | **85** | **33.1** | **33.1** | **100** | **100** | **100** | **100** |
| | MEM-CMT | 5 | 2 | 3.4 | 3 | 6 | 17 (70.8%) | 100 | 7.8 | 26.2 | 0 | 54.3 | 19.8 | 19.8 | 38.1 | 83.3 | 33.3 | 0 |
| | MEM-CNTR | 6 | 2 | 3.3 | 2.5 | 6 | 20 (83.3%) | 80 | 14.1 | 35.9 | 0 | 19.3 | 16.4 | 16.3 | 33.3 | 100 | 0 | 0 |
| | *Reference (method-level)* | 3 | 73 | 103.3 | 105 | 132 | 302 (100%) | 66.7 | 77.5 | 50.5 | 59.8 | 96.2 | 33.1 | 33.1 | - | - | - | - |
| | toMicroservices | 3* | 8 | 76.3 | 8 | 213 | 229 (75.8%) | 66.7 | 34.6 | **61.2** | **95.1** | 59.7 | 33 | 33 | 51.3 | 50 | 50 | 0 |
| | CARGO | 3* | 7 | 25.7 | 23 | 47 | 77 (25.5%) | 66.7 | 52.1 | 44 | 52 | 83.4 | 32.4 | 32.4 | 46.7 | 75 | 25 | 0 |
| | MOSAIC | 3 | 57 | 96 | 107 | 124 | 288 (95.4%) | **100** | **80.3** | 40.2 | 58.8 | **96.2** | **33.1** | **33.1** | **89.5** | **100** | **75** | **75** |
| **Spring-PetClinic** | *Reference (class-level)* | 3 | 6 | 9 | 10 | 11 | 23 (100%) | 100 | 85.4 | 48.9 | 65.4 | 70.7 | 34.4 | 38.1 | - | - | - | - |
| | Mono2Micro | 3* | 3 | 5.7 | 7 | 7 | 17 (73.9%) | 66.7 | **77.8** | 33.3 | 28.5 | 64.9 | 34.6 | 39.3 | 36.8 | **100** | 75 | 25 |
| | HyDec | 3 | 4 | 5.3 | 5 | 7 | 16 (69.6%) | **100** | 42.3 | **36.3** | **40.4** | 47.2 | **36** | **43.2** | 36.8 | **100** | 75 | 25 |
| | Data-Centric | 3 | 3 | 7.7 | 6 | 14 | 23 (100%) | 66.7 | 64.2 | 13 | 30.9 | **77.1** | 33.5 | 36.1 | **57.9** | **100** | **100** | **66.7** |
| | Log2MS | 7 | 1 | 2.7 | 2 | 6 | 19 (82.6%) | 85.7 | 29.2 | 9.8 | 3.3 | 57.1 | 14.1 | 16.9 | 52.6 | **100** | 37.5 | 0 |
| | MEM-CMT | 6 | 2 | 2.5 | 2 | 4 | 15 (65.2%) | 100 | 36.9 | 45 | 1.2 | 43.3 | 17.9 | 24.4 | 31.6 | 100 | 28.6 | 0 |
| | MEM-CNTR | 5 | 1 | 2.6 | 2 | 5 | 13 (56.5%) | 100 | 13.3 | 20 | 27.5 | 44.3 | 23.7 | 31.2 | 31.6 | 83.3 | 33.3 | 0 |
| | *Reference (method-level)* | 3 | 27 | 44 | 30 | 75 | 112 (100%) | 66.7 | 79.8 | 48.9 | 65.3 | 73 | 34 | 36.8 | - | - | - | - |
| | toMicroservices | 3* | 9 | 22.7 | 9 | 50 | 68 (60.7%) | **100** | 60.3 | 33.5 | 55.5 | 68.1 | 32.7 | 33.9 | 56.5 | **100** | 50 | 25 |
| | CARGO | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| | MOSAIC | 2 | 28 | 54 | 54 | 80 | 95 (84.8%) | **100** | **100** | **59.8** | **100** | **100** | **50.8** | **53.7** | **69.4** | **100** | **100** | **100** |
| **Parts Unlimited MRP** | *Reference (class-level)* | 5 | 5 | 12.4 | 13 | 17 | 53 (100%) | 100 | 90.2 | 29.7 | 12.7 | 57.5 | 20.2 | 20.1 | - | - | - | - |
| | Mono2Micro | 5* | 1 | 6.2 | 7 | 10 | 31 (58.5%) | **60** | 42.9 | **40.7** | 19.5 | 33 | 20 | 19.9 | 33.3 | 83.3 | 0 | 0 |
| | HyDec | 3 | 5 | 15.3 | 5 | 36 | 46 (86.8%) | 33.3 | **89.4** | 10.8 | **37** | **87.3** | **33.2** | **33.2** | 37.5 | **100** | 75 | 0 |
| | Data-Centric | 4 | 7 | 13 | 10 | 25 | 52 (98.1%) | 33.3 | 71.7 | 1.5 | 8 | 75.3 | 25 | 25 | **50** | 80 | **80** | **40** |
| | Log2MS | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| | MEM-CMT | 2 | 2 | 2 | 2 | 2 | 4 (7.5%) | 100 | 0 | 1.7 | 59.5 | 100 | 49.1 | 49.1 | 25 | 33.3 | 0 | 0 |
| | MEM-CNTR | 12 | 1 | 3.8 | 3 | 8 | 44 (83%) | 89.4 | 11.5 | 11.4 | 0 | 8.8 | 8.2 | 8.2 | 22.9 | 61.5 | 7.7 | 0 |
| | *Reference (method-level)* | 5 | 42 | 82.2 | 95 | 111 | 412 (100%) | 90 | 89.7 | 29.7 | 24.2 | 81.8 | 20.1 | 20.1 | - | - | - | - |
