A Comprehensive Microservice Extraction Approach Integrating Business Functions and Database Entities

Article *in* The International Arab Journal of Information Technology · January 2024

DOI: 10.34028/iajit/21/1/3

3 authors, including:

Deepali Bajaj
University of Delhi
22 PUBLICATIONS **183** CITATIONS

A Comprehensive Microservice Extraction Approach Integrating Business Functions and Database Entities

Deepali Bajaj
Department of Computer Science, University of Delhi, India
deepali.bajaj@rajguru.du.ac.in

Anita Goel
Department of Computer Science, University of Delhi, India
goel.anita@gmail.com

Suresh Gupta
Department of Computer Science, Indian Institute of Technology, India
guptadrsc@gmail.com

Abstract: *Cloud application practitioners are building large-scale enterprise applications as microservices, to leverage scalability, performance, and availability. Microservices architecture allows a large monolithic application to be split into small, loosely coupled services. A service communicates with other services using lightweight protocols such as RESTful APIs. Extracting microservices from the monolith is a challenging task and is mostly performed manually by system architects based on their skills. This extraction involves both: 1) Partitioning of business logic, 2) Partitioning of database. For partitioning of business logic, the existing research studies focus on decomposition by considering the dependencies in the application at the class-level. However, with the passage of time, monolith application classes have outgrown their size defying the Single Responsibility Principle (SRP). So, there is a need to consider the code within the classes when identifying microservices. Current studies also lack the partitioning of database and ignore the mapping of Database Entities (DE) to the microservices. In this paper, we present a Comprehensive Microservice Extraction Approach (CMEA) that considers: 1) Both classes and their methods to define and refine microservices, 2) Associate the DE to microservices using newly devised eight guiding rules handling ownership conflicts. This approach has been applied to three benchmark web applications implemented in Java and one in-house application implemented in both Java and Python. Our results demonstrate better or similar software quality attributes in comparison to the existing related studies. CMEA improves software quality attributes by 22%. System architects can easily identify microservices along with their DE using our approach. The CMEA is generic and language-independent so it can be used for any application.*

Keywords: *Microservices, static code analysis, microservice identification, software refactoring, database partitioning, move method refactoring.*

Received June 27, 2022; accepted July 17, 2023
https://doi.org/10.34028/iajit/21/1/3

1. Introduction

Microservices are gaining recognition in the software industry. Microservices Architecture allows a legacy enterprise software system to be split into many fine-grained microservices that are developed and deployed independently [14]. Generally, these services are developed using different technological stacks by separate teams [12]. Thus, many software business players like Amazon, Netflix, Uber, Spotify, and numerous small to medium-sized enterprises are opting for microservices-based solutions for their complex monolithic systems [4].

According to Chris Richardson's Scale Cube, a three-dimension scalability model [21], scaling in Microservice Architecture (MSA) corresponds to Y-axis scaling. It suggests the decomposition of a software application into multiple, distinct autonomous services. Each service is accountable for one or more closely-related functions. However, finding these optimized sets of services is intellectually hard and takes time to implement as well [28]. Software architects who re-modularize their monolithic applications to get microservices without clearly understanding their pros and cons invite risk and unforeseen problems [31]. Out of all the set of activities needed to achieve a microservice-based software solution, the most important and determining key activity is the identification of the architectural components that qualify as microservices [7].

During the development of microservices, brownfield development implies the development or improvement of an existing software system [3]. This development strategy utilizes existing system artifacts like static code files, code revision history, or application/web access logs to identify services. Generally, existing microservice extraction approaches build their migration strategy by static analysis of code with an underlying assumption that "classes with strong relationships should be in the same service". However, the challenge is that some classes might be tangled into more than one microservice. These shared classes are called crossovers and additional hard steps need to be taken for their mapping into microservices [15].

Additionally, as monolithic application programs grow and become complex, methods are added in an unstructured manner and eventually, some classes have more responsibilities than envisioned earlier [25]. This leads to overall architecture model drift and erosion of the monolithic software system. This imbalance at the class-level motivates to break a large class into smaller classes due to reasons, such like,

1. A class has grown too large and is taking too much responsibility. To achieve the Single Responsibility Principle (SRP), the bloated class can be further divided into subclasses.
2. Class design lacks a clear separation of concerns.
3. A long-lived monolithic codebase that has passed multiple iterations of change may have a high code toxicity level and poor design of the class code. For example, a method is used more in another class than in its own class.
4. Excessively long methods need to be refactored into a separate class.

In the above scenarios, there is a need to split the class into smaller, more cohesive classes or reorganize the methods of a class while extracting microservices. Kumar *et al.* [18] suggest moving a method to the class that uses it the most. This Move Method Refactoring (MMR) makes classes more cohesive internally and also eliminates dependency between classes [28]. Our research is motivated to find methods that are in need of refactoring.

In our work, we focus on both class-level and method-level dependency analysis between software entities to segregate a cohesive set of classes and methods that form a reasonable set of bounded contexts as microservices. Such mapping analysis may decisively identify a few methods within a class that may be required to be pulled-out from their native classes. Fundamentally, this reorganization will generate clear context and minimize tight coupling among microservices. The refactoring of classes shall help to maintain the SRP for microservices.

Typically, for performing business transactions, the identified granular microservices interact with each other and in most cases, are implemented by joining the data residing on different microservices. For complex queries, the communication overhead and latency may increase too much. There is a need for an optimized database design model that ensures minimum inter-service communication. It is worth noting that existing decomposition techniques available in literature largely ignore the DE aspect that is behind business functions. In fact, in some approaches, a single shared database is maintained behind all the microservices [30]. In such a design, any database outage would adversely affect multiple services and lead to substantial downtime, defeating the purpose of MSA. A careful partition of DE across microservices enables modification to database content without impacting other microservices [6].

This paper proposes a Comprehensive Microservice Extraction Approach (CMEA) based on

1) Both classes and methods.
2) Access rights to DE. For a monolith application, class-level call list and method-level call list is collected using static code analysis.

This data is filtered and further summarized to obtain class and method-level mapping patterns. A grouping technique is applied to this data with the aim to package close-knit classes along with methods as microservices. Based on the frequency of calls, methods are also refactored to their appropriate classes resulting in more internal cohesion. Guiding rules to determine ownership of business entities have been formulated so that extracted microservices have share-nothing [23] or share-as-little-as-possible database dependency on other microservices. Significant contributions of this research paper are as follows:

- Contribution 1: a five-step Comprehensive microservice extraction approach by grouping business classes and methods that are highly cohesive and loosely coupled at the same time.
- Contribution 2: eight guiding rules for decomposing Database Entities (DE) and resolving their ownership conflicts with microservices.
- Contribution 3: applying the proposed approach to three Java benchmark applications,

  1) JPetStore.
  2) AcmeAir.
  3) Cargo tracking system.

- Contribution 4: applying the proposed approach to an in-house ‘Teachers-Feedback Web Application (TFWA)’ written in both Java and Python as a Proof of Concept (POC).
- Contribution 5: validating our results qualitatively and quantitatively.

The remainder of the paper is organized as follows: Section 2 presents the related work going on in this domain. Section 3 describes the methodology for microservice identification. Section 4 walks through the approach on a sample benchmark Java application taken as an example. Section 5 shows the qualitative and quantitative analysis of our methodology along with the results. Section 6 discusses threats to the methodology and section 7 states the conclusion and future work.

2. Related Work

Source code is the only artifact that is essentially available for an application. It reflects the real functionality implemented in the application. Other artifacts might get obsolete or eroded with time, such as documentation. For this reason, several researchers have presented their approach for microservice identification using codebase analysis.

