Received 29 March 2022, accepted 23 May 2022, date of publication 13 June 2022, date of current version 28 June 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3182495

GreenMicro: Identifying Microservices From Use Cases in Greenfield Development

DEEPALI BAJAJ[1], ANITA GOEL[2], AND S. C. GUPTA[3]
[1]Department of Computer Science, Shaheed Rajguru College of Applied Sciences for Women, University of Delhi, Delhi 110096, India
[2]Department of Computer Science, Dyal Singh College, University of Delhi, New Delhi 110003, India
[3]Department of Computer Science, Indian Institute of Technology Delhi, New Delhi 110016, India

Corresponding author: Deepali Bajaj (deepali.bajaj@rajguru.du.ac.in)

ABSTRACT Microservices architecture is a new paradigm for developing a software system as a collection of independent services that communicate via lightweight protocols. In greenfield development, identifying the microservices is not a trivial task, as there is no legacy code lying around and no old development to start with. Thus, identification of microservices from requirements becomes an important decision during the analysis and design phase. Use cases play a vital role in the requirements analysis modeling phases in a model-driven software engineering process. Use cases capture the high-level user functions and the scope of system. In this paper, we propose GreenMicro, an automatic microservice identification technique that utilizes the use cases model and the database entities. Both features are the artifacts of analysis and design phase that depict complete functionality of an overall system. In essence, a collection of related use cases indicates a bounded context of the system that can be grouped in a suitable way as microservices. Therefore, our approach GreenMicro clusters close-knit use cases to recover meaningful microservices. We investigate and validate our approach on an in-house proprietary web application and three sample benchmark applications. We have mapped our approach to the state-of-the-art software quality assessment attributes and have presented the results. Preliminary results are motivating and the proposed methodology works as anticipated in identifying functionally cohesive and loosely coupled microservice candidate recommendations. Our approach enables the system architects to identify microservice candidates at an early analysis and design phase of development.

INDEX TERMS Automatic decomposition, architectural restructuring, greenfield development, microservices, migration, use cases.

I. INTRODUCTION

Monolithic architecture is one of the most extensively used architectures for web application. It bundles the user interface, business logic, and the data store into a single executable file that is eventually deployed as a single package. The web server accepts incoming HTTP requests, executes the request and produces a response. But, as the monolithic applications get bigger in size, codebase becomes more complex. Eventually, maintainability and scalability of monolithic application becomes tough and expensive. A small change in the application necessitates testing and redeployment of the entire application. Furthermore, regarding scaling of the monolithic application, an increased traffic involves deploying the entire codebase even though barely a small subset of its component is overloaded.

The associate editor coordinating the review of this manuscript and approving it for publication was Claudia Raibulet.

Recently, Microservice Architecture (MSA) style is growing expeditiously, and many Internet-based organizations like Netflix, Amazon, Google, Twitter, eBay, and Uber are exploiting the digital transformation to restructure their legacy codebase [1]. In MSA, applications are developed as a set of tiny, distinct, highly cohesive, loosely coupled, autonomously scalable, independently deployable, and distributed services [2].

In microservices, the key idea is that each microservice owns its domain representation i.e. data, logic, and behavior. Related functionalities or requirements of the monolithic application are combined into a single business capability called microservice. This decomposition is based on the Single Responsibility Principle (SRP) that suggests “Gather together those functionalities that change for the same reason, and separate those that change for different reasons'' [3]. This way MSA helps to handle complexities of bulky applications by breaking them into tiny services, where each service satisfies its own bounded-context. Another important aspect of MSA is traceability between the functional requirements and microservice system structure. Therefore, only one service has to be scaled, updated and redeployed in case of a change in domain. This initiative enables faster time-to-market and reduces turnaround time for each release for microservice-based architecture. However, the advantages come at the price of complex deployments of individual services, management overhead, and monitoring challenges.

Microservice application development can be categorized as Greenfield or Brownfield [4]. Greenfield development approach refers to the development plan starting from a clean slate i.e. no legacy code around. Brownfield development means development of a new software system from an existing application. Here, we focus on greenfield development. It means development for a new environment from scratch with no restrictions or dependencies. System analysis and design artifacts of SDLC (Software Development Life Cycle) are available for decomposition in such scenarios. Major SDLC artifacts that can be used are use cases, functional requirements, non-functional requirements, DFD, BPMN, API specifications, class diagrams, UML diagrams, Domain Driven Design and application and data design. As there is no clear direction about microservices development, the degree of risk is comparatively higher for greenfield approaches which makes these developments more challenging.

A use case is a methodology used in system analysis to identify, elucidate, and organize system requirements. Use cases are usually written by software analysts and can be used extensively during different stages of SDLC. Use case diagrams abstract high-level view of business functionality. Use case modeling is accepted and widely used in industry. A use case model includes use cases, actors and relationships between use cases [5]. Use cases may be related to other use cases by the relationships: Generalization, Include, and Extend [6], as shown in Figure 1.

Generalization is an inheritance relationship between two use cases such that one use case inherits all the properties and relationships of another use case. *Generalization* between use cases is represented as a solid directed line with a large arrowhead toward the parent use case. In *Generalization*, a parent use case may be a specialized use case with one or more child use cases that indicate more specific forms of the parent. In such scenarios, the child inherits all structure, behavior, and relationships of the parent. Children of the same parent are all specializations of the parent.

Include relationship denotes the inclusion of a use case as a sub-process of another base use case. Here, the base use case is dependent on the included use case and without them the base use case is incomplete as the included use case represents a sub-sequence of interactions that may always happen. In other words, if a certain use case must function at the end of another use case then there will be an Include relationship between the two use cases. Base use cases require the completion of included use cases in order to be completed. Dotted Arrow is directed from base use case to included use case.

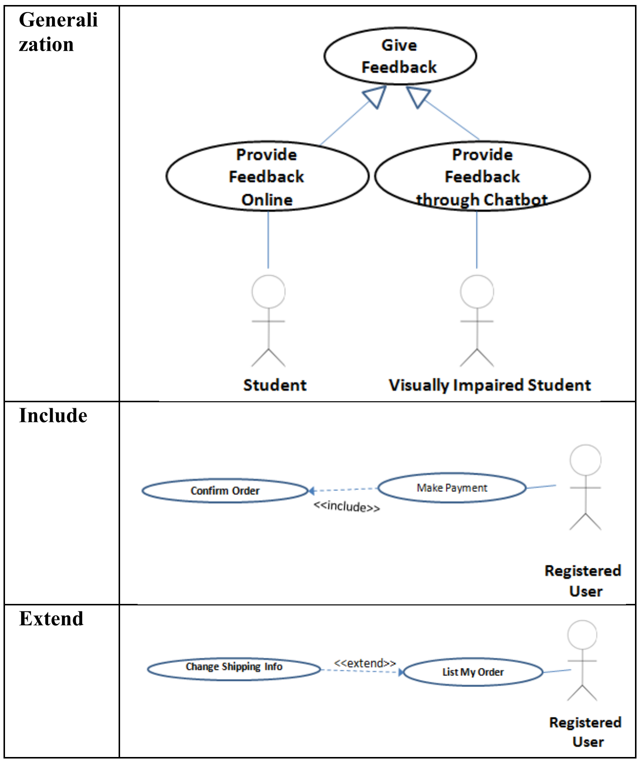


FIGURE 1. **Modeling options in use case relationships.**

The *Extend* use case is dependent on the base use case. It exactly extends the behavior described by the base use case. Base use case should be a fully-functional use case in its own way. Arrow is directed from the extended use case towards the base use case. These three relationships can be used to group use cases and to discover appropriate partitions of the system. Typically, microservices are designed intuitively, utilizing the expertise of the software architects and designers. However, developing wrong service boundaries can prove to be very expensive [7]. It will lead to a higher inter-service communication, and highly coupled components, and consequently it might be worse than just having a single monolithic system.[1] Unfortunately, in this scenario, a microservice application will work like a *distributed monolith* i.e., an application deployed like a microservice but built like a monolith.