| | toMicroservices | 5* | 9 | 32.2 | 9 | 125 | 161 (39.1%) | 60 | 29.5 | **80** | **79.2** | 2.8 | 19.4 | 19.2 | 29.3 | 33.3 | 0 | 0 |
| | CARGO | 5* | 8 | 38.4 | 20 | 122 | 192 (46.6%) | 60 | 35.1 | 6.1 | 26.5 | 58.8 | 19.6 | 19.5 | 36.7 | 83.3 | 16.7 | 0 |
| | MOSAIC | 5 | 36 | 59.4 | 69 | 78 | 297 (72.1%) | **90** | **90.8** | 29.3 | 49.1 | **78.1** | **20.2** | **20.1** | **86.2** | **100** | **100** | **100** |
| **7ep-demo** | *Reference (class-level)* | 4 | 13 | 17.5 | 18 | 21 | 47 (100%) | 100 | 100 | 53.3 | 93.9 | 57.6 | 24.4 | 44.6 | - | - | - | - |
| | Mono2Micro | 4* | 3 | 7.5 | 4 | 19 | 30 (63.8%) | **100** | **80.3** | **83** | **60.4** | **84.1** | **24.6** | **48.4** | 51.2 | **100** | 40 | 20 |
| | HyDec | 8 | 3 | 5 | 3.5 | 12 | 40 (85.1%) | 96.4 | 47.1 | 68.8 | 37.2 | 8.4 | 12 | 34.8 | **69.8** | 88.9 | 22.2 | 11.1 |
| | Data-Centric | 4 | 2 | 11 | 9 | 24 | 44 (93.6%) | 33.3 | 38 | 77.2 | 51.4 | 43.9 | **24.6** | 38.9 | 62.8 | 60 | **60** | **40** |
| | Log2MS | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| | MEM-CMT | 7 | 1 | 2.9 | 3 | 6 | 20 (42.6%) | 100 | 25.4 | 55.1 | 35.3 | 23 | 13.2 | 43.9 | 37.2 | 62.5 | 12.5 | 12.5 |
| | MEM-CNTR | 18 | 1 | 1.8 | 2 | 4 | 33 (70.2%) | 97.4 | 2.1 | 55.6 | 0 | 0 | 4.8 | 13.7 | 20.9 | 26.3 | 5.3 | 0 |
| | *Reference (method-level)* | 4 | 76 | 100.8 | 93.5 | 140 | 295 (100%) | 100 | 96.5 | 53.3 | 77.9 | 37.5 | 24.6 | 47.1 | - | - | - | - |
| | toMicroservices | 4* | 8 | 30.3 | 8.5 | 96 | 121 (41%) | 50 | 35 | 72.5 | 68.4 | 44.4 | 24.3 | 27.6 | 36.2 | 40 | 40 | 0 |
| | CARGO | 4* | 6 | 29 | 11 | 89 | 117 (39.7%) | 50 | 40.6 | 29.9 | 60.1 | **82.7** | 24.3 | 31.5 | 36.2 | 60 | 40 | 0 |
| | MOSAIC | 4 | 6 | 38 | 36 | 73 | 151 (51.2%) | **100** | **100** | **80** | **100** | 43.7 | **24.5** | **45.7** | **64.5** | **80** | **80** | 0 |
| **Summary** | *Reference (class-level)* | 4 | 7.3 | 11.9 | 12.8 | 15 | 36.8 (100%) | 91.7 | 86.8 | 45.6 | 54.4 | 67.7 | 28 | 34 | - | - | - | - |
| | Mono2Micro | 3.8* | 2.8 | 6.3 | 5.8 | 11 | 23.8 (66.8%) | 56.7 | **67.3** | **47.7** | **31.2** | 59.8 | 28.1 | **35.2** | 44.6 | 95.8 | 47.5 | 23.8 |
| | HyDec | 6 | 3.5 | 7.1 | 3.9 | 15 | 31 (83.3%) | **81.5** | 52.7 | 41.9 | **31.2** | 50.5 | 23.3 | 30.8 | 50.3 | 97.2 | 51.4 | 11.8 |
| | Data-Centric | 4 | 4.3 | 9.9 | 7.8 | 19 | 35.8 (97.9%) | 50 | 61.2 | 35.4 | 30.2 | 69 | **29.1** | 33.3 | 61.7 | 85 | **85** | **61.7** |
| | Log2MS | 5 | 2.5 | 5.4 | 6 | 8 | 21.5 (91.3%) | 76.2 | 49.6 | 29.4 | 23.5 | **71.1** | 23.6 | 25 | **76.3** | **100** | 68.8 | 50 |
| | MEM-CMT | 5 | 1.8 | 2.7 | 2.5 | 4.5 | 14 (46.5%) | 100 | 17.5 | 32 | 24 | 55.2 | 25 | 34.3 | 33 | 69.8 | 18.6 | 3.1 |
| | MEM-CNTR | 10.3 | 1.3 | 2.9 | 2.4 | 5.8 | 27.5 (73.3%) | 91.7 | 10.3 | 30.7 | 6.9 | 18.1 | 13.3 | 17.4 | 27.2 | 67.8 | 11.6 | 0 |
| | *Reference (method-level)* | 4 | 54.5 | 82.6 | 80.9 | 114.5 | 280.3 (100%) | 80.9 | 85.9 | 45.6 | 56.8 | 72.1 | 28 | 34.3 | - | - | - | - |
| | toMicroservices | 3.8* | 8.5 | 40.4 | 8.6 | 121 | 144.8 (54.2%) | 69.2 | 39.9 | **61.8** | 74.6 | 43.8 | 27.4 | 28.4 | 43.3 | 55.8 | 35 | 6.3 |
| | CARGO | 4* | 7 | 31.1 | 18 | 86 | 128.7 (37.3%) | 58.9 | 42.6 | 26.7 | 46.2 | 75 | 25.4 | 27.8 | 39.9 | 72.8 | 27.2 | 0 |
| | MOSAIC | 4 | 31.8 | 61.8 | 66.5 | 88.8 | 207.8 (75.9%) | **97.5** | **92.8** | 52.3 | **77** | **79.5** | **32.2** | **38.2** | **77.4** | **95** | **88.8** | **68.8** |