Mazlami *et al.* [23] presented a static algorithmic extraction technique based on three formal coupling strategies- logical coupling, semantic coupling, and contributor coupling. Based on the coupling, a graph is constructed which is decomposed into microservice candidates based on the minimum spanning tree algorithm. An automatic extraction approach is proposed by Eski and Buzluca [11] using code repositories. The authors use code coupling and evolutionary coupling to find microservices. Both Mazlami *et al*. [23] and Eski and Buzluca [11] take into account code revision history for microservice candidate identification. Consequently, if revision history is not updated or is unavailable, their approach becomes unusable. Selmadji *et al.* [26] propose a semi-automatic extraction approach for object-oriented monolithic applications. Their microservices identification approach is based on the structure and behaviour of the application.

Gysel *et al.* [13] suggested a structured approach for the identification of microservices using artifacts and documents based on software engineering principles. It results in a graph representation that is dissected using graph-cutting algorithms. The limitation of this approach is that if documents and artifacts are unavailable or are not updated then the approach is not viable. Jin *et al.* [16] use a dynamic analysis of legacy application logs to find microservice candidates. While it provides decent results, it is greatly dependent on a sufficient pool of test cases to properly execute and cover the whole system. Al-Debagy and Martinek [1] proposed a novel decomposition method by utilizing code to understand similarity within the classes and cluster semantically similar classes together using a neural network model, code2vec. Lohnertz and Oprescu [20] devised an approach to automatically find microservice candidates by utilizing three coupling criteria: static, semantic, and evolutionary. A combined weighted graph is created by aggregating these coupling criteria. Lastly, clustering is applied to isolate microservice recommendations. Raj and Bhukya [24] suggested a fully automated approach for migrating Service Oriented Architecture (SOA) based applications to microservices in three steps. First, a Service Graph (SG) is constructed for the SOA application. Later, for each service, task graphs are built. A microservices extraction algorithm using the SG is applied to generate the candidate microservices. El Kholy and El Fatatry [10] advised the "Managing Database for Microservice Architecture" (MDMA) approach for organizing databases in MSA. But their approach is superficial as it does not mention the functional decomposition of microservices.

All the above-stated approaches may not work suitably well when classes are large and bloated and must be broken into smaller classes thereby moving groups of methods into classes that use it the most. This idea of MMR automatically helps to reduce coupling, bad-smells and increase cohesion [29].

Though few approaches have been suggested for dealing with the concern of microservice identification from code, none of them describe measures for partitioning the database. To a certain extent, many existing approaches keep the data stored in one monolithic database and all services interact with this database. We do not appreciate this approach to database design because services by their very definition must be loosely coupled so that they can be independently deployed and scaled. Also, if database partitioning is not done properly, microservices will keep on connecting to the private DE of other services leading to high coupling and interdependence between them. This shared persistence scenario is a technical anti-pattern in microservice identification [27].

Here, we present guiding criteria that shall help the service architects and developers in deciding the alignment of DE with microservices. We understand that microservice identification achieved using method-level dependency analysis in addition to class-level analysis will help to find methods that are in need of refactoring. Thus, our approach will potentially enhance the granularity and preciseness of the code rearrangement and reorganization. To the best of our knowledge, there is no well-established research that comprehensively identifies microservices based on classes, methods, and DE around them so as to achieve optimized database design models for microservices.

3. Methodology for Microservice Extraction

In this section, we will briefly about the proposed methodology for the extraction of services that are cohesive and based on the SRP [3].

3.1. Basic Definitions

We list a few symbolic notations needed to comprehend the proposed methodology as shown in Table 1.

DE is a set of objects or items around which the data is captured and stored in the form of tables or collections. A database entity has a set of attributes and is related to other DEs. We define it as DE=*{DE_1, DE_2,. . .,DE_k}*.

Table 1. Symbolic notations used for CMEA.

Symbol	Description
M	Microservice.
$\sum \mu$: { μ_1, μ_2,…, μ_n}	Set of microservices.
C_i : i=1...m	Set of class.
M_j : j=1...n	Set of methods.
CL: <c_1,c_2,c_3,…,c_n>	Sequence of class calls.
ML: <m_1,m_2,m_3,…, m_n>	Sequence of method calls.
DE={DE_1, DE_2,. . ., DE_k}	Database entities.

Let μ be a microservice and $\sum\mu$:{ μ_1, μ_2, …, μ_n} is the set of microservices in the monolithic application. Each microservice is characterized by a set of classes and DE like, [μ_n:{(C_i, DE_j), *i*=1...*n*, *j=1*...*m*}]. Each microservice

must satisfy *non-intersection* characteristics i.e., {$\forall \mu_1$ and μ_2, $\mu_1 \cap \mu_2 = (\emptyset)$}. It implies that the microservices have nothing in common in terms of classes and DE. Conversely, it also suggests that the union of two microservices $\mu_1 \cup \mu_2$ is equivalent to merging them.

Microservice Access Control Matrix (MACM) describes the access rights of microservices (represented in rows) over DE (represented in columns) as shown in Figure 1. Consider a set of microservices as $\mu=\{\mu_1, \mu_2 \ldots, \mu_n\}$ and $DE=\{DE_1, DE_2, \ldots, DE_m\}$ as the set of DE, MACM *A* is a *nxm* matrix, where each element *A*(i, j) shows the database operations that a microservice performs on a database entity.

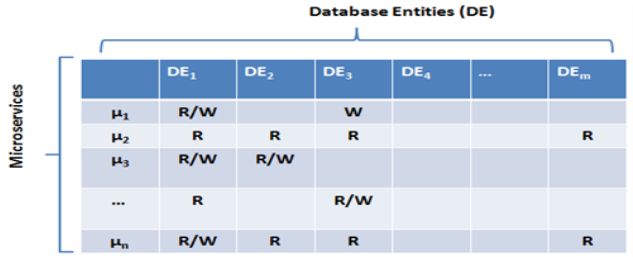


Figure 1. Microservice access control matrix.

3.2. Proposed Approach-CMEA

Extraction of microservices from a monolithic is performed incrementally. Our proposed approach has five steps and it works on brownfield applications where system designers and architects use the application's codebase repositories to re-modularize the application into microservices architectural pattern. A complete outline of the proposed systematic approach is shown in Figure 2. The steps to be performed are discussed in the following subsections.

1. Perform Static Analysis of Code: our approach employs static analysis of the code files. We analyze both class-level and method-level dependency patterns in the monolithic application to extract core business classes and methods that should be grouped together in a microservice. To get this, we generate both:

   1) Class caller-callee call list (CL)
   2) Method caller-callee call list (ML).

CL is basically a sequence of class-to-class calls and ML is a method to method calls that correspond to the calling behaviour in the given monolithic application. The syntax of the CL is

C: class1 class2

Each entry represents that some method(s) in class1 is called some method(s) in class2. Similarly, the syntax of the ML is

M:$class_i$*:* <$method_j$> *(arg_types)* $class_k$*:* <$method_l$> *(arg_types).*

Each entry represents $method_j$ of $class_i$ is calling a $method_l$ of $class_k$.