Since a microservice is fine-grained, its functionality characteristically comprises several related use cases. As an example, *Flight Booking* service has a functionality that can encompass multiple use cases such as *Book Flight*, *View Booking*, and *Cancel Booking*. Theoretically, these use cases should be bundled together in a microservice. Often, grouping of the use cases exceeds the functionality provided by a single class.

[1] https://samnewman.io/blog/2015/04/07/microservices-for-greenfield/

We have devised a five-step approach namely GreenMicro to identify microservices for greenfield developments based on clustering of the use cases. GreenMicro uses various criteria involving functional requirement specifications- i) use cases, ii) dependency relationships among use cases, and iii) database entities. A similarity matrix is computed using these criteria and a clustering algorithm is applied to find candidate microservices. We have evaluated our approach on an in-house proprietary application and three sample benchmark applications.

In a microservice-based application, multiple services communicate with one another. Representational State Transfer (REST) is an established way of designing, architecting, and developing modern APIs. For our work, we also make use of RESTful API design which is stateless in nature. Thus proposed approach identifies stateless microservices.

At a high level, several services in the application need to work with each other in order to perform certain business task. There are two approaches for integration of microservices – orchestration (centralized) and choreography (reactive). The adoption of any one of them depends upon user's business needs and goals. In our work, we make use of API Gateway that act as the orchestration module. Thus we have considered orchestration for integration of microservices.

Key contributions of this paper are as follows:

C1: A five-step microservice extraction approach GreenMicro based on grouping of use cases that are highly cohesive and loosely coupled at the same time.

C2: Applying the proposed approach on 'Teachers Feedback Web Application (TFWA)' as a Proof of Concept (PoC).

C3: Applying our approach on three benchmark applications - i) JPetStore, ii) AcmeAir, and iii) Cargo Tracking System.

C4: Validating our results quantitatively.

We have identified microservices using GreenMicro for some of the projects. Their code is available on Github (JPetStore,[2] TFWA[3]).

The remainder of the paper is organized as follows: Section 2 presents the related work going in the Greenfield decomposition domain. Section 3 describes the proposed methodological plan for microservice identification. Section 4 walks through an in-house Java application taken as an example project to describe our approach and illustrate the implementation on three sample benchmark web applications. Section 5 shows quantitative analysis of our methodology along with the results. Section 6 states the conclusion.

II. RELATED WORK

There exist several techniques for identifying microservices in greenfield development. The techniques use the requirement documents, requirement models or design documents to achieve candidate services. Here, we discuss the existing approaches.

Both [8], [9] use dataflow-driven decomposition techniques for identifying microservices from the given detailed software requirements. In [8] a top-down decomposition approach is used where authors have converted traditional dataflow diagram (DFD) to get purified DFD. A two-phase automation algorithm for decomposition is proposed: (1) generating a decomposable DFD from purified DFD; (2) identifying candidate microservice from decomposable DFD. Li *et al.* [9] suggested a semi-automatic dataflow-driven microservice decomposition approach to generate a process-data store version of DFD ($\mathrm{DFD_{PS}}$). $\mathrm{DFD_{PS}}$ shows the relation between processes and related data stores. Next, a condensed DFD called decomposable DFD is taken out from $\mathrm{DFD_{PS}}$ by extracting the sentence sets in which a process reads or writes data to a data store. Last step of their proposed approach is to group modules of fine-grained processes and the related data stores to identify microservices.

Amiri *et al.* [10] presents a microservice identification method based on a set of business processes. Authors identified fine-grained services from business processes. They used the notions of structural and object dependency between business activities represented as business process model notation (BPMN). Ahmadvand [11] proposes a methodology that reconciles security and scalability requirements to be included in the requirements engineering phase. Their approach maps functional and non-functional requirements to identify more optimal system decomposition.

Baresi *et al.* [12] propose semantic similarity of available functionality evaluated through OpenAPI specifications. Their approach identifies potential candidate microservices by matching the key terms in the specifications against a reference vocabulary and suggests possible decompositions. The success of their approach is dependent on well-defined Application Programming Interfaces that give meaningful names.

Another service identification technique [13] exist which is based on functional decomposition of use case requirement. They first create a model of the system that contains a finite set of system operations and of the system's state space. Later, authors created an operational/ relational dependency graph using some automated tools to derive possible decomposition.

Fan *et al.* [14] analyze the system architecture using Domain-Driven Design (DDD) and extract the candidate microservices. Later, they analyze the database schema to verify if it is consistent with the candidate microservices. Finally they filter out inappropriate service candidates. In [15], an automatic identification approach has been proposed from a set of business processes. Their multi-model approach combines different independent models that represent a business process like control, data, and semantic dependencies [16]. Rivera *et al.* [17] propose another intelligent and optimal method that works on user stories to decompose the functionalities or requirements of the application into microservices. Most of these existing approaches have

[2] https://github.com/anitagoel/JPetStore.git
[3] https://github.com/anitagoel/TFWA.git

two important disadvantages: 1) Lack of validation of the results, and 2) Over dependency of their approach on expert opinion.

Until now, we see that researchers have used various techniques in greenfield developments. We build our approach on use cases as use cases are the most common and straightforward way to model the business functional requirements of a system that are defined during the early phase of software development. Use cases delineate a high-level view of the system without delving into the system intricacies. A complete set of use cases indicates all the functionality and behavior of the system. Thus, in our work, we harness the ability of use cases model, and finally group these business use cases into a set of candidate microservices. To achieve this, we describe a systematic approach GreenMicro to identify microservices in the early analysis and design phase, based on the system's functional requirements illustrated as use cases.

III. METHODOLOGY

Use case modeling is extensively used in contemporary software development engineering as an approach for requirements elicitation [43]. Further, relationships between use cases illustrates the dependency and connection between individual use cases elements of the system. These relationships add semantics to use case model by defining structure and behavior between the model elements. Also, use cases accessing the same data are more related than other use cases. Therefore, we present a systematic methodology to find similarity of use cases by making use of use case relationships and database entities which lead to finding candidate microservices.

A. BASIC DEFINITION

In this section, we will mathematically represent the problem of determining a set of microservices using use cases.