* indicates cases where tools were given the number of partitions as an additional input.

case study separately, to enable the comparison of the tools on the same underlying code base. For each case study, we also show the values of metric calculations for the reference decomposition, for class-level and method-level versions, separately. Then, in the "Summary" row, we present the average performance for each tool across all four case studies.

After the names of the case studies and the tools, the third column of the table shows the number of partitions in the reference and produced decompositions. We mark by * cases where tools were given the number of partitions as an additional input. The "# Observed Entities" column shows the number and the fraction of elements from the original monolithic application that are included in a decomposition produced by a tool (unobserved entities occur because of static and dynamic unreachability, tool bugs, and more.)

The next two major sections of the table show the values of the microservice quality metrics (*CiD*, *CMod*, *BCP*, *DI*, *DTP*, *TC*, and *LC*) and the results of the comparison between the produced and reference decompositions for each case study (*MoJoFM* and $c2c_{cvg}$). We boldface the best value for each metric in each of the case studies, for class- and method-level decompositions separately.

**Mono2Micro.** The tool works well for case studies where business use cases are well separated, e.g., 7ep-demo. However, as could be expected, it does not perform as well for case studies with highly entangled business use cases, e.g., JPetStore (see *BCP*). It also tends

to sacrifice the by-use-case decomposition (*DI*) for the single responsibility principle (*BCP*). The tool successfully identifies some of the utility classes in our case studies. Yet, it does not provide special treatment to exceptions, factories, and abstract base classes, all of which were duplicated in the reference decompositions. This is also the primary reason Mono2Micro does not achieve the same level of code modularity as the reference decompositions (*CMod*), despite having some of the best overall scores among all class-level tools, as shown in the "Summary" part at the bottom of the table.

**HyDec.** Unlike Mono2Micro, this tool uses linguistic similarity to identify business domains, relying on the assumption that developers use the same terminology when naming elements from the same domain. Similar to all other tools, it also optimizes for code modularity. Considering linguistic similarity works particularly well in the 7ep-demo application, where the tool is able to correctly split the *Authentication* and *LibraryManagement* partitions, as they indeed use distinct terminology. Many other tools end up unifying these partitions because of their strong code-level dependencies. However, one weakness of the tool is its inability to identify contextually-related yet linguistically-dissimilar terms, which leads to the creation of a decomposition that is too fine-grained. For example, for both JPetStore and 7ep-demo, the tool created eight service candidates instead of three and four in the reference decompositions, respectively, making the result largely dissimilar to the reference architecture (see *MoJoFM* and $c2c_{cvg}$). Likewise, in PartsUnlimitedMRP, the tool pulled many unrelated elements into one partition, resulting in a high modularity score (*CMod*) but low similarity with the reference decomposition. Building a proper ontology capturing application-specific terminology and using it to compute linguistic similarity could help largely improve the quality of the tool.

**Data-Centric.** Despite optimizing for database access relationships, this tool does not achieve the highest *DTP* in all but one of our case studies. However, it achieves an overall high similarity to the reference decomposition (see *MoJoFM* and $c2c_{cvg}$ in the "Summary" part of the table). This is particularly evident in PartsUnlimitedMRP, where Data-Centric has the highest *MoJoFM* score of the tools; in this application, there is a one-to-one mapping of database tables to the associated partitions, providing evidence that developers consider database tables when decomposing applications.