2. Filter Class Call List and Method Call List: the CL obtained in the previous step can be completely exhaustive for bigger projects. It includes extra classes that will definitely distract from the actual analysis and utilize an inordinate amount of time and effort as well. Thus, there is a need to filter out classes that are not adding any significance to business value. This filtering step uses heuristics to extract classes that are relevant to the core business functionality of the application. In this step, we will filter CL by removing abstract classes, initialization classes, libraries, utility classes, and other classes. Remove self-calling between classes also. Similarly, ML is also filtered to mine useful methods needed for core business functionality. Examples of such filtered methods are: wrappers, getters, setters, library functions, sample data generator functions, *init* methods, and exception handling functions.
3. Service Identification by Generating a Class and Method Dependency Graph: create a class dependency graph G=(*V, E, w*) by linking web interface classes to internal classes. The weights *w* of the edges represents the frequency of call between classes. Internal classes called by multiple interface classes will belong to the group where *w* is higher. Internal classes called by multiple interface classes having the same *w* can be further investigated based on domain knowledge. Now, the classes with strong dependencies are grouped together to generate candidate microservices. Repeat these steps for Method Call List (ML) as well. Albeit, few methods might exist that access other classes more frequently than where they are actually defined. Such methods must be investigated further for possible refactoring. In this scenario, our methodology suggests MMR to redefine them to other classes where they are mostly called [30]. In case, a method is called by another class with the same frequency as that to its own class where the method is defined, then methods can be allowed to remain in the original defining class. We understand if methods are bundled with the microservice where they are frequently used rather than where they are originally defined, inter-service communication between them will be greatly reduced.
4. Creation of MACM: now we will look for the issues of data entity ownerships. For the identified microservices in step 3, we need to determine the ownership of the databases. We create a MACM to determine relationships between the identified microservices and DE. MACM is populated to represent Read/Write access privileges of microservices to the DE.
5. Determining Ownership of DE: the MACM is further analysed to address the splitting or sharing of databases among microservices. Our proposed approach follows the Database-per-service pattern wherein DE pertaining to a microservice are

encapsulated along with its code. In essence, a microservice should privately the own data it needs and other microservices are allowed to use this data via APIs. To partition data between microservices, we present a systematic approach for deciding the ownership of DE between microservices.

This step is an integral component of the whole approach. If not done correctly, microservices will communicate excessively with each other and will become too chatty. As a result, careful consideration must be given to who will "own" the database entity. Here, we define eight guiding criteria that can be used in determining ownership of a database entity as shown in Figure 3.

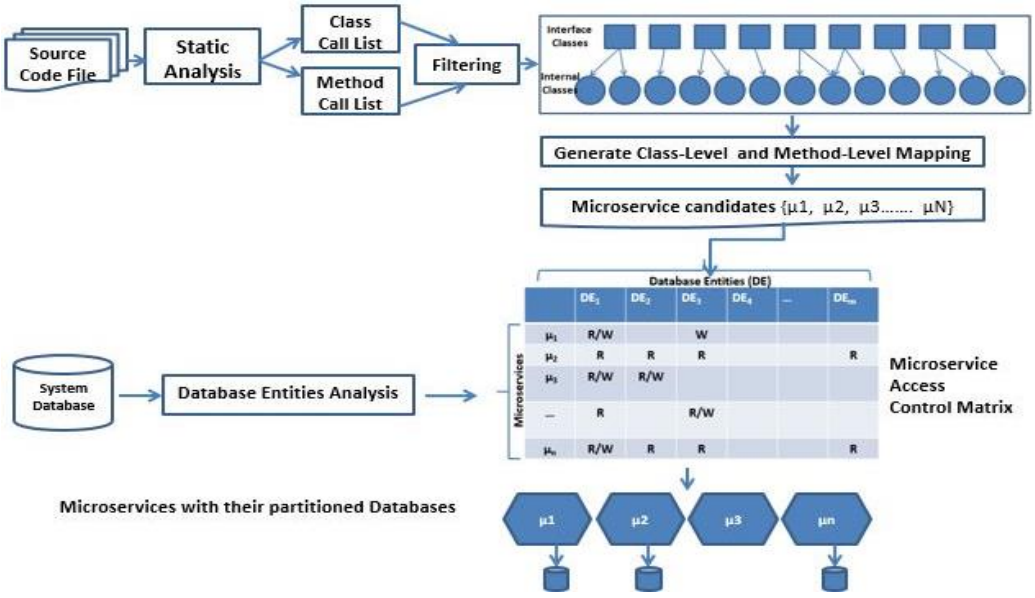


Figure 2. Outline of the proposed approach.

The Guiding Criteria to define ownership of DE is as follows:

- Guiding Criteria 1 (GC1): let μ_1 be the only microservice accessing a database entity DE_1, then μ_1 be assigned the owner of DE_1.
- Guiding Criteria 2 (GC2): let μ_2, μ_3, μ_5, and μ_N be four microservices accessing database entity DE_2 i.e., μ_2, μ_5, and μ_N are reading DE_2 and μ_3 is both reading and writing to DE_2, then μ_3 should be considered the owner of DE_2. The idea is to assign ownership of *DE* to the microservice that writes data to it.
- Guiding Criteria 3 (GC3): let μ_1, μ_2, μ_5, and μ_N are accessing database entity DE_3, i.e., μ_1 is writing to DE_3, μ_5 is reading and writing to DE_3, and μ_2 *and* μ_N is just reading from DE_3. Microservice that interacts more with DE (more read and write operations) shall be considered its owner.
- Guiding Criteria 4 (GC4): let μ_2 and μ_N read some same DE like DE_2, DE_3, and DE_N, then their functionalities may be merged together as none of them is the owner of any database entity.
- Guiding Criteria 5 (GC5): let a database entity DE_I that is accessed by all or the majority of microservices then it indicates a poor database design. In such a scenario, design optimization or re-modeling of the database entity is suggested. In Figure 3, DE_4 is an entity where GC5 can be applied. The DE_4 column is filled with read or write permissions for all microservices, so it should be re-modeled.
- Guiding Criteria 6 (GC6): let μ_N is a microservice that accesses all or majority of the DE $DE = \{DE_1, DE2, \ldots, DE_n\}$, so it can be named as *Super_Microservice*. Such scenarios can be easily identified by finding rows filled with read/write permissions for all DE in MACM e.g., μ_5. These Super_Microservice might be too complex, prone to defects and violate SRP. In this situation, a design optimization or re-factoring of microservice is suggested.
- Guiding Criteria 7 (GC7): let μ_3 and μ_4 are two microservices writing to a database entity DE_5 leading to a conflict in deciding the ownership of DE_5. This conflicting scenario may be investigated in detail by capturing additional data access operations i.e., *CREATE (C)* and *UPDATE (U)* for both conflicting microservices [22]. Technically, *CREATE* operation writes values to all the mandatory fields of a record in a database table and *UPDATE* is updating/ writing in just a subset of fields. While designating weights to C and U operations in the CRUD matrix, C always carries more weight than U. Intuitively, a microservice that creates a database entity has a higher claim of ownership than a microservice updating it. Thus, understanding these additional *C* or *U* operations on a database entity can be used to resolve data ownership conflicts between two or more microservices.

- Guiding Criteria 8 (GC8): let two microservices μ_i and μ_j write to a database entity DE_N and both are performing the same data access operations including *C* and *U* also. This scenario cannot be resolved using GC7. For these scenarios, gaining a deeper understanding from the software architects is significant. They can thoroughly analyze the frequency of Create and Update operations in conflicting microservices. We assert that a microservice having a greater number of write operations to a database entity has more claims for it and should be given its ownership.

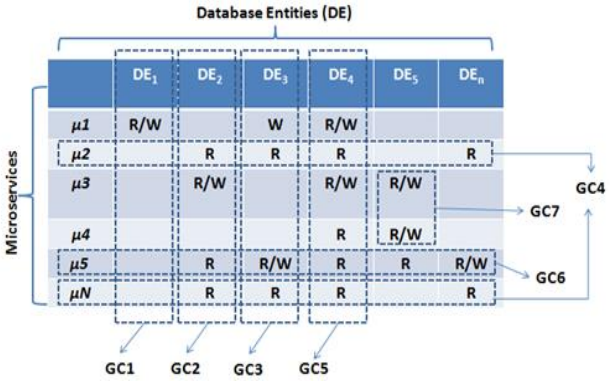


Figure 3. GC to define ownership of DE.

In general, these guiding criteria shall help software practitioners to comprehend and identify data ownership for microservices.