Consider a set of use cases as UC^A such that $UC^A = \{UC^1, UC^2, \ldots, UC^k\}$ where UC^1represents an individual use case. Using this, we define a set of microservices as $\mu^A = \{\mu_1, \mu_2, \ldots, \mu_n\}$ defined on UC^A such that

- $\bigcup_{i=1}^{n} \mu_i = UC^A$. It means all use cases are designated to some microservice.
- $\mu_i \neq \emptyset, \forall i = 1, \ldots, n$. It means there is no empty microservice.
- $\mu_i \cap \mu_j = \emptyset, \forall i, j = 1, \ldots, n$. It means each microservice is unique.

The main idea for our approach is to identify microservices by assessing the functional dependencies/ relationship between use cases. This can be implemented by clustering closely associated use cases into a service. Due to the informal nature of functional description of use cases, the degree of dependency between them may not be calculated directly. Therefore, we propose the approach GreenMicro to effectively determine the functional dependency between two use cases, UC^i and UC^j.

B. PROPOSED APPROACH

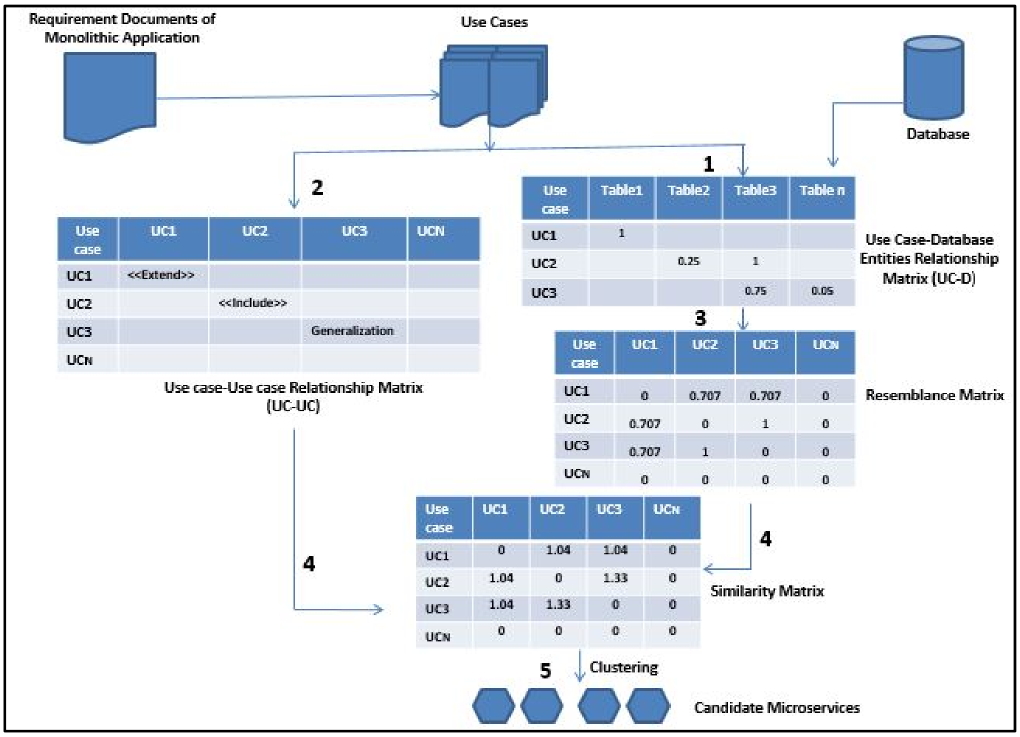
Use cases are most popular means of capturing business functional requirements. Use cases describe the high-level functions and scope of a system. So, our proposed approach is making use of business functionalities captured via use cases. GreenMicro is a systematic approach for microservice identification comprising of a sequence of steps that are logical and easy to apply in practice. The proposed approach consists of five steps as shown in Figure 2. We assume that use case models are fundamental artifacts of any object-oriented system and are readily available to system designers. Use case set, UC^A, can be gathered manually from the requirement documents or can be automated by parsing the given software requirements using NLP based language models, as discussed in [44].

An input dataset is a $<$ component $-$ attribute $>$ data matrix. Components are the entities (use cases) that we want to combine on the basis on their similarities. Attributes are the properties of the components. Two datasets utilized in our work for identifying microservices are as - i) Use Case-Use Case Relationship Matrix, and ii) Use Case-Database Entities Relationship Matrix. These two datasets are the artifacts of requirements analysis and design phase in application development. Both these relationship matrices are aggregated to get a combined similarity matrix. In subsequent steps of the approach, clustering is performed on the combined similarity matrix to get the desired results. Below are steps we utilize for microservice candidate identification:

1) Use Case-Database Entities Relationship Matrix (UC-D) - Use cases accessing the same data are more related than other use cases. In other words, use cases manipulating the same database entities have some degree of relationship. Thus, by examining the functionality of use cases, one can determine the database entities manipulated by each use case. So, database entities are regarded as features and CRUD operations applicable on them by specific use cases are recorded in Use Case-Database Entities Relationship Matrix (also known as CRUD matrix). This matrix indicates use cases UC^A as rows, database entities as columns, and semantic relationship tags as Create - C, Read - R, Update - U and Delete - D as cells of the matrix [18].
   - "C" means use case CREATE database entity.
   - "R" means use case READ database entity.
   - "U" means use case UPDATE database entity.
   - "D" means use case DELETE database entity.

Now, this tag-based matrix is transformed into numeric values. Therefore, each tag is replaced with the corresponding value (weight) in the matrix according to the priority as C $>$ U $>$ D $>$ R. In our work, to simplify computations, we adopt these substitutions as: C: $=1$, U: $= 0.75$, D: $= 0.5$, R: $= 0.25$, as suggested in [18].

1) Use Case-Use Case Relationship Matrix (UC-UC) - This matrix represents the degree of interdependence among all the use cases. Three types of relationships


Requirement Documents of Monolithic Application
Use Cases
Database
1
2
3
4
4
5
Use Case-Database Entities Relationship Matrix (UC-D)
Use case-Use case Relationship Matrix (UC-UC)
Resemblance Matrix
Similarity Matrix
Clustering
Candidate Microservices

Use case	UC1	UC2	UC3	UCN
UC1	<<Extend>>			
UC2		<<Include>>		
UC3			Generalization	
UCN				

Use case	Table1	Table2	Table3	Table n
UC1	1			
UC2		0.25	1	
UC3			0.75	0.05

Use case	UC1	UC2	UC3	UCN
UC1	0	0.707	0.707	0
UC2	0.707	0	1	0
UC3	0.707	1	0	0
UCN	0	0	0	0

Use case	UC1	UC2	UC3	UCN
UC1	0	1.04	1.04	0
UC2	1.04	0	1.33	0
UC3	1.04	1.33	0	0
UCN	0	0	0	0

FIGURE 2. **Complete outline of the GreenMicro approach.**

between use cases exist - Include, Extend and Generalization as discussed in the Introduction section. Here, use cases, UC^A, are represented in both rows and columns and their relationships are marked as cells of the matrix. In this paper, we adopt substitutions as: Generalization: = 1, Include: = 0.66, Extend: = 0.33, as suggested in [19].