**Log2MS.** The authors only provided us with the decomposition results for the JPetStore and Spring-PetClinic case studies; we thus exclude the other two case studies from further analysis for this tool. For JPetStore, Log2MS produced a result that is highly similar (almost identical) to the reference decomposition (both *MoJoFM* and all $c2c_{cvg}$ scores of 100%). This is because JPetStore has only four controller classes and the reference decomposition has four partitions. The tool starts by assigning each controller class to its own partition and then merges highly-coupled partitions; it thus can achieve a result similar to the reference decomposition. However, this strategy does not work for Spring-PetClinic, which has six controller classes. As these classes are rather disconnected, Log2MS produces more partitions than the expected three, resulting in low similarity with the reference decomposition.

**MEM.** Our experience running the tool shows that it produces drastically different decompositions in each run. While investigating possible causes of this indeterminism, we observed that the tool is configured to consider only 12 files of each commit; we changed this number to the maximal possible – 30 (a restriction set by the *powerset* function of Google's Guava [25] library the tool uses, to avoid powerset sizes exceed the 'int' range). We further increased the maximal size of each partition to 25 rather than the default 10 as, otherwise, the tool produced many (different) fine-grained partitions. Even after these adjustments, the results produced by the tool were indeterministic. Lacking support from developers, we could not reliably repair the tool. For completeness, we report the best result out of five runs of the tool for each case study in Table 3 and exclude the tool from further qualitative analysis.

**toMicroservices.** This tool performs method-level decomposition. While this design allows the tool to split bloated classes into multiple partitions, it also often splits methods that are grouped together in the reference decomposition. Like Mono2Micro, toMicroservices optimizes for business use case purity. We observe that the tools tend to create unevenly sized partitions, with one large partition that serves the majority of the business decomposition and several small partitions. This is particularly apparent in PartsUnlimitedMRP, where the tool creates one large partition (125 methods) and four smaller partitions (9 methods each). It does this because it groups methods that support the same business use case together, and many methods support multiple business use cases. While this decision makes it possible for the tool to achieve high *BCP* and *DI* scores (with means of 61.8% and 74.6%, respectively, across all use cases), optimizing solely for business purity metric does not create good decompositions in this case. The produced decompositions also have low similarity with the reference ones.

**CARGO**. Overall, this tool fails to achieve reasonable results on our case studies due to the incomplete static analysis it performs. The tool relies on Doop [15] to perform reachability analysis. However, Doop appears to deem many reachable methods as unreachable. For example, in the JPetStore application with 302 methods, CARGO's reachability analysis reports only 97 methods as reachable and the tool further observes only 77 of them (25.5% of all methods). Method-level coverage for this application, calculated using the provided test suite is, in fact, 78.1%, which is also confirmed by the high fraction of methods observed by toMicroservices (75.8%). For Spring-PetClinic, the tool produced a result which included only framework and no application methods, which led us to exclude this case study altogether. The reachability issue resulted in CARGO being able to observe only 37.3% of the methods in our case studies, on average – lowest of all tools (see the "Summary" part of the table). Upon a discussion with the authors, they confirmed that more investigation is required to better understand why the tool behaves in this manner and how to update it to reach more application elements.

**MOSAIC**. This tool produces decompositions that are highly similar to the reference ones: *MoJoFM* of 77.4%, on average, the highest across all tools). The decompositions produced by the tool are also close, or even higher, than the reference decompositions w.r.t. other quality metrics. There are two factors contributing to the success of the tool: first, the decision to perform hybrid class- and method-level analysis allowed the tool to split classes only when necessary. The tool's ability to duplicate classes and methods among partitions is also remarkable and contributes to its success. However, its decision

to focus on entity, logic-layer, and persistence-layer classes only (potentially manually annotated) causes it to miss several classes, especially in the 7ep-demo case, where the tool could only observe a bit more than half of the code elements. We note that, per our communication with the authors, the results they shared with us are produced by the next release of the tool, which is not publicly available yet, so we could not verify how the improved features of that tool version contributed to the quality of its results.

# 5 Lessons Learned and Implications

We discuss the main observations from our study and suggestions for future research in two dimensions: observations about the tool evaluation process and observations about the tools themselves.

## 5.1 Tool Evaluation Process

**Relying on Metrics.** Our study shows that solely relying on metrics (or a subset of metrics, as done in some prior work) does not provide a comprehensive view and is insufficient to properly evaluate the tools. For example, pulling many classes into one partition can result in a high modularity score, as in the case of HyDec for PartsUnlimitedMRP. Likewise, observing only a small fraction of monolithic application elements can result in high metric values, as in the case of CARGO for *DTP*, but that does not translate to high-quality decompositions. Overreliance on (a subset of) metrics also prevents a reliable comparison of tools with each other. A holistic approach that qualitatively analyzes the produced decompositions and considers values of different metrics in combination, like we performed in this paper, is expensive but necessary.

**Comparison with Manually-Produced Decompositions.** As there could be multiple valid ways to decompose an application into microservices, the comparison with manually-produced decomposition should also be augmented with a qualitative discussion of the results. Even though in our experiments we did not observe any produced decomposition that appeared to be a reasonable alternative to the reference ones we used, such situation could occur in other studies.