4. Implementation

In this section, we show the application of CMEA on three sample benchmark Java applications and on an in-house Java application: TFWA. A detailed run-through of the proposed approach is depicted on the JPetStore application (monolithic version).

4.1. Applying CMEA to Benchmark Applications

In order to validate our proposed decomposition approach, we carry out a case study on sample open-source monolithic Java applications namely-JPetStore[1], AcmeAir[2], and cargo tracking system[3]. For these web applications, their server-side packaged code is available on Github as Web Application Archive (WAR) or Enterprise Application Archive (EAR) files in the SRC folder.

4.1.1. Description of Case Studies

A brief description of these applications is exhibited in Table 2. Since refactoring a monolithic application into microservices at the enterprise level is a complex and lengthy task; we have opted for these applications as they are of manageable size. Another reason for their selection is that these applications are predominantly used in other related studies as well.

Table 2. Description of sample legacy monolithic application.

Sample benchmark application	Description	LOC	No. of packages	No. of classes	No. of methods
JPetStore	e-store for buying pets	2059	5	24	299
AcmeAir	Flight Reservation	3471	8	33	196
Cargo tracking system	Tracking system for shipping cargo	8635	24	64	525

4.1.2. Proposed Methodology in Practice

Below are the steps we followed to implement the proposed approach on the selected benchmark applications.

We used Java Call Graph utilities[4] for generating static call lists. "*javacg-static*" program reads classes from application's *jar* file, moves down to the method body, and prints a list of 'caller-callee relationships' for both classes and methods. Snippets of the output produced for both CL and ML by this utility for the JPetStore application is shown in Figure 4-a) and (b).

Next, we filter both CL and ML for our sample applications: JPetStore, AcmeAir, Cargo Tracking System, and TFWA. This step achieves a considerable reduction in the number of entries in both CL and ML as shown in Table 3. The detailed CL and ML can be found in our repository[5].

Table 3. Percentage of reduction in CL entries.

Sample benchmark application	# CL	# ML	# Filtered CL	# Filtered ML	Reduction % in CL	Reduction % in ML
JPetStore	173	352	53	133	69.36	62.21
AcmeAir	513	1529	82	193	84.01	87.37
Cargo tracking system	1669	4356	204	345	88.88	92.07
TFWA	1303	2733	210	276	83.89	89.90

Our next task is to group classes and methods that may be bundled together as microservices. Figure 4-c) shows the mapping results of classes for JPetStore application.

For JPetStore application, four groups corresponding to microservices - *Catalog*, *Cart*, *Order*, and *Account* was achieved as shown in Figure 4-d).

For AcmeAir, we achieve four functionally autonomous services: Flight, Booking, Authentication, and Customer. For the cargo tracking system, our approach achieved four microservices: CargoBooking, Handling, Location, and Voyage and Planning. Tables 4 and 5 show the identified microservices and composed classes for AcmeAir and cargo tracking system.

[1]https://github.com/mybatis/jpetstore-6
[2]https://github.com/acmeair/acmeair
[3]https://github.com/citerus/dddsample-core
[4]https://github.com/gousiosg/java-callgraph
[5]https://github.com/anitagoel/CMEA

4.1.3. Examine Ownership of Database Entities

Determining ownership of DE is a critical step in microservice identification. Based on the proposed eight guidelines principles discussed earlier in section 3.2, we have handled this issue.

In JPetStore, we identify thirteen entities i.e., SupplierId, SignOn, Account, Profile, BannerData, Order, OrderStatus, LineItem, Category, Product, Item, Inventory, and Sequence. MACM is shown in Appendix A, Table A.1. Based on guiding criteria, we comprehend that the Account service may own the Account, Profile, SignOn, and BannerData tables. Order service can take ownership of Order, OrderStatus, LineItem, and Sequence entities. Catalog service can take ownership of Item, Category, Product, Inventory, and SupplierID entities.

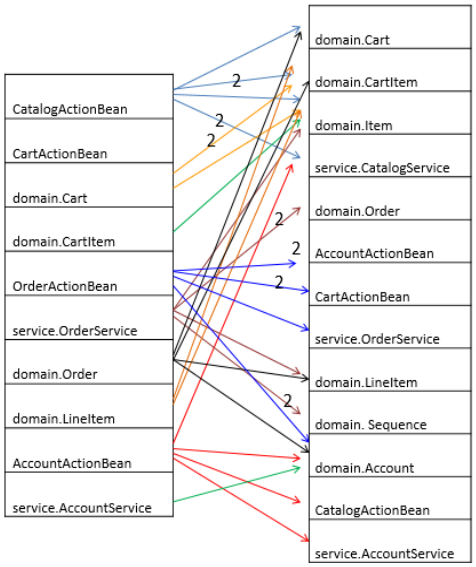
AcmeAir application contains six MongoDB collections-Booking, CustomerSession, Flight, Customer, FlightSegment, and AirportCodeMapping. Applying guiding criteria, we found that Flight service owns Flight, FlightSegment, and AirportCodeMapping collections. Likewise, Booking service possesses Booking collection and Customer service will be the owner of the Customer collection and the Authentication service will become the owner of CustomerSession collection as shown in Appendix A, Table A.2.

Type	Type	Caller Class (A)	Called Class (B)
ClassDependency	C:	org.mybatis.jpetstore.domain.Category	org.mybatis.jpetstore.domain.Category
ClassDependency	C:	org.mybatis.jpetstore.domain.Category	java.lang.Object
ClassDependency	C:	org.mybatis.jpetstore.domain.Category	java.io.Serializable
ClassDependency	C:	org.mybatis.jpetstore.domain.Category	java.lang.String
ClassDependency	C:	org.mybatis.jpetstore.mapper.CategoryMapper	org.mybatis.jpetstore.mapper.CategoryMapper
ClassDependency	C:	org.mybatis.jpetstore.mapper.CategoryMapper	java.lang.Object
ClassDependency	C:	org.mybatis.jpetstore.web.actions.AbstractActionBean	net.sourceforge.stripes.action.SimpleMessage
ClassDependency	C:	org.mybatis.jpetstore.web.actions.AbstractActionBean	java.lang.Object
ClassDependency	C:	org.mybatis.jpetstore.web.actions.AbstractActionBean	org.mybatis.jpetstore.web.actions.AbstractActionBean
ClassDependency	C:	org.mybatis.jpetstore.web.actions.AbstractActionBean	net.sourceforge.stripes.action.ActionBean
ClassDependency	C:	org.mybatis.jpetstore.web.actions.AbstractActionBean	java.io.Serializable
ClassDependency	C:	org.mybatis.jpetstore.web.actions.AbstractActionBean	net.sourceforge.stripes.action.ActionBeanContext
ClassDependency	C:	org.mybatis.jpetstore.web.actions.AbstractActionBean	java.util.List
ClassDependency	C:	org.mybatis.jpetstore.domain.Item	java.lang.StringBuilder
ClassDependency	C:	org.mybatis.jpetstore.domain.Item	org.mybatis.jpetstore.domain.Item
ClassDependency	C:	org.mybatis.jpetstore.domain.Item	java.lang.Object
ClassDependency	C:	org.mybatis.jpetstore.domain.Item	java.io.Serializable
ClassDependency	C:	org.mybatis.jpetstore.domain.Item	java.lang.String
ClassDependency	C:	org.mybatis.jpetstore.web.actions.CartActionBean	org.mybatis.jpetstore.domain.Cart
ClassDependency	C:	org.mybatis.jpetstore.web.actions.CartActionBean	net.sourceforge.stripes.action.ForwardResolution
ClassDependency	C:	org.mybatis.jpetstore.web.actions.CartActionBean	org.mybatis.jpetstore.web.actions.CartActionBean
ClassDependency	C:	org.mybatis.jpetstore.web.actions.CartActionBean	org.mybatis.jpetstore.web.actions.AbstractActionBean
ClassDependency	C:	org.mybatis.jpetstore.web.actions.CartActionBean	org.mybatis.jpetstore.domain.CartItem