2) Constructing a Resemblance Matrix for UC-D - Here, we find the resemblance coefficient between each use case entry in UC-D Matrix. Each row of the UC-D matrix can be seen as a vector whose similarity needs to be evaluated. To find degree of similarity or dissimilarity between these two use cases, we use the Cosine Similarity between pair wise component vectors to represent the weight of the relationship between use cases. Cosine similarity is a measurement that quantifies similarity between two or more vectors. Cosine similarity is a common approach and has been used in many other related studies for microservice identification [20], [21]. This score indicates how much two use cases are related in terms of accessing the database entities. The higher the value, the stronger the relationship between use cases.

$$\cos(x, y) = x.y/||X||^{*}||Y||$$

where, x. y = dot product of the vectors 'x' and 'y' and $||x|| * ||y||$ = cross product of the two vectors 'x' and 'y'.

1) Generate Combined Similarity Matrix - In this step, we create a Combined Similarity Matrix of use cases which is a N×N symmetric matrix. Here (i, j)-th element represents the similarity measure for the UC^i and UC^j where i, j = 1,…,N. For this, Resemblance Matrices for UC-D and UC-UC are aggregated together to generate a Combined Similarity Matrix.

2) Clustering of use cases - The Combined Similarity Matrix generated in the previous step is the input for clustering technique. We perform clustering to obtain a cohesive set of use cases that may be bundled together as microservices. Given a set of use cases, UC^A, it involves organizing each use case into a specific group called cluster. In our work, we consider each use case UC^i as a distinct object. Use cases of the same cluster are likely to be as homogeneous as possible to make sure the cohesion property of a cluster. In contrast, use cases belonging to different groups are likely to be as distinct as possible to make sure the loose coupling among clusters.

For our work, we have applied classical Hierarchical Agglomerative Clustering (HAC) [22] for two reasons. Firstly, it has been utilized in numerous earlier works on software re-modularization [23] and microservice candidate identification [24], [25], [20]. Secondly, it has less time complexity in comparison to the hill-climbing technique [26] and genetic algorithms [27], [10]. Despite the fact that hierarchical clustering provides a graphical representation of a fully connected hierarchical tree as dendrograms, we can find the optimal number of clusters to be extracted. For this, we executed the Silhouette method that suggested the optimal number of clusters. Each group of use cases can be considered a potential microservice candidate. By the end of this step,

TABLE 1. Subject applications used in evaluation.

Application	# Use Case	# Database Tables	Used in prior evaluations
JPetStore	21	13	[27], [28], [29], [30], [31], [21], [32]
AcmeAir	21	6	[28], [33], [34], [35], [36], [37], [38]
Cargo Tracking System	32	9	[17], [9], [12], [16], [39], [38], [3]
TFWA	22	7	[40]

we find possible microservices μ^A from the given set of use cases UC^A.

IV. EMPIRICAL EVALUATIONS

In this section, we present subject applications on which empirical evaluations are being performed. We also describe baseline techniques with which results of GreenMicro are compared. We also elaborate quality assessment metrics that are utilized in this evaluation work.

A. SUBJECT APPLICATIONS

We use four Java Enterprise Edition (J2EE) applications for the evaluation - three open-source benchmark applications namely JPetStore, AcmeAir, and Cargo Tracking System, and one in-house application namely Teachers Feedback Web Application (TFWA). The three open-source benchmark applications have been chosen as they have been used most predominantly in previous evaluations in academic research related to microservice identification. Table 1 presents the basic information about the four applications, like, number of use cases and number of database tables.

TFWA: Teachers Feedback Web Application (TFWA) automates the process of the feedback system for teachers. Students use TFWA to submit their feedback for all subjects and the respective teachers. Students can give their feedback online. Visually impaired student is a specialized actor that can give feedback through a voice-bot. Teacher In-Charge (TIC) of every department and Principal can check the status of the feedback procedure and can view and analyze the feedback data from different analytics perspectives (department-wise, teacher-wise, class-wise and subject-wise). All use cases are depicted in Figure 3. There are mainly four actors that interact with the system - Student, TIC, Principal, and Admin.

We extracted twenty two use cases and seven database tables for TFWA. Thus the dimension of the UC-D matrix is 22 × 7. UC-UC matrix is a 22 × 22 matrix depicting relationships between use cases. Combined Similarity Matrix is also a 22 × 22 matrix on which finally the clustering algorithm is applied. Figure 4, Figure 5 and Figure 6 show the excerpts of Use Case-Database Entities Relationship Matrix (UC-D), Use Case-Use Case Relationship Matrix (UC-UC) and Combined Similarity Matrix respectively for TFWA.

Brief description of the sample benchmark applications is as follows:

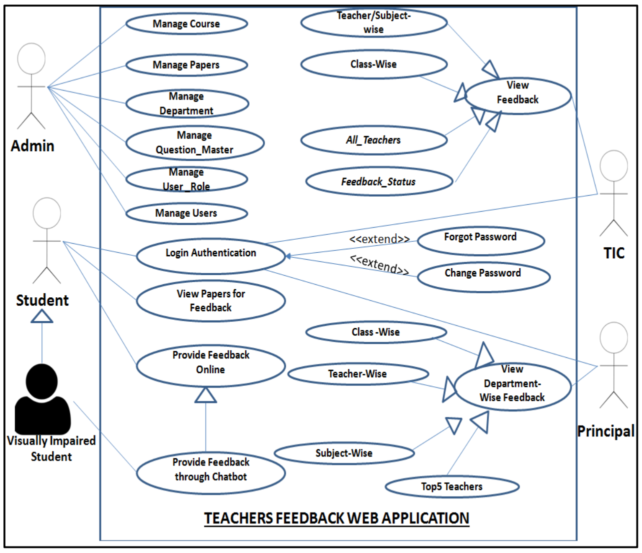


FIGURE 3. Use case diagram for TFWA.

Use cases	Database Tables						
	Course	Dept	Subject	Feed back	Ques Templ ate	User Role	User Deta il
UC1	0	0	0	0	0	0.25	0.25
UC2	0	0	0	1	0.25	0	0
…….	..	..	..	..	..	..	..
UCn	0.25	0.25	0.25	0.25	0	0	0

(UC1: Login, UC2: Provide Feedback Online, UCn: View Class Feedback Data by Principal)

FIGURE 4. Excerpts of UC-D matrix for TFWA.

	UC1	UC2	UC3	UC4	……	UC22
UC1	-	0.33	0	0	0	0
UC2	0.33	-	0.66	0	0	0
UC3	0	0.66	-	0.33	0	0
UC4	0	0	0.33	-		0
……	0	0	0	0	-	1
UC22	0	0	0	0	1	-

FIGURE 5. Excerpts of UC-UC matrix for TFWA.

JPetStore[4]: An e-commerce application for pet shopping that allows users to browse catalog, products and items, add

[4]https://github.com/mybatis/jpetstore-6

	UC1	UC2	UC3	UC4	……	UC22
UC1	0	1.037	1.037	0	0	0
UC2	1.037	0	1.33	0	0	0
UC3	1.037	1.33	0	0	0	0
UC4	0	0	0	0	0.816	0.816
……	0	0	0	0.816	0	2
UC22	0	0	0	0.816	2	0

FIGURE 6. Excerpts of combined similarity matrix for TFWA.

items to cart, remove items from cart, update cart items and purchase items within several categories of pets.