**Non-functional and Execution-time Properties.** A usable decomposition should be performant, scalable, and secure. Unfortunately, none of the tools produces a runnable decomposition, hindering the analysis of these properties. Creating deployable microservices, as well as creating performance-, security-, and scalability-friendly decompositions, is a fruitful direction for future work. Such work could entail converting code dependencies into inter-service code, setting up API gateways, porting the app into a particular framework (e.g., Spring), adjusting the decomposition to avoid passing sensitive information over the internet, and more.

**Tool Performance.** Another missing aspect of evaluation is the performance of the tools. As our analysis shows, many tools utilize static analysis, which causes them to miss application elements and might not scale well for larger case studies. Another possible performance bottleneck is clustering, when the number of nodes becomes large (especially for method-level techniques). Our experience is that clustering can take several days/weeks for large applications [14]. Properly evaluating the performance and scalability of existing tools would be useful for possible future tool comparisons.

## 5.2 Tools

**Heuristics Employed by the Tools.** Tools use a variety of heuristics to decompose applications by business domains: while all tools consider static and/or dynamic relationships between elements, they also rely on using use case log executions (Mono2Micro and toMicroservices), linguistic similarity (HyDec), database relationships (Data-Centric and CARGO), and version histories (MEM). Our analysis shows that (1) using use case execution logs is helpful when use cases do not cross-cut multiple business domains; (2) linguistic similarity can be a useful approximation of business domains when developers use consistent and precise terminology, but an ontology of terms might need to be carefully defined for other cases; (3) using database decomposition to infer code decomposition is beneficial as database decomposition provides valuable insights into the business domains embedded in the application.

Even though we carefully selected case study applications that contain sufficient data for evaluating each tool, we observed that some applications can benefit from some decomposition principles but not all applications can benefit from all decomposition principles. Future work could look at dynamically adapting optimization goals and granularity to the properties of the decomposed application or even its different parts. MOSAIC makes a valuable first step in this direction, with its hybrid class- and method-level analysis.

Interestingly, our interviewees generally preferred class-level to full method-level decompositions ( «I probably would not go down to the layer of methods. Unless we are working in a manner that I would find to be less than professional, we ought to have extremely related work in classes»). However, they saw method-level decomposition as a good opportunity to obtain input for a possible refactoring and noted that method-level decomposition tools can be useful as a «quality problem indicator».

**Types of Application Elements.** In the same vein, we also observe that not every part of a monolithic application contributes to a business use case directly. For example, in the 7ep-demo application, persistence and helper classes provide services to different business use cases. Such classes should be considered separately from business classes. Both Mono2Micro and MOSAIC already took the first steps towards identifying such classes, which is remarkable. However, not all non-business-logic-related classes should be treated the same. As stated by one of our interviewees, some of the classes, e.g., in the persistence layer, are «kitchen drawers» of similar operations provided to different domains. Such classes «could be split by different needs [domains]». Other types of classes, like helpers, are «often meant to be cross-cutting helpful things, like convert strings to bytes. [They] needs to run really fast and be just available in a flash». As such, they should be duplicated to all partitions.

Duplication is not always the best solution and, pragmatically, using a shared library could be «better than duplication, even though it is an anti-pattern [in microservices]». One way to decide which classes should be duplicated vs. kept in a shared library is to look at the class evolution: «if [things] do not change much, go with a shared library». Future decomposition tools should look into identifying application classes of different types, such as business capability, persistence layer, and helper classes, and treat them appropriately. The tools should also develop strategies to decide when to duplicate classes, when to split by methods, etc.

**One-shot Approach to Decomposition.** We observed that several of the manually-produced reference microservice-based architecture underwent a refactoring during the decomposition process. In fact, both interviewed developers pointed out that the process of decomposition can induce such a refactoring. Even though the refactoring can, in theory, be done before the decomposition, the ROI is low: «It's a case of cost and benefit and the amount of energy it takes to do that refactoring. I wouldn't spend that much energy if I didn't think that I was going to refactor it into services. It's just not worth it.» Yet, most microservice decomposition techniques, including all those we analyzed in this study, perform one-shot decomposition. As the decomposition process is typically entangled with refactoring, future work dedicated to performing these processes simultaneously could be of high practical value.

**Developers-in-the-loop.** Developers often start decomposition knowing what microservices they desire: «I've already made clear that in my mind, authentication and library and mathematics are the domains». Moreover, the decomposition process is often iterative and requires human-in-the-loop, as reported in literature [21] and confirmed by our interviewees: «I would then go back and make modifications until [the tool] is saying 'this is all red and this is all blue [partitions]'». Another productive research direction could be exploring iterative approaches that keep developers in the loop, which will help complement the information missed by the analysis.

**Implementation Issues.** Several microservice decomposition tools implement advanced and useful ideas for helping developers with decomposition tasks. However, they tend to fail because of implementation issues, especially issues related to static analysis of monolithic applications and the need to support multiple frameworks, such as Spring Boot and Spring MVC. This was the main reason for failures in tools such as CARGO, MOSAIC, and more. While the difficulties of conducting an accurate analysis are well known [31], evaluating and adapting ideas for practical use is often overlooked. Future work should look into the potential of developing analysis-based techniques that will scale for large, industrial applications and the return on investment (ROI) in such development.