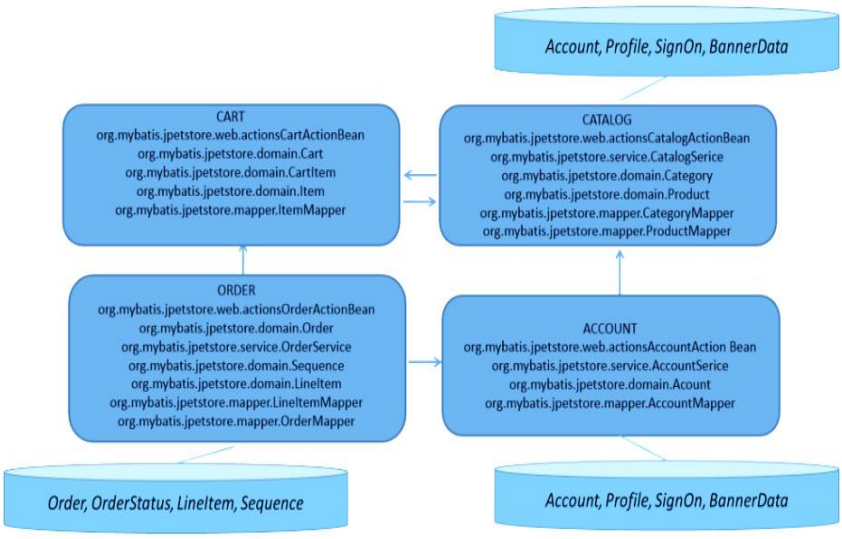
a) Snippet of class call list.

Type	Type	CallingClass	CallingMethod	CalledClass	CalledMethod
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	addItemToCart()	(M)org.mybatis.jpetstore.domain.Cart:	containsItemId(java.lang.String)
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	addItemToCart()	(M)org.mybatis.jpetstore.domain.Cart:	incrementQuantityByItemId(java.lang.String)
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	addItemToCart()	(M)org.mybatis.jpetstore.service.CatalogService:	isItemInStock(java.lang.String)
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	addItemToCart()	(M)org.mybatis.jpetstore.service.CatalogService:	getItem(java.lang.String)
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	addItemToCart()	(M)org.mybatis.jpetstore.domain.Cart:	addItem(org.mybatis.jpetstore.domain.Item,
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	removeItemFromCart()	(M)org.mybatis.jpetstore.domain.Cart:	removeItemById(java.lang.String)
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	removeItemFromCart()	(M)org.mybatis.jpetstore.web.actions.CartActionBean:	setMessage(java.lang.String)
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	updateCartQuantities()	(M)org.mybatis.jpetstore.web.actions.CartActionBean:	getCart()
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	updateCartQuantities()	(M)org.mybatis.jpetstore.domain.Cart:	getAllCartItems()
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	updateCartQuantities()	(I)java.util.Iterator:	hasNext()
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	updateCartQuantities()	(I)java.util.Iterator:	next()
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	updateCartQuantities()	(M)org.mybatis.jpetstore.domain.CartItem:	getItem()
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	updateCartQuantities()	(M)org.mybatis.jpetstore.domain.Item:	getItemId()
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	updateCartQuantities()	(I)javax.servlet.http.HttpServletRequest:	getParameter(java.lang.String)
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	updateCartQuantities()	(M)org.mybatis.jpetstore.web.actions.CartActionBean:	getCart()
MethodCall	M:	org.mybatis.jpetstore.web.actions.CartActionBean	updateCartQuantities()	(M)org.mybatis.jpetstore.domain.Cart:	setQuantityByItemId(java.lang.String,int)

b) Snippet of ML.



c) Two level mapping between classes.



d) Identified microservices along with composed classes.

Figure 4. Run through JPetStore application.

Table 4. Identified microservices and composed classes for AcmeAir.

S.No.	Service name	Composed classes
1	Booking	BookingREST, BookingService, BookingServiceImpl, BookingLoader
2	Customer	CustomerREST, ServiceLocator, CustomerService, CustomerServiceImpl, CustomerLoader, CustomerInfo, AddressInfo, RestCookieSessionFilter
3	Flight	FlightREST, FlightService, FlightLoader, FlightServiceImpl, AirportCodeMapping
4	Authentication	LoginREST, AuthService, SessionLoader, AuthServiceImpl, KeyGenerator

Table 5. Identified microservices and composed classes for cargo tracking system.

S.No.	Service Name	Composed Classes
1	Cargo Booking	BookingServiceImpl, Cargo, CargoRepository, Delivery, RoutingStatus, Itinerary, Leg, RouteSpecification,TrackingId
2	Handling	HandlingActivity, HandingEvent, HandlingEventRepository, Handling History
3	Location	Location, LocationRepository, SampleLocation, UnLoCode
4	Voyage and Planning	CarrierMovement, SampleVoyage, Schedule, Voyage, VoyageNumber, VoyageRepository

For the cargo tracking system, we identify six entities, Cargo, Leg, Location, HandlingEvent, Voyage, and CarrierMovement table and three components RouteSpecification, Itinerary, and Delivery associated with cargo.hbm.xml. CargoBooking service writes to the Cargo and RouteSpecification table. So, applying GC2, Cargo and RouteSpecification table ownership can be given to CargoBooking microservice as shown in Appendix A, Table A.3. Only the Location service writes to the Location table. It suggests that Location can take ownership of this table (GC2). Likewise, the HandlingEvent entity is owned by the Handling service. Similarly, Voyage, and CarrierMovement entities are owned by VoyagePlanning service.

4.2. Applying CMEA on in-House Application

In this section we will discuss the application of CMEA in a case study based on the TFWA as a POC. TFWA automates the teachers' feedback mechanism in a university system. This in-house application is implemented in both Java and Python

4.2.1. Description of Case Study

TFWA is used by students to give their feedback to all the respective teachers and subjects who teach that subject. To confirm the complete participation of all students in this feedback process, the Teacher-Coordinator (TC) of every department can check the feedback status. TC can analyze the departmental feedback data from different analysis views. The principal has access to analyze college feedback data from different analytics perspectives [2].

Java Implementation of TFWA is a Spring Boot monolithic application having Lines of code: 2504, Number of packages: 10, Number of classes: 30, and Number of methods: 128.

Python Implementation of TFWA is designed on the Model View Template (MVT) architectural pattern and is developed using Django framework 3.0.2 that is a free and open-source web framework. In our implementation we have used SQLite as a database backend, a default option supported by Django.

4.2.2. Ownership of Database Entities

Feedback microservice writes to Feedback table and reads QTemplate table as shown in Appendix A in Table A.4. No other microservice performs any write operation on the Feedback table and reads the Qtemplate table. Applying guiding criteria 1 and 2, Feedback and QTemplate ownership are given to Feedback microservice. Only Authentication microservice writes to tables like UserRole and UserDetails. It suggests that Authentication takes ownership of these tables. Analytics read Department, Course, Paper and Feedback tables. So, the ownership of the Department, Course, and Paper is given to the Analytics service.

5. Quantitative Evaluation

In this section, we'll brief about the quantitative and quantitative evaluation of the proposed approach on chosen sample benchmark applications. The purpose of this evaluation is to determine whether CMEA creates effective microservice candidates.

5.1. Qualitative Evaluation-From Software Industry Expert

Since the whole decomposition process is quite methodical and subject to errors, we have validated CMEA from two agile software industry experts working in a Dubai-based organization in the domain of transforming enterprise-scale applications to MSA. Both have significant experience in building and designing services from monolithic applications. They modeled the business domain of sample benchmark applications using domain-driven design orientation and agreed to our decomposition proposal for microservice extraction for JPetStore, AcmeAir, and cargo tracking system applications.