AcmeAir[5]: A fictitious airline system that handles booking flights, cancel flights, search flights, customer account information, authentication, and baggage services.

Cargo Tracking System[6]: The focus of the application is to track the movement of a Cargo (which is uniquely identified by a TrackingId) between two Locations listed through a RouteSpecification. Once a Cargo is booked, it is linked with one of the Itineraries (record of CarrierMovements) as selected from existing Voyages. HandlingEvents tracks the movement of the Cargo on the Itinerary. When the cargo is handled, its delivery status changes. The life cycle of a cargo ends when the cargo is claimed by the customer.

B. RELATED TECHNIQUES FOR QUANTITATIVE EVALUATION

The objective of our evaluation is to assess whether GreenMicro can identify effective microservice candidates. For JPetStore, and AcmeAir application, we compare our approach with four well-known baselines for microservice identification that have presented their results for selected applications: FoSCI [27], CoGCN [36], Mono2Micro [28] and MEM [2].

FoSCI[7] collects and processes the execution traces of the monolithic application, and identifies services candidates using a search-based functional atom grouping algorithm using a hierarchical clustering approach. Later FoSCI assigns these functional atoms to microservice candidates by merging them using a genetic algorithm.

CoGCN proposes a multi-objective Graph Convolution Network approach to partition monolith applications using graph based clustering. The technique minimizes the effect of structural and attribute outlier classes that could be in the embeddings of other classes.

Mono2Micro[8] employs a spatial-temporal decomposition technique that leverages running of selected business use cases of the monolithic application and dynamically collects runtime call traces to find functionally cohesive clusters of application classes. Business use cases comprise the space dimension and control flow of the dynamic runtime traces convey the time dimension.

MEM[9] makes use of Kruskal's algorithm to find minimum spanning tree (MST) for the monolithic application. This technique has two transformation stages. In the construction step, the monolith is transformed into the graph representation using three coupling criteria – logical, contributor and semantic coupling. In the clustering step, graph representation is decomposed to generate partitions behaving as microservice candidates.

The above listed baseline approaches do not perform their analysis on the Cargo Tracking System. So, for this application, we compare our approach with other four well-known baselines for microservice identification: Service Cutter [39], API Interface Analysis [12], DFD Analysis [9] and Business Processes Analysis [16].

Service Cutter[10] is a state-of-the-art approach for microservice identification. The inputs to the tool are a set of requirement artifacts, and weighted coupling criteria. Tool outputs a graph where nodes indicate candidate services, and weight along the arcs represents how cohesive/ coupled two candidate services are. Lastly, a clustering algorithm suggests the most appropriate microservice cuts.

API Interface Analysis[11] involves semantic similarity of available functionality evaluated through OpenAPI specifications. This approach identifies potential candidate microservices by matching the key terms in the specifications against a reference vocabulary and suggests possible decompositions.

DFD Analysis suggests a semi-automatic dataflow-driven microservice decomposition approach to generate a process-data store version of DFD (DFD_{PS}). DFD_{PS} shows the relation between processes and related data-stores. This $DFD_{PS}$is condensed to get a decomposable DFD, in which the sentences between processes and data-stores are joined.

Business Processes Analysis utilizes a multi-model approach that combines different independent models that represent business process activities like control, data, and semantic dependencies. A clustering algorithm further applied to combine all extracted dependencies for identifying microservices.

C. METRICS UTILIZED

For the selected benchmark applications, we could not find quality assessment metrics that are common for all three applications. Researchers have validated their work on varied sets of quality attributes. Therefore, we test these benchmark

[5]https://github.com/acmeair/acmeair
[6] https://github.com/citerus/dddsample-core
[7]https://github.com/wj86/FoSCI/releases

[8]https://github.com/kaliaanup/Mono2Micro-FSE-2021
[9] https://github.com/gmazlami/microserviceExtraction-backend
[10] https://github.com/ServiceCutter/ServiceCutter
[11] https://github.com/mgarriga/decomposer

applications for those quality assessment metrics that are used in other baseline studies. This eventually gives us an opportunity to assess and validate our approach on a diverse spectrum of quality attributes.

We apply the following mentioned five quality metrics for JPetStore and AcmeAir applications namely i) SM, ii) ICP, iii) BCP, iv) IFN, and v) NED to measure the effectiveness of partitions recommended using GreenMicro.

Structural Modularity (SM), as defined in [28], [36], [27], measures modular quality of a microservice from a structural view. Higher the SM, better modularized the microservice is.

$$SM = \frac{1}{N}\sum_{i=1}^{N} scoh_i - \frac{1}{\frac{N(N-1)}{2}}\sum_{i \neq j}^{N} scop_{i,j}$$
$$where\ scoh_i = \frac{u_i}{N_i * N_i} \text{ and } scop_{i,j} = \frac{\sigma_{i,j}}{2(N_i X N_j)}$$

scoh$_\mathrm{i}$ measures the structural cohesiveness of a microservice (intra-connectivity) and **scop**$_\mathrm{i,j}$measures coupling between microservices (inter-connectivity). N refers to the number of services. u_i refers to the number of edges inside a service i. $\sigma_{i,j}$ refers to the number of edges between microservice i and j. N_i or N_j means the number of entities (both classes and methods) in microservice i or j. The larger scoh, and the lesser scop, the better SM.

Inter-Partition Call percentage (ICP) [28], [35] amounts to the percentage of calls between two microservices i and j.

$$ICP_{i,j} = \frac{C_{i,j}}{\sum_{i=1,j=1,i\neq j}^{N} C_{i,j}}$$

where $C_{i,j}$ measures the number of call between microservices i and j. The lesser the value of ICP, the better is the microservice identification.

Business Context Purity (BCP) [28], [41] indicates the mean entropy of business use cases per partition. A microservice is considered functionally cohesive if it implements a lesser number of use cases. Mathematically,

$$BCP = \frac{1}{N}\sum_{i=1}^{N} \frac{BC_i}{\sum_j BC_j} \log_2 \left(\frac{BC_i}{\sum_j BC_j}\right)$$

where N is the number of services and B_{Ci} indicates the number of business use cases in microservice i. Since BCP is primarily based on entropy, lesser values are better.

Interface Number (IFN) [27], [28], [36] indicates the average number of published interfaces exposed by a microservice to other services. Smaller the value of IFN, the better it is as the service follows the Single Responsibility Principle. A service publishing a large number of interfaces may provide numerous functionalities, thus violating SRP. Mathematically, it can be represented as

$$IFN = \frac{1}{N}\sum_{j=1}^{N} ifn_j$$

Non-Extreme Distribution (NED) [28], [36] measures how evenly distributed the sizes of the recommended microservice is. In general, it is preferred not to have a microservice that has too many or too few classes.

$$NED = 1 - \frac{\sum_{i=1,i\ not\ extreme}^{N} n_i}{|V|}$$

where $\mathrm{n_i}$ is the number of classes in service i and V is the set of classes. i is not extreme if its size is within bounds of $\{5{:}20\}$. The Less the value of NED, the better it is.