## 6 Limitations and Threats to Validity

For **external validity**, our results may not generalize beyond our selected case study applications. We attempted to mitigate this threat by selecting applications of different sizes and domains, ensuring they satisfy the input requirements of all participating tools. We included applications previously decomposed to microservices and an additional "challenge" case study, 7ep-demo, that was not decomposed before. We acknowledge that the selected case studies do not fully reflect the scale of industry applications. Yet, we opted to use open source applications for transparency and reproducibility. We thus believe our selection of applications is reliable and representative.

For **internal validity**, we might have misinterpreted the tools' results. We mitigated this threat in several ways: first, two authors of the paper analyzed the results of all tools on all case studies independently and cross-validated their observations. We also reached out to the tool owners when unsure. Finally, we shared a draft of the paper with all tool authors to obtain their feedback and addressed all feedback we received. Similarly, to mitigate the treat of misinterpreting the statements of practitioners we interviewed, all authors of this paper attended the interviews and both interviews were recorded for further detailed analysis. Our interview data analysis was performed independently by two authors of the paper and discussed by all the authors. We shared a draft of this paper with the practitioners and addressed all feedback we received.

## 7 Discussion and Related Work

**Internal tool evaluation.** The authors of three tools participating in our study, namely HyDec [51], Log2MS [33], and CARGO [45], compared their approaches with Mono2Micro [28, 29], as well as with earlier tools such as MEM [40] and FoSCI [27]. Data-Centric [48] and toMicroservices [11, 16] were not compared with these or other approaches before. However, the existing comparisons do not use a consistent set of case studies and metrics.

Our work fills these gaps by performing a comprehensive selection of case studies and a uniform set of metrics appropriate for evaluating a wide range of tools. Moreover, by including eight tools in our study, we were able to compare closely related tools with each other, e.g., CARGO and Data-Centric, which both optimize for database transaction purity; Mono2Micro and toMicroservices, which both optimize for business use case purity; and toMicroservices and CARGO, which both work on the method level. Such comparison was not attempted before.

Finally, only a few works perform qualitative evaluations with developers [11, 16, 28]. Such evaluations are mostly focused on the usefulness of a particular tool rather than attempting to compare multiple tools with each other, like we do in our work.

**External tool evaluation.** We are only aware of one work in which the authors collected a number of automated microservice decomposition techniques and evaluated their usability and performance [9]. The authors included only three tools in their study, which are all substantially older (published in 2016 and 2017) than the tools we used in our work: Service cutter [26], MEM [40], and Decomposer [12]. Our study complements and extends this work by exploring a large set of contemporary tools, allowing us to provide suggestions for additional necessary future work in this area.

## 8 Conclusion

In this paper, we reported on the results of our study comparing eight microservice decomposition tools on a set of systematically selected benchmark applications and metrics. As our team was not involved in developing these tools, we provide an independent assessment of the tools' performance, evaluating them both qualitatively and quantitatively. We also engaged with developers who are the main contributors of two of our benchmark applications, to collect their insights on the decomposition process and results. We hope that such a comprehensive comparison will allow the community to better understand the strengths and weaknesses of existing solutions, facilitate development of more advanced approaches, and provide grounds for a more systematic evaluation of the existing and new tools.

## 9 Acknowledgments

We would like to thank the authors of all tools and the owners of Spring-PetClinic and 7ep-demo projects for their involvement in our study. We also thank the anonymous reviewers for their insightful feedback, which helped us improve the paper.