5.2. Quantitative Evaluation

For the chosen applications, we fail to find software quality metrics that are used in existing research papers for all three benchmark applications. Researchers have confirmed their results on diverse sets of quality metrics. Hence, we assess these benchmark applications against those quality metrics that are used in other

related studies. This enables us to test and verify our approach on a diverse range of quality metrics.

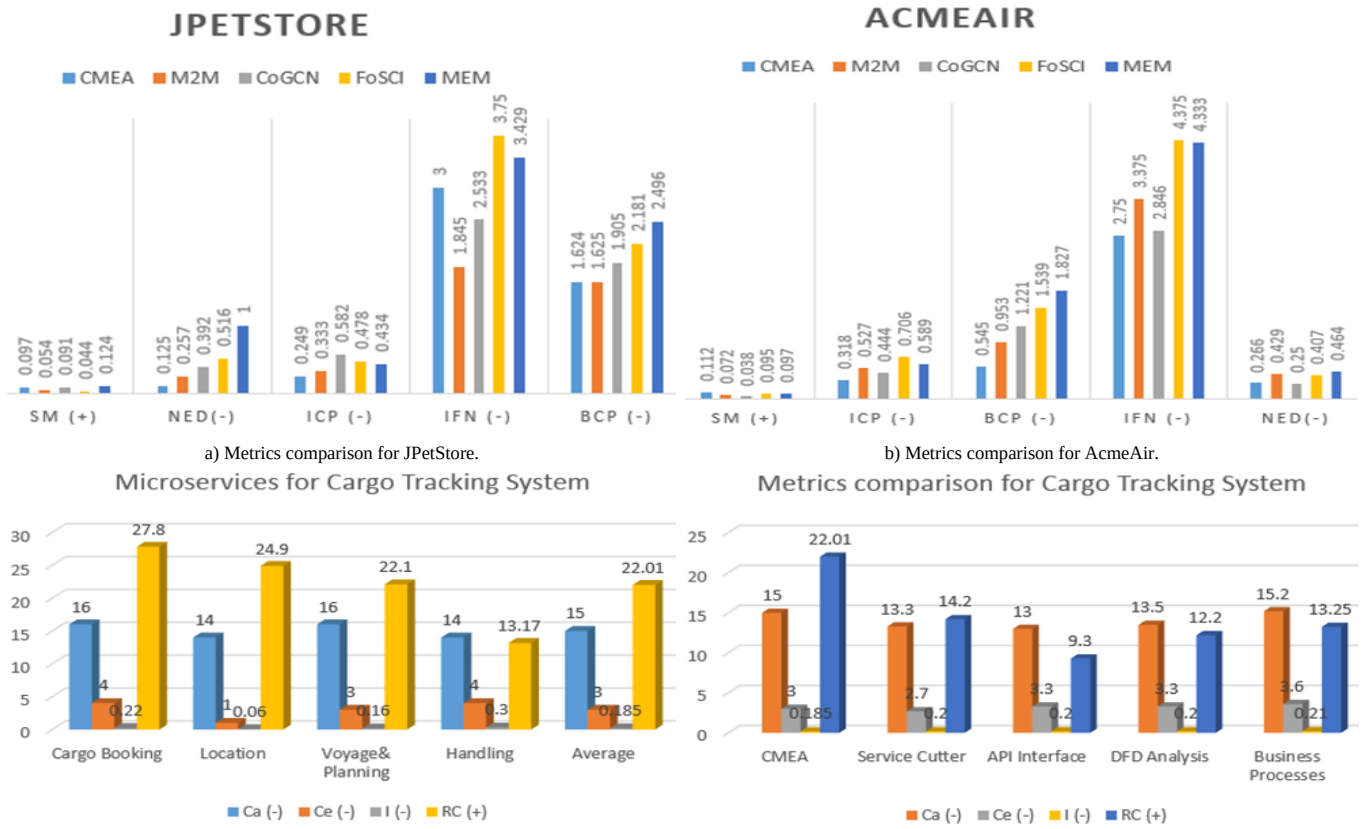
In general, a good microservice decomposition approach should produce services that are loosely coupled and highly cohesive. Cohesion indicates the strength of associations between methods. High cohesion means better reliability, reusability, robustness, and understandability of microservices. Coupling shows interconnections and dependencies among services.

We evaluate CMEA with four established microservice identification techniques-Mono2Micro [17], CoGCN [9], FoSCI [15], and MEM [23] for JPetStore, and AcmeAir application. We apply five quality performance metrics namely-Structural Modularity (SM), Non-Extreme Distribution (NED), Inter-Partition Call percentage (ICP), Interface Number (IFN), and Business Context Purity (BCP) to measure the effectiveness of CMEA. A brief description for these metrics is mentioned in [4].

For cargo tracking system, we were not able to find any research article where the above-mentioned quality evaluation metrics are used. So, we employ another set of four object-oriented metrics namely-Number of Incoming Dependencies (Ca), Number of Outgoing Dependencies (Ca), Instability (I), and Relational Cohesion[6] (RC). These metrics are used and discussed in many reference techniques [5, 8, 13, 29].

The comparison of our results across two sets of metrics are presented in Figure 5. For all the quality assessment metrics, two types of tags are assigned : "(-)" or "(+)". Tag "(-)" shows lower values are better, while Tag "(+)" shows higher values are better.

For JPetStore, CMEA gives superior results for NED, ICP, and BCP. NED specifies that the majority of the identified microservices hold 5 to 20 classes as shown in Figure 5-a). Lower ICP shows reduced calling between microservices. It is noteworthy that SM is slightly lower than MEM (highest) but significantly greater than the other three approaches. IFN is better than FoSCI and MEM but M2M and CoGCN have better values than our approach.



a) Metrics comparison for JPetStore.

b) Metrics comparison for AcmeAir.

c) Quality Metrics for cargo tracking system microservices.

d) Metrics comparison for cargo tracking system.

Figure 5. Quantitative evaluation-JPetStore application.

For the AcmeAir application, CMEA performed better than other techniques for SM, ICP, IFN, and BCP as illustrated in Figure 5-b). CMEA results of SM indicates that partitions have better modular quality. For NED, our approach yields slightly higher than CoGCN, but much better than rest three techniques namely M2M, FoSCI, MEM.

We used SonarGraph-Architect (12.0.4.713 version) [32] to evaluate metrics for cargo tracking system. SonarGraph-Architect a technical quality assessment tool. Figure 5-c) shows values of the quality assessment metrics for microservice generated by CMEA. Our results reveal a better performance in Relational Cohesion and Instability Index. It indicates more

[6]http://eclipse.hello2morrow.com/doc/standalone/content/core_metrics.html

maintainable, robust, and reusable services. Number of Incoming and Outgoing Dependencies metrics gives a bit increased value as compared to other existing techniques [5, 8, 13, 19], as shown in Figure 5-d).

For TFWA, we identified three microservices namely Feedback, Analytics, and Authentication. For Java implementation, we compared the TFWA-monolithic and microservices application (developed according to CMEA). Table 6 shows the software quality metrics[7] (generated using SonarGraph-Architect).

Table 6. Metrics comparison for TFWA (Java implementation).

Metrics	Monolithic TFWA	Microservice TFWA using CMEA
Physical cohesion (+)	1.91	**2.38**
Physical coupling (-)	1.79	**1.32**
System maintainability level (+)	60	**70**
Structural debt index (-)	124	**78**
Cyclic Java packages (-)	8	**6**
Component dependencies to remove (-)	9	**6**

For Python implementation, we again compared the TFWA-monolithic and microservices application (developed according to CMEA). Table 7 shows the software quality metrics[8] (generated using SonarGraph-Architect).