For quantitative evaluation of the Cargo Tracking System, we couldn't find any research paper where above listed metrics are utilized. So, we make use of another four object-oriented design metrics namely i) Number of Incoming Dependencies, ii) Number of Outgoing Dependencies, iii) Instability, and iv) Relational Cohesion[12] as used in other baseline techniques [9], [12], [16], [39]. In general, all the metrics are based on coupling, cohesion and number of interactions between microservices. A concise description of these metrics is given below:

Number of Incoming Dependencies - Measures the number of classes outside this microservice that depend upon classes within this microservice. It is also called afferent coupling (Ca).

Number of Outgoing Dependencies - Measures the number of classes inside this microservice that depend on classes outside this microservice. It is also called efferent coupling (Ce).

Instability Index (I) - Indicates service's resilience to change. It has a range from 0 to 1(both inclusive). I = 0 (maximally stable service), means no method in this service has a dependency to any other method or class in another service. If there are no outgoing dependencies, then Instability will be 0 and the measured service is stable. If there are no incoming dependencies, then Instability will be 1 and the measured element is unstable. Stable means that the element is not so easy to change. It can be calculated as the ratio Ce / (Ca + Ce).

Relation Cohesion (RC) - It is a measure of the number of internal relations that represent class inheritance, method invocations, access to class attributes etc. Higher values of relational cohesion suggest more cohesion.

V. RESULTS

We evaluated the performance of GreenMicro for the three benchmark applications and one in-house application. For each application, we applied our approach and grouped a set of cohesive use cases to obtain microservices. We compared the Cargo Tracking System against four baselines on four evaluation metrics. Further, we compared AcmeAir and JPetStore against anther four baselines on five evaluation metrics.

Table 2, 3, 4 and 5 present the comparison of our results for all four applications across two sets of metrics. For all the

[12]http://eclipse.hello2morrow.com/doc/standalone/content/java_metrics.html

TABLE 2. Comparison of metrics for acmeair.

AcmeAir	Green Micro	M2M [28]	FoSCI [27]	CoGCN [36]	MEM [2]
SM (+)	**0.112**	0.072	0.095	0.038	0.097
ICP (-)	**0.318**	0.527	0.706	0.444	0.589
BCP (-)	**0.545**	0.953	1.539	1.221	1.827
IFN (-)	**2.75**	3.375	4.375	2.846	4.333
NED (-)	0.266	0.429	0.407	**0.250**	0.464

TABLE 3. Comparison of metrics for jpetstore.

JPetStore	Green Micro	M2M [28]	FoSCI [27]	CoGCN [36]	MEM [2]
SM (+)	0.102	0.054	0.044	0.091	**0.124**
ICP (-)	**0.220**	0.333	0.478	0.582	0.434
BCP (-)	**0.502**	1.625	2.181	1.905	2.496
IFN (-)	**1.75**	1.845	3.750	2.533	3.429
NED (-)	**0.125**	0.257	0.516	0.392	1.000

TABLE 4. Comparison of metrics for cargo tracking system.

Cargo Tracking System	Green Micro	Service Cutter [39]	API Interface [12]	DFD Analysis [9]	Business Processes [16]
Number of Incoming Dependencies (-)	**11.75**	13.3	13	13.5	15.2
Number of Outgoing Dependencies (-)	5	**2.7**	3.3	3.3	3.6
Instability Index (-)	0.30	**0.2**	**0.2**	**0.2**	0.21
Relational Cohesion (+)	**15.19**	14.2	9.3	12.2	13.25

metrics, we have assigned two labels as "(-)" or "(+)". Label "(-)" indicates lower values are better, and a label "(+)" indicates higher values are better.

For the AcmeAir application, GreenMicro performed better than other approaches for SM, ICP, BCP and IFN as shown in Table 2. GreenMicro's better results for BCP highlight that use case based partitions are more functionally consistent. For NED, our values are slightly higher than the values presented by CoGCN.

For JPetStore, GreenMicro yields better results for ICP, BCP, IFN and NED. Lower ICP indicates lesser call percentage between services. Winning in terms of NED indicates that the majority of the services contain 5 to 20 classes as shown in Table 3. It may be noted that the value of SM metric is slightly lower than MEM (highest).

For the Cargo Tracking System, metrics are evaluated using SonarGraph Architect (12.0.4.713 version) [42]. SonarGraph can measure and monitor the technical quality assessment metrics. From a comparative perspective, our results reveal better performance in terms of *Number of Incoming Dependencies* and *Relational Cohesion* strength. It signifies more reusable, maintainable, and robust microservices. *Number of Outgoing Dependencies* and *Instability* metrics yields marginally higher value in comparison to other related studies, as shown in Table 4.

TABLE 5. Metrics for TFWA.

TFWA	GreenMicro
SM (+)	0.388
ICP (-)	0.338
BCP (-)	1.28
IFN (-)	1.75
NED (-)	0

(a)

TFWA	Candidate Microservices			
	Authentication	Feedback	Analytics	Average
Number of Incoming Dependencies (-)	5	2	2	3
Number of Outgoing Dependencies (-)	3	1	2	2
Instability Index (-)	0.38	0.33	0.33	0.34
Relational Cohesion (+)	14.71	13	5.92	11.21

(b)

TABLE 6. Comparison of metrics for TFWA.

Metrics	Legacy TFWA	Microservice TFWA using GreenMicro
System Maintainability Level (+)	60	**70**
Cyclic Java Packages (-)	8	**6**
Component Dependencies to Remove (-)	9	**6**
Structural Debt Index (-)	124	**78**
Physical Cohesion (+)	1.91	**2.38**
Physical Coupling (-)	1.79	**1.32**

For TFWA, we achieve three microservices namely *Authentication*, *Feedback*, and *Analytics*. Table 5(a) and (b) represents quality assessment metrics for both sets of metrics discussed above. GreenMicro improves the long-term health, quality, maintainability of the application and yields better software restructuring.

For TFWA, we also compare and present the results of legacy monolithic application and microservices application, developed according to GreenMicro. Table 6 aggregates the quality assessment parameters provided by SonarGraph Architect as described below:

- **System Maintainability Level** - Evaluates maintainability (in %) by assessing dependency structure between components in source files. Cyclic dependencies and incoming dependencies negatively influence the metric.

- **Cyclic Java Packages** - Number of Java packages involved in a cycle.
- **Component Dependencies to Remove** - Number of component dependencies to remove to break up all Java package cycle groups.
- **Structural Debt Index** - An estimation of the work needed to clean a software project from structural drift and erosion which happened due to unwanted dependencies that violates architectural rules and cyclic dependencies between packages.
- **Physical Cohesion** - Number of dependencies 'to' and 'from' other components in the same module.
- **Physical Coupling** - Number of dependencies 'to' and 'from' other components in other modules.

Our results show that microservice application yields reduced cyclic package dependencies, structural erosion, structural debt index and improved system maintainability level. It will consequently improve long-term health, quality, and maintainability of the application. To conclude, microservice identification performed by our approach has greater cohesion, smaller coupling, lesser number of operations offered by a microservice and lesser average calls from one microservice to another. Thus, these results display satisfactory PoC.