## References


[1] 2020. CORE ranking (Journal Portal). http://portal.core.edu.au/jnl-ranks/.
[2] 2021. CORE ranking (Conference Portal). http://portal.core.edu.au/conf-ranks/.
[3] 2022. IBM - Mono2Micro. https://www.ibm.com/cloud/mono2micro.
[4] 2024. What's new in IBM Mono2Micro. https://www.ibm.com/docs/en/mono2micro?topic=overview-whats-new-in-mono2micro.
[5] 7ep. [n. d.]. Demo - demonstrates an application and tests. https://github.com/7ep/demo.
[6] Muhammad Abdullah, Waheed Iqbal, and Abdelkarim Erradi. 2019. Unsupervised Learning Approach for Web Application Auto-Decomposition into Microservices. *Journal of Systems and Software (JSS)* 151 (2019), 243–257.
[7] Yalemisew Abgaz, Andrew McCarren, Peter Elger, David Solan, Neil Lapuz, Marin Bivol, Glenn Jackson, Murat Yilmaz, Jim Buckley, and Paul Clarke. 2023. Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering* (2023), 1–32.
[8] Shivali Agarwal, Raunak Sinha, Giriprasad Sridhara, Pratap Das, Utkarsh Desai, Srikanth Tamilselvam, Amith Singhee, and Hiroaki Nakamuro. 2021. Monolith to Microservice Candidates using Business Functionality Inference. In *2021 IEEE International Conference on Web Services (ICWS)*. 758–763.
[9] Kerem Akkaya and Tolga Ovatman. 2022. A Comparative Study of Meta-Data-Based Microservice Extraction Tools. *International Journal of Service Science, Management, Engineering, and Technology (IJSSMET)* 13, 1 (2022), 1–26.
[10] Carlos Arguelles, Marko Ivanković, and Adam Bender. 2020. Code Coverage Best Practices. https://testing.googleblog.com/2020/08/code-coverage-best-practices.html
[11] Wesley K. G. Assunção, Thelma Elita Colanzi, Luiz Carvalho, Juliana Alves Pereira, Alessandro Garcia, Maria Julia de Lima, and Carlos Lucena. 2021. A Multi-Criteria Strategy for Redesigning Legacy Features as Microservices: An Industrial Case Study. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 377–387.
[12] Luciano Baresi, Martin Garriga, and Alan De Renzis. 2017. Microservices Identification Through Interface Analysis. In *European Concerence on Service-Oriented and Cloud Computing (ESOCC 2017)*. 19–33.
[13] Amit Kanti Barua. 2022. Test Coverage Definition - Unit Testing. https://learn.microsoft.com/en-us/answers/questions/778016/test-coverage-definition-unit-testing
[14] Evelien Boerstra, John Ahn, and Julia Rubin. 2022. Stronger Together: On Combining Relationships in Architectural Recovery Approaches. 305–316.
[15] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses *(OOPSLA '09)*.
[16] Luiz Carvalho, Alessandro Garcia, Thelma Elita Colanzi, Wesley K. G. Assunção, Juliana Alves Pereira, Baldoino Fonseca, Márcio Ribeiro, Maria Julia de Lima, and Carlos Lucena. 2020. On the Performance and Adoption of Search-Based Microservice Identification with toMicroservices. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 569–580.
[17] L. Carvalho, A. Garcia, W. K. G. Assunção, R. de Mello, and M. Julia de Lima. 2019. Analysis of the Criteria Adopted in Industry to Extract Microservices. In *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*. 22–29.
[18] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. 2023. Data Preparation for Software Vulnerability Prediction: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 49 (2023), 1044–1063.
[19] Adambarage Anuruddha Chathuranga De Alwis, Alistair Barros, Colin Fidge, and Artem Polyvyanyy. 2021. Microservice Remodularisation of Monolithic Enterprise Systems for Embedding in Industrial IoT Networks. In *Advanced Information Systems Engineering*. 432–448.
[20] Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth Tamilselvam. 2021. Graph Neural Network to Dilute Outliers for Refactoring Monolith Application. In *Thirty-Fifth AAAI Conference on Artificial Intelligence*. 72–80.
[21] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. 2018. Migrating Towards Microservice Architectures: an Industrial Survey. In *Proceedings of IEEE International Conference on Software Architecture (ICSA)*. 29–38.
[22] Thomas Engel, Melanie Langermeier, Bernhard Bauer, and Alexander Hofmann. 2018. Evaluation of Microservice Architectures: A Metric and Tool-Based Approach. In *International Conference on Advanced Information Systems Engineering (CAiSE)*.
[23] Gianluca Filippone, Nadeem Qaisar Mehmood, Marco Autili, Fabrizio Rossi, and Massimo Tivoli. 2023. From Monolithic to Microservice Architecture: An Automated Approach Based on Graph Clustering and Combinatorial Optimization. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. 47–57.
[24] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. 2013. A Comparative Analysis of Software Architecture Recovery Techniques. In *International Conference on Automated Software Engineering (ASE)*. 486–496.
[25] Google. 2024. Google core libraries for Java. https://github.com/google/guava
[26] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. 2016. Service Cutter: A Systematic Approach to Service Decomposition. In *Service-Oriented and Cloud Computing*. 185–200.
[27] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. 2019. Service Candidate Identification from Monolithic Systems based on Execution Traces. *IEEE Transactions on Software Engineering (TSE)* (2019).
[28] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. 2021. Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1214–1224.
[29] Anup K. Kalia, Jin Xiao, Chen Lin, Saurabh Sinha, John J. Rofrano, Maja Vukovic, and Debasish Banerjee. 2020. Mono2Micro: An AI-based Toolchain for Evolving Monolithic Enterprise Applications to a Microservice Architecture. In *Tool Demos of ESEC/FSE*. 1606–1610.
[30] Lisa J. Kirby, Evelien Boerstra, Zachary J.C. Anderson, and Julia Rubin. 2021. Weighing the Evidence: On Relationship Types in Microservice Extraction. In *International Conference on Program Comprehension (ICPC)*. 358–368.
[31] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study. In *International Conference on Software Engineering (ICSE)*. 507–518.
[32] James Lewis and Martin Fowler. 2014. Microservices: a Definition of This New Architectural Term. https://www.martinfowler.com/articles/microservices.html.
[33] Bo Liu, Jingliu Xiong, Qiurong Ren, Shmuel Tyszberowicz, and Zheng Yang. 2022. Log2MS: A Framework for Automated Refactoring Monolith Into Microservices Using Execution Logs. In *2022 IEEE International Conference on Web Services (ICWS)*. 391–396.
[34] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep Learning for Android Malware Defenses: A Systematic Literature Review. *Comput. Surveys* 55 (2022).
[35] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. 2015. Comparing Software Architecture Recovery Techniques Using Accurate Dependencies. In *International Conference on Software Engineering (ICSE)*. 69–78.
[36] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. 2017. Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques. *IEEE Transactions on Software Engineering* 44 (2017), 159–181.
[37] Robert C. Martin. 2014. The Single Responsibility Principle. https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html. [Online; accessed February 2024].
[38] Alex Mathai, Sambaran Bandyopadhyay, Utkarsh Desai, and Srikanth Tamilselvam. 2022. Monolith to Microservices: Representing Application Software through Heterogeneous Graph Neural Network. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)*. 3905–3911.
[39] T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira, and A. Restivo. 2020. Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis. In *European Conference on Software Architecture*. 315–332.
[40] Genc Mazlami, Jurgen Cito, and Philipp Leitner. 2017. Extraction of Microservices from Monolithic Software Architectures. In *International Conference on Web Services (ICWS)*. 524–531.
[41] microsoft. [n. d.]. Parts Unlimited MRP. https://github.com/microsoft/partsunlimitedMRP.
[42] microsoft. https://github.com/microsoft/partsunlimitedMRPmicro. Parts Unlimited MRP Microservices.
[43] B. S. Mitchell and S. Mancoridis. 2006. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering* 32, 3 (March 2006), 193–208.
[44] MyBatis.org. [n. d.]. JPetstore Demo 6 - MyBatis Spring. http://mybatis.org/jpetstore-6/.
[45] Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. 2023. CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Article 20, 12 pages.
[46] I. Pigazzini, F. A. Fontana, and A. Maggioni. 2019. Tool Support for the Migration to Microservice Architecture: An Industrial Case Study. In *Software Architecture*. 247–263.
[47] Pooja Rani, Arianna Blasi, Nataliia Stulova, Sebastiano Panichella, Alessandra Gorla, and Oscar Nierstrasz. 2023. A Decade of Code Comment Quality Assessment: A Systematic Literature Review. *Journal of Systems and Software* 195 (2023), 111515.
[48] Yamina Romani, Okba Tibermacine, and Chouki Tibermacine. 2022. Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. 15–19.
[49] Khaled Sellami, Ali Ouni, Mohamed Aymen Saied, Salah Bouktif, and Mohamed Wiem Mkaouer. 2022. Improving Microservices Extraction Using Evolutionary Search. *Information and Software Technology* 151 (2022), 106996.
[50] Khaled Sellami, Mohamed Aymen Saied, and Ali Ouni. 2022. A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications. In *Proceedings of the 26th International Conference on Evaluation and Assessment in*