Table 7. Metrics comparison for TFWA (Python implementation).

Metrics	Monolithic TFWA	Microservice TFWA using CMEA
Component dependencies to Remove (-)	60	**40**
Structural debt Index (-)	923	**469**
Number of critical Python Package cycle groups (-)	4	**1**
Average complexity (-)	5.99	**1**

CMEA results show that microservice application yields lesser structural debt index and cyclic package dependencies. At the same time, higher system maintainability level, and physical cohesion. Therefore, our approach improves the maintainability, quality, and long-term health of the application. To summarize, CMEA achieves better cohesion, lesser coupling, a smaller number of operations performed by a service, and lastly, a smaller number of calls between services. Therefore, these results exhibit an acceptable POC.

6. Threats to Validity

We understand that CMEA is a generic approach and can be applied to projects having varied languages, sizes, and architectural structures. We have applied our approach to Java and Python projects but they can be applied to Net applications equally well.

The microservice extraction procedure proposed in this work focus on "splitting design" i.e., defining the functional boundaries for microservices. Accordingly, we perform analysis of the identified microservices using various object-oriented quality metrics of the design stage of the software life cycle.

Another possible threat to this study is the fact no standard quality metrics are defined for the assessment of identified microservice. For Teachers' Feedback application, and cargo tracking system, we relied on the SonarGraph-Architect tool to collect quality assessment attributes. For JPetStore and AcmeAir, we implemented the metrics discussed in a related publication [25] and took clarifications from the publication's authors in case of doubts.

Our approach works on monolithic source code. In our design, the code repositories of the chosen benchmark applications (available at Github) have only one SRC folder, ensuring them to be monolithic applications. Thus, it becomes imperative not to choose projects having multiple project folders (multiple SRC folders) which indicate them to be SOA based application or already microservice application.

Lastly, selected benchmark projects used in this research are open-source projects. We anticipate and predict the consistency of our results for proprietary and enterprise level software as well.

7. Conclusions and Future Work

For microservice extraction, the most challenging task is to identify the right microservice candidates. Few brownfield approaches exist in academic literature to identify microservices using source code repositories. However, these approaches rely on static class-level coupling information and largely neglect 1) method-level refactoring, 2) splitting of bloated classes, and 3) database decomposition. Our research is motivated to fill these gaps existing in the decomposition approaches proposed in this area so far. We make use of MMR which is applied when a method depends more on members of other classes than on its own original class. MMR improves organization, maintainability, reliability, and reusability of code and also results in achieving fine-grained microservices.

Sharing a single database among multiple services as recommended so far is incomplete and could be more error-prone. For MSA, we should split the monolith database such that each microservice totally encapsulates its own data. For this, we have proposed a CMEA, utilizing both business functions and DE to identify microservices. This approach identifies functionally independent microservices by grouping cohesive business classes and methods. For ownership of DE, we have also recommended step-by-step guiding criteria for a microservice. Largely, the results of our technique are positive and outperform other state-of-the-art baseline techniques. Our approach facilitates

[7]http://eclipse.hello2morrow.com/doc/standalone/content/java_metrics.html
[8]http://eclipse.hello2morrow.com/doc/standalone/content/python_metrics.html

system developers and architects in identifying logically cohesive microservices from a legacy application and partitioning DE around these services. In the future, we will apply our approach to large-scale enterprise applications. Also, we aspire to evaluate the database performance of the microservice-based systems implemented using CMEA.

Acknowledgment

We express gratitude to the developers of JPetStore, AcmeAir, and cargo tracking system for providing their applications codebase for this research. We also thank HELLO2MORROW Inc. for providing a trial and evaluation license for SonarGraph-Architect. We also like to express our special thanks of gratitude to the team of system architects from Tribal Scale, Dubai, for the qualitative assessment of our approach.

References

[1] Al-Debagy O. and Martinek P., "Dependencies-Based Microservices Decomposition Method," *International Journal of Computers and Applications*, vol. 44, no. 9, pp. 814-821, 2022. https://doi.org/10.1080/1206212X.2021.1915444

[2] Bajaj D., Bharti U., Goel A., and Gupta S., "Partial Migration for Re-Architecting a Cloud Native Monolithic Application into Microservices and Faas," *in Proceedings of the* 5th *International Conference on Information, Communication and Computing Technology*, New Delhi, pp. 111-124, 2020. https://doi.org/10.1007/978-981-15-9671-1_9

[3] Bajaj D., Bharti U., Goel A., and Gupta S., "A Prescriptive Model for Migration to Microservices Based on SDLC Artifacts," *Journal of Web Engineering*, vol. 20, no. 3, pp. 817-852, 2021. DOI: 10.13052/jwe1540-9589.20312

[4] Bajaj D., Goel A., and Gupta S., "GreenMicro: Identifying Microservices from Use Cases in Greenfield Development," *IEEE Access*, vol. 10, pp. 67008-67018, 2022. DOI:10.1109/ACCESS.2022.3182495

[5] Baresi L., Garriga M., and De Renzis A., "Microservices Identification through Interface Analysis," *in Proceedings of the Service-Oriented and Cloud Computing 6th IFIP WG 2.14 European Conference*, Oslo, pp. 19-33, 2017. https://doi.org/10.1007/978-3-319-67262-5_2

[6] Baškarada S., Nguyen V., and Koronios A., "Architecting Microservices: Practical Opportunities and Challenges," *Journal of Computer Information Systems*, vol. 60, no. 5, pp. 428-436, 2020. https://doi.org/10.1080/08874417.2018.1520056

[7] Cerny T., "Aspect-Oriented Challenges in System Integration with Microservices, SOA and IoT," *Enterprise Information Systems*, vol. 13, no. 4, pp. 467-489, 2019. DOI:10.1080/17517575.2018.1462406

[8] Daoud M., El Mezouari A., Faci N., Benslimane D., Maamar Z., and El Fazziki A., "A Multi-Model Based Microservices Identification Approach," *Journal of Systems Architecture*, vol. 118, pp. 102200, 2021. https://doi.org/10.1016/j.sysarc.2021.102200

[9] Desai U., Bandyopadhyay S., and Tamilselvam S., "Graph Neural Network to Dilute Outliers for Refactoring Monolith Application," *SFU Public Knowledge Project*, vol. 35, no. 1, pp. 72-80, 2021. DOI:10.1609/aaai.v35i1.16079

[10] El Kholy M. and El Fatatry A., "Framework for Interaction between Databases and Microservice Architecture," *IT Professional*, vol. 21, no. 5, pp. 57-63, 2019. DOI:10.1109/MITP.2018.2889268

[11] Eski S. and Buzluca F., "An Automatic Extraction Approach-Transition to Microservices Architecture from Monolithic Application," *in Proceedings of the 19th International Conference on Agile Software Development: Companian*, Porto, pp. 1-6, 2018. https://doi.org/10.1145/3234152.3234195

[12] Ghlala R., Kodia Z., and Ben Said L., "Using MCDM and FaaS in Automating the Eligibility of Business Rules in the Decision-Making Process," *The International Arab Journal of Information Technology*, vol. 20, no. 2, pp. 224-233, 2023. https://doi.org/10.34028/iajit/20/2/9

[13] Gysel M., Kölbener L., Giersche W., and Zimmermann O., "Service Cutter: A Systematic Approach to Service Decomposition," *in Proceedings of the 5th IFIP WG 2.14 European Conference on Service-Oriented and Cloud Computing*, Vienna, pp. 185-200, 2016. https://doi.org/10.1007/978-3-319-44482-6_12

[14] Jamshidi P., Pahl C., Mendonça N., Lewis J., and Tilkov S., "Microservices: The Journey so far and Challenges Ahead," *IEEE Software*, vol. 35, no. 3, pp. 24-35, 2018. DOI:10.1109/MS.2018.2141039