VI. CONCLUSION

Microservices are one of the most popular concepts in web application development. A microservice is a small, independent, loosely coupled and high cohesive service that is based on bounded-context. This new paradigm brings a lightweight, independent, reuse-oriented, and fast service deployment approach that minimizes infrastructural risks. However, microservices identification remains an important hurdle for system architects and designers. This task becomes even more challenging in greenfield deployments. Finding microservices before the code exists, as done in our approach, enables system architects to design software that is of higher design quality. We propose a microservice identification approach, GreenMicro that makes use of business use cases, their inter-dependence and associated data dependencies as the primary sources of input. We applied a clustering algorithm on the combined similarity matrix to identify microservices.

We understand that orchestration and choreography is an implementation choice consideration which is an excellent area to explore in future work. For evaluation, GreenMicro is applied to four enterprise Java applications to recommend candidate microservices. The initial results are promising demonstrating better structural modularity, higher cohesion and lower inter-service calling. In our future work, we will perform further evaluations of GreenMicro on real-world big scale enterprise applications. We also understand that orchestration and choreography is an implementation time consideration which is an excellent area to explore in future work.

REFERENCES

[1] I. K. Aksakalli, T. Celik, A. B. Can, and B. Tekinerdogan, "Systematic approach for generation of feasible deployment alternatives for microservices," *IEEE Access*, vol. 9, pp. 29505–29529, 2021.

[2] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jun. 2017, pp. 524–531, doi: 10.1109/ICWS.2017.61.

[3] H. Vural and M. Koyuncu, "Does domain-driven design lead to finding the optimal modularity of a microservice?" *IEEE Access*, vol. 9, pp. 32721–32733, 2021, doi: 10.1109/ACCESS.2021.3060895.

[4] D. Bajaj, U. Bharti, A. Goel, and S. C. Gupta, "A prescriptive model for migration to microservices based on SDLC artifacts," *J. Web Eng.*, vol. 20, no. 3, pp. 817–852, Jun. 2021, doi: 10.13052/jwe1540-9589.20312.

[5] G. R. Shahmohammadi, S. Jalili, and S. M. H. Hasheminejad, "Identification of system software components using clustering approach," *J. Object Technol.*, vol. 9, no. 6, pp. 77–98, 2010, doi: 10.5381/jot.2010.9.6.a4.

[6] T. von der Maßen and H. Lichter, "Modeling variability by UML use case diagrams," in *Proc. Int. Workshop Requirements Eng. Product Lines*, 2002, pp. 19–25.

[7] M. Kalske, "Transforming monolithic architecture towards microservice architecture," M.S. thesis, Dept. Comput. Sci., Univ. Helsinki, Helsinki, Finland, 2017, p. 72.

[8] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *Proc. 24th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2017, pp. 466–475, doi: 10.1109/APSEC.2017.53.

[9] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, and Z. Shan, "A dataflow-driven approach to identifying microservices from monolithic applications," *J. Syst. Softw.*, vol. 157, Nov. 2019, Art. no. 110380, doi: 10.1016/j.jss.2019.07.008.

[10] M. J. Amiri, "Object-aware identification of microservices," in *Proc. IEEE Int. Conf. Services Comput. (SCC), IEEE World Congr. Services*, Jul. 2018, pp. 253–256, doi: 10.1109/SCC.2018.00042.

[11] M. Ahmadvand and A. Ibrahim, "Requirements reconciliation for scalable and secure microservice (De)composition," in *Proc. IEEE 24th Int. Requirements Eng. Conf. Workshops (REW)*, Sep. 2016, pp. 68–73, doi: 10.1109/REW.2016.026.

[12] L. Baresi, M. Garriga, and A. De Renzis, "Microservices identification through interface analysis," in *Proc. Eur. Conf. Service-Oriented Cloud Comput.*, in Lecture Notes in Computer Science: Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics, vol. 10465, Nov. 2017, pp. 19–33, doi: 10.1007/978-3-319-67262-5_2.

[13] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, "Identifying microservices using functional decomposition," in *Proc. Int. Symp. Dependable Softw. Eng., Theories, Tools, Appl.*, in Lecture Notes in Computer Science: Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics, vol. 10998, 2018, pp. 50–65, doi: 10.1007/978-3-319-99933-3_4.

[14] C.-Y. Fan and S.-P. Ma, "Migrating monolithic mobile application to microservice architecture: An experiment report," in *Proc. IEEE Int. Conf. AI Mobile Services (AIMS)*, Jun. 2017, pp. 109–112, doi: 10.1109/AIMS.2017.23.

[15] M. Daoud, A. El Mezouari, N. Faci, D. Benslimane, Z. Maamar, and A. El Fazziki, "Towards an automatic identification of microservices from business processes," in *Proc. IEEE 29th Int. Conf. Enabling Technol., Infrastruct. Collaborative Enterprises (WETICE)*, Sep. 2020, pp. 42–47, doi: 10.1109/WETICE49692.2020.00017.

[16] M. Daoud, A. El Mezouari, N. Faci, D. Benslimane, Z. Maamar, and A. El Fazziki, "A multi-model based microservices identification approach," *J. Syst. Archit.*, vol. 118, Sep. 2021, Art. no. 102200, doi: 10.1016/j.sysarc.2021.102200.

[17] F. H. Vera-Rivera, E. G. Puerto-Cuadros, H. Astudillo, and C. M. Gaona-Cuevas, "Microservices backlog—A model of granularity specification and microservice identification," in *Proc. Int. Conf. Services Comput.*, in Lecture Notes in Computer Science: Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics, vol. 12409, Sep. 2020, pp. 85–102, doi: 10.1007/978-3-030-59592-0_6.

[18] M. Marbout and F. Shams, "An automated service realization method," *Int. J. Comput. Sci. Issues*, vol. 9, no. 4, pp. 188–195, 2012.

[19] D. Kang, B. Xu, J. Lu, and W. C. Chu, "A complexity measure for ontology based on UML," in *Proc. 10th IEEE Int. Workshop Future Trends Distrib. Comput. Syst. (FTDCS)*, May 2004, pp. 222–228.

[20] L. J. Kirby, E. Boerstra, Z. J. C. Anderson, and J. Rubin, "Weighing the evidence: On relationship types in microservice extraction," in *Proc. IEEE/ACM 29th Int. Conf. Program Comprehension (ICPC)*, May 2021, pp. 358–368, doi: 10.1109/ICPC52881.2021.00041.

[21] M. Brito, J. Cunha, and J. Saraiva, "Identification of microservices from monolithic applications through topic modelling," in *Proc. 36th Annu. ACM Symp. Appl. Comput.*, Mar. 2021, pp. 1409–1418, doi: 10.1145/3412841.3442016.

[22] W. H. E. Day and H. Edelsbrunner, "Efficient algorithms for agglomerative hierarchical clustering methods," *J. Classification*, vol. 1, no. 1, pp. 7–24, 1984.