*Software Engineering (EASE)*. 201–210.

[51] Khaled Sellami, Mohamed Aymen Saied, Ali Ouni, and Rabe Abdalkareem. 2022. Combining Static and Dynamic Analysis to Decompose Monolithic Application into Microservices. In *International Conference on Service-Oriented Computing (ICSOC)*. 203–218.

[52] C. E. Shannon. 1948. A mathematical theory of communication. *The Bell System Technical Journal* 27, 3 (1948), 379–423.

[53] spring petclinic. [n. d.]. Distributed version of Spring Petclinic built with Spring Cloud. https://github.com/spring-petclinic/spring-petclinic-microservices.

[54] spring projects. [n. d.]. Spring PetClinic Sample Application. https://github.com/spring-projects/spring-petclinic.

[55] D. Taibi and V. Lenarduzzi. 2018. On the Definition of Microservice Bad Smells. *IEEE Software* 35, 3 (2018), 56–62.

[56] Yingying Wang, Sarah Bornais, and Julia Rubin. 2024. Microservice Decomposition Techniques: An Independent Tool Comparison. https://resess.github.io/artifacts/MicroserviceToolStudy/.

[57] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. 2021. Promises and Challenges of Microservices: an Exploratory Study. *Journal of Empirical Software Engineering (EMSE)* (2021).

[58] Kaiyuan Yang, Junfeng Wang, Zhiyang Fang, Peng Wu, and Zihua Song. 2022. Enhancing Software Modularization via Semantic Outliers Filtration and Label Propagation. *Information and Software Technology* 145 (2022), 106818.

[59] Pascal Zaragoza, Abdelhak-Djamel Seriai, Abderrahmane Seriai, Anas Shatnawi, and Mustapha Derras. 2022. Leveraging the Layered Architecture for Microservice Recovery. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. 135–145.

[60] Yukun Zhang, Bo Liu, Liyun Dai, Kang Chen, and Xuelian Cao. 2020. Automated Microservice Identification in Legacy Systems with Functional and Non-Functional Metrics. In *2020 IEEE International Conference on Software Architecture (ICSA)*. 135–145.

[61] Zhihua Wen and V. Tzerpos. 2004. An Effectiveness Measure for Software Clustering Algorithms. In *IEEE International Workshop on Program Comprehension (WPC)*. 194–203.