[15] Jin W., Liu T., Zheng Q., Cui D., and Cai Y., "Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering," *in Proceedings of the IEEE International Conference on Web Services*, San Francisco, pp. 211-218, 2018. DOI:10.1109/ICWS.2018.00034

[16] Jin W., Liu T., Cai Y., Kazman R., Mo R., and Zheng Q., "Service Candidate Identification from Monolithic Systems Based on Execution Traces," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 987-1007, 2021. DOI:10.1109/TSE.2019.2910531

[17] Kalia A., Xiao J., Krishna R., Sinha S., Vukovic M., and Banerjee D., "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, pp. 1214-1224, Athens, 2021. https://doi.org/10.1145/3468264.3473915

[18] Kumar L., Satapathy S., and Murthy L., "Method Level Refactoring Prediction on five Open Source Java Projects Using Machine Learning Techniques," *in Proceedings of the 12th Innovations on Software Engineering Conference*, Pune, pp. 1-10, 2019. https://doi.org/10.1145/3299771.3299777

[19] Li S., Zhang H., Jia Z., Li Z., Zhang C., Li J., Gao Q., Ge J., and Shan Z., "A Dataflow-Driven Approach to Identifying Microservices from Monolithic Applications," *Journal of Systems and Software*, vol. 157, pp. 110380, 2019. https://doi.org/10.1016/j.jss.2019.07.008

[20] Lohnertz J. and Oprescu A., "Steinmetz : Toward Automatic Decomposition of Monolithic Software into Microservices," *Seminar Series on Advanced Techniques and Tools for Software Evolution*, vol. 2754, pp. 1-8, 2020. https://ceur-ws.org/Vol-2754/paper2.pdf

[21] Márquez G., Villegas M., and Astudillo H., "A Pattern Language for Scalable Microservices-Based Systems," *in Proceedings of the 12th European Conference on Software Architecture: Companion*, Madrid, pp. 1-7, 2018. https://doi.org/10.1145/3241403.3241429

[22] Matalqa S. and Mustafa S., "The Effect of Horizontal Database Table Partitioning on Query Performance," *The International Arab Juornal of Information Technology*, vol. 13, no. 1A, pp. 184-189, 2016. https://iajit.org/PDF/Vol%2013,%20No.%201A%20(Special%20Issue)/329.pdf

[23] Mazlami G., Cito J., and Leitner P., "Extraction of Microservices from Monolithic Software Architectures," *in Proceedings of the IEEE 24th International Conference on Web Services*, Honolulu, pp. 524-531, 2017. DOI:10.1109/ICWS.2017.61

[24] Raj V. and Bhukya H., "Assessing the Impact of Migration from SOA to Microservices Architecture," *Springer Nature Journal of Computer Science*, vol. 4, pp. 577, 2023. https://doi.org/10.1007/s42979-023-01971-2

[25] Schmidt F., MacDonell S., and Connor A., *Studies in Computational Intelligence*, Springer, 2012. https://doi.org/10.1007/978-3-642-23202-2_7

[26] Selmadji A., Seriai A., Bouziane H., Mahamane R., Zaragoza P., and Dony C., "From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach," *in Proceedings of the IEEE International Conference on Software Architecture*, Salvador, pp. 157-168, 2020. DOI:10.1109/ICSA47634.2020.00023

[27] Taibi D., Lenarduzzi V., and Pahl C., *Microservices: Science and Engineering*, Springer, 2018. https://doi.org/10.1007/978-3-030-31646-4_5

[28] Trabelsi I., Abdellatif M., Abubaker A., Moha N., Mosser S., Ebrahimi-Kahou S., and Guéhéneuc Y., "From Legacy to Microservices: A Type-based Approach for Microservices Identification Using Machine Learning and Semantic Analysis," *Journal of Software: Evolution and Process*, vol. 35. no. 10, pp. 1-27, 2023. https://doi.org/10.1002/smr.2503

[29] Tsantalis N. and Chatzigeorgiou A., "Identification of Move Method Refactoring Opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347-367, 2009. DOI:10.1109/TSE.2009.1

[30] Walker A., Das D., and Cerny T., "Automated Code-Smell Detection in Microservices through Static Analysis: A Case Study," *Applied Sciences*, vol. 10, no. 21, pp. 1-20, 2020. https://doi.org/10.3390/app10217800

[31] Zimmermann O., "Microservices Tenets: Agile Approach to Service Development and Deployment," *Computer Science-Research and Development*, vol. 32, no. 3-4, pp. 301-310, 2017. DOI:10.1007/s00450-016-0337-0

[32] Von Zitzewitz A., "Mitigating Technical and Architectural Debt with Sonargraph," *in Proceedings of the IEEE/ACM International Conference on Technical Debt*, Montreal, pp. 66-67, 2019. DOI:10.1109/TechDebt.2019.00022

Deepali Bajaj is Associate Professor in Department of Computer Science, Shaheed Rajguru College of Applied Sciences for women (University of Delhi). She has over 17 years of teaching experience at university level. She has done her Ph.D. in the area of Cloud and Distributed Computing. Her key research areas are Microservices, Function-as-a-Service (FaaS) and Serverless Technology. She has authored several national and international research publications. She has also authored and edited books in Computer Science.

Anita Goel is Professor in Department of Computer Science, Dyal Singh College, University of Delhi, India. She has a work experience of more than 30 years. She is a visiting faculty to several Universities in India. She has guided several students for their doctoral studies and has travelled internationally to present research papers. Her research interests include Cloud Computing, Microservices, Serverless Computing, Software Engineering, and Technology-Enhanced education (MOOC). She has authored books in Computer Science and has several national and international research publications.

Suresh Gupta is B.Tech. from IIT Delhi. He worked as Deputy Director General, Scientist-G and Head of Training at National Informatics Centre, New Delhi. He has extensive experience in design and development of large Complex Software Systems. Currently he is a Visiting Faculty at Department of Computer Science and Engineering, IIT Delhi. His research interests include Software Engineering, Data Bases and Cloud Computing.

Appendix A

Table A.1. MACM for JPetStore.

μName	**Supplier Id**	**Sign On**	**Account**	**Profile**	**BannerData**	**Order**	**Order status**	**Line item**	**Category**	**Product**	**Item**	**Inventory**	**Sequence**
Cart	R	-	-	-	-	-	-	-	R	R	R	R	-
Catalog	RW	-	-	-	RW	-	-	-	RW	RW	RW	RW	-
Order	-	-	R	-	-	RW	RW	RW	-	-	R	-	RW
Account	-	RW	RW	RW	RW	-	-	-	-	-	-	-	-

Table A.2. MACM for AcmeAir

μName	**Booking**	**Customer session**	**Flight**	**Customer**	**Flight segment**	**AirportCode mapping**
Flight	-	-	RW	-	RW	RW
Booking	RW	-	-	R	R	R
Authentication	-	RW	-	R	-	-
Customer	R	R	R	RW	-	-

Table A.3. MACM for cargo tracking system.

μName	**Cargo**	**Route Specification**	**Itinerary**	**Leg**	**Location**	**Handling event**	**Delivery**	**Voyage**	**Carrier movement**
Cargo booking	RW	RW	R	R	R	-	RW	R	R
Handling	R	-	R	-	-	RW	R	R	R
Location		-	-	-	RW	-	-	-	-
Voyage and planning	R	R	R	R	R	-	-	RW	RW

Table A.4. MACM for TFWA

μName	**Department**	**Course**	**Paper**	**QTemplate**	**Feedback**	**User role**	**User details**
Feedback	-	R	R	R	RW	-	-
Analytics	R	R	R	-	R	-	-
Authentication	-	-	-	-	-	RW	RW