[23] C. Patel, A. Hamou-Lhadj, and J. Rilling, "Software clustering using dynamic analysis and static dependencies," in *Proc. 13th Eur. Conf. Softw. Maintenance Reeng.*, 2009, pp. 27–36.

[24] A. Selmadji, A.-D. Seriai, H. L. Bouziane, C. Dony, and R. O. Mahamane, "Re-architecting OO software into microservices," in *Proc. Eur. Conf. Service Cloud Comput.*, 2018, pp. 65–73, doi: 10.1007/978-3-319-99819-0_5.

[25] S. Eski and F. Buzluca, "An automatic extraction approach—Transition to microservices architecture from monolithic application," in *Proc. ACM Int. Conf. Agile Softw. Develop.*, 2018, pp. 1–6, doi: 10.1145/3234152.3234195.

[26] K. Mahdavi, M. Harman, and R. M. Hierons, "A multiple Hill climbing approach to software module clustering," in *Proc. Int. Conf. Softw. Maintenance (ICSM)*, 2003, pp. 315–324.

[27] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Trans. Softw. Eng.*, vol. 47, no. 5, pp. 987–1007, May 2021, doi: 10.1109/TSE.2019.2910531.

[28] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, and D. Banerjee, "Mono2Micro: A practical and effective tool for decomposing monolithic Java applications to microservices," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Aug. 2021, pp. 1214–1224, doi: 10.1145/3468264.3473915.

[29] M. Architecture, "Refactoring monolithic object-oriented source code to materialize motivating example?: Information," in *Proc. ICSOFT*, vol. 117, 2021, p. 126.

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

[31] I. Saidani, A. Ouni, M. W. Mkaouer, and A. Saied, "Towards automated microservices extraction using muti-objective evolutionary search," in *Proc. Int. Conf. Service-Oriented Comput.*, in Lecture Notes in Computer Science: Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics, vol. 11895, Oct. 2019, pp. 58–63, doi: 10.1007/978-3-030-33702-5_5.

[32] O. Al-Debagy and P. Martinek, "A microservice decomposition method through using distributed representation of source code," *Scalable Comput.*, vol. 22, no. 1, pp. 39–52, 2021, doi: 10.12694:/scpe.v22i1.1836.

[33] R. Yedida, R. Krishna, A. Kalia, T. Menzies, J. Xiao, and M. Vukovic, "Lessons learned from hyper-parameter tuning for microservice candidate identification," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2021, pp. 1141–1145, doi: 10.1109/ase51524.2021.9678704.

[34] M. Abdullah, W. Iqbal, and A. Erradi, "Unsupervised learning approach for web application auto-decomposition into microservices," *J. Syst. Softw.*, vol. 151, pp. 243–257, May 2019, doi: 10.1016/j.jss.2019.02.031.

[35] R. Yedida, R. Krishna, A. Kalia, T. Menzies, J. Xiao, and M. Vukovic, Partitioning cloud-based microservices (via deep learning), vol. 1, no. 1, 2021, *arXiv:2109.14569*.

[36] U. Desai, S. Bandyopadhyay, and S. Tamilselvam, "Graph neural network to dilute outliers for refactoring monolith application," in *Proc. 35th AAAI Conf. Artif. Intell.*, 2021, vol. 35, no. 1, pp. 72–80.

[37] R. Nakazawa, T. Ueda, M. Enoki, and H. Horii, "Visualization tool for designing microservices with the monolith-first approach," in *Proc. IEEE Work. Conf. Softw. Vis. (VISSOFT)*, Sep. 2018, pp. 32–42, doi: 10.1109/VISSOFT.2018.00012.

[38] O. Al-Debagy and P. Martinek, "Dependencies-based microservices decomposition method," *Int. J. Comput. Appl.*, pp. 1–8, Apr. 2021, doi: 10.1080/1206212X.2021.1915444.

[39] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Proc. Eur. Conf. Service-Oriented Cloud Comput.*, in Lecture Notes in Computer Science: Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9846, 2016, pp. 185–200, doi: 10.1007/978-3-319-44482-6_12.

[40] D. Bajaj, U. Bharti, A. Goel, and S. C. Gupta, "Partial migration for re-architecting a cloud native monolithic application into microservices and FaaS," in *Proc. Int. Conf. Inf., Commun. Comput. Technol.*, 2020, pp. 111–124, doi: 10.1007/978-981-15-9671-1_9.

[41] A. K. Kalia, J. Xiao, C. Lin, S. Sinha, J. Rofrano, M. Vukovic, and D. Banerjee, "Mono2Micro: An AI-based toolchain for evolving monolithic enterprise applications to a microservice architecture," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Nov. 2020, pp. 1606–1610, doi: 10.1145/3368089.3417933.

[42] A. von Zitzewitz, "Mitigating technical and architectural debt with sonargraph," in *Proc. IEEE/ACM Int. Conf. Tech. Debt (TechDebt)*, May 2019, pp. 66–67.

[43] H. Gomaa and E. Olimpiew, "The role of use cases in requirements and analysis modeling," in *Proc. Workshop Use Cases Model-Driven Softw. Eng.*, Montego Bay, Jamaica, 2005, pp. 1–15.

[44] D. Bajaj, A. Goel, S. C. Gupta, and H. Batra, "MUCE: A multilingual use case model extractor using GPT-3," *Int. J. Inf. Technol.*, vol. 14, no. 3, pp. 1543–1554, May 2022, doi: 10.1007/s41870-022-00884-2.

DEEPALI BAJAJ is currently pursuing the Ph.D. degree in the area of cloud and distributed computing. She is also an Associate Professor with the Department of Computer Science, Shaheed Rajguru College of Applied Sciences for Women (University of Delhi). She has over 15 years of teaching experience at university level. She has authored several national and international research publications. Her research interests include microservices and function-as-a-service (FaaS) and serverless technology.

ANITA GOEL is currently a Professor with the Department of Computer Science, Dyal Singh College, University of Delhi, India. She has a work experience of more than 30 years. She is also a visiting faculty to several universities in India. She has been a fellow of Computer Science with the Institute of Life Long Learning (ILLL), University of Delhi. She has guided several students for their doctoral studies and has travelled internationally to present research papers. She is a serving member of program committee of several international conferences. She has authored books in computer science. She has several national and international research publications. Her research interests include cloud computing, microservices, serverless computing, software engineering, and technology-enhanced education (MOOC).

S. C. GUPTA is currently pursuing the B.Tech. degree in EE with IIT Delhi. He has worked with the Computer Group, Tata Institute of Fundamental Research and NCSDCT (now C-DAC Mumbai), until recently, he worked as the Deputy Director General of Scientist-G, and the Head of Training with the National Informatics Centre, New Delhi. He was responsible for keeping its 3000 scientists/engineers up to date in various technologies. He has extensive experience in design and development of large complex software systems. He is currently a Visiting Faculty with the Department of Computer Science and Engineering, IIT Delhi, where he has been teaching cloud computing, which includes emerging disruptive technologies like SDN and SDS. He has guided many M.Tech., and Ph.D. research students in these technologies. He has many publications in software engineering and cloud technology in national and international conferences and journals. His research interests include software engineering, data bases, and cloud computing.