GreenMicro: Identifying Microservices from Use Cases in Greenfield Development

Deepali Bajaj\textsuperscript{1}, Anita Goel\textsuperscript{2}, S. C. Gupta\textsuperscript{3}

\textsuperscript{1}Department of Computer Science, Shaheed Rajguru College of Applied Sciences for Women, University of Delhi, Delhi, India,
\textsuperscript{2}Department of Computer Science, Dyal Singh College, University of Delhi, Delhi, India, goel.anita@gmail.com
\textsuperscript{3}Department of Computer Science, Indian Institute of Technology Delhi, India, gutpa.drsc@gmail.com

Corresponding author: Deepali Bajaj (e-mail: 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 through 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. It is a technique of capturing high-level user functions and scope of the 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 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 producing 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 Developments, 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 a tough and expensive. A smallest 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.

Recently, Microservice Architecture (MSA) style is growing expeditiously, and many internet-based companies like Netflix, Amazon, eBay, and Uber are exploiting the digital transformation to restructure their legacy codebase \cite{1}. In MSA, applications are developed as a set of tiny, distinct, highly cohesive, loosely coupled, autonomously scalable, independently deployable, and distributed services \cite{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 that suggests “Gather together those functionalities that change for the same reason, and separate those that change for different reasons” \cite{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. These 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 totally 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. Since 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 several stages of SDLC. Use case diagrams abstract high-level view of 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 following relationships: Generalization, Include, and Extend [6] as shown in Figure 1.

In Generalization, a parent use case may be specialized to 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. Generalization is simply the inheritance relationship between two use cases by which 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.

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](image)

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. We use the use cases in the requirements phase for the greenfield development.

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 system1. 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

---

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

We have devised a five-step approach namely GreenMicro to identify microservices for greenfield developments based on clustering of 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 on which the clustering algorithm is applied to find candidate services. We have evaluated our approach on an in-house proprietary application and three sample benchmark applications. 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) Cargo Tracking System, ii) AcmeAir, and iii) JPetStore.

C4: Validating our results quantitatively.

Microservice implementation of some of the projects is available on Github (JPetStore\textsuperscript{2}, TFWA\textsuperscript{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] used dataflow-driven decomposition techniques for identifying microservices from the given detailed business 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 (DFD\textsubscript{DS}). DFD\textsubscript{DS} shows the relation between processes and related data stores. Next, a condensed DFD called decomposable DFD is taken out from DFD\textsubscript{DS} 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] proposed 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] proposed 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 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 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
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 re-engineer applications 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 model the dependency and connection between individual use cases elements in the system. These relationships add semantics to use case models by defining the 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 $UC^A = \{UC^1, UC^2, \ldots, UC^n\}$ where $UC^i$ represents 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, $UC$ and $UC$.

B. PROPOSED APPROACH

GreenMicro is a systematic approach for microservice identification comprising a sequence of steps. Sequence of steps is 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 $<\text{component - attribute}>$ 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, Update-U, Delete-D, and Read-R as cells of the matrix [18].

   - “C” means the use case CREATES the database entity.
   - “U” means the use case UPDATES the database entity.
   - “D” means the use case DELETES the database entity.
   - “R” means the use case READS the database entity.

2. Use Case-Use Case Relationship Matrix ($UC-UC$) - This matrix represents the degree of interdependence among all the use cases. Three typed of relationships 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].

3. 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 basically a measurement that quantifies the 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.

\[
\cos(x, y) = \frac{x \cdot y}{\|x\| \times \|y\|}
\]

where, \( x \cdot y \) = dot product of the vectors ‘x’ and ‘y’ and \( \|x\| \times \|y\| \) = cross product of the two vectors ‘x’ and ‘y’.

4. Generate combined Similarity Matrix - In this step, we create a combined Similarity Matrix of use cases which is a NxN 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.

5. 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\(^A\) 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 of a cluster.

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. 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, we find possible microservices \( \mu^A \) from the given set of use cases UC\(^A\).

FIGURE 2. Complete Outline of the GreenMicro Approach
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 Cargo Tracking System, AcmeAir and JPetStore and one proprietary 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. Table I presents the basic information about the four applications, like, number of use cases and number of database tables.

| 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] |

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 chatbot. 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. Thus the dimension of the UC-D matrix is 22x7. UC-UC matrix is a 22x22 matrix depicting relationships between use cases. Combined Similarity Matrix is also a 22x22 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:

**Cargo Tracking System**: 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.

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

**JPetStore**: An e-commerce shopping application that allows users to browse catalog, products and items, adding items to cart, removing items from cart, update cart items and purchase of items within several categories of pets.

### B. RELATED TECHNIQUES FOR QUANTITATIVE EVALUATION

The objective of our evaluation is to assess whether GreenMicro can produce 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** 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 attributes outlier classes that could be in the embeddings of other classes.

**Mono2Micro** 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 the control flow of the dynamic runtime traces convey the time dimension.

**MEM** makes use of Kruskal’s algorithm to find the 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** 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 offers the most appropriate microservice cuts.

**API Interface Analysis** 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** suggested a semi-automatic dataflow-driven microservice decomposition approach to generate a process-data store version of DFD (DFDPS). DFDPS shows the relation between processes and related data-stores. This DFDP 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 and capture dependencies for them. A clustering algorithm further applied to combine all extracted dependencies for identifying microservices.

---

4. https://github.com/citerus/dddssample-core
5. https://github.com/acmeair/acmeair
6. https://github.com/mybatis/jpetstore
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
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 applications for those quality assessment metrics which 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 below mentioned five quality metrics for JPetStore and AcmeAir applications to measure the effectiveness of partitions recommended using GreenMicro.

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

![Mathematical expression for SM](image)

where $scoh_i$ measures the structural cohesiveness of a microservice (intra-connectivity) and $scop_{i,j}$ 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 $scoh$, 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 partition $j$. 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 $BC_i$ 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

$IFN = \frac{1}{N} \sum_{j=1}^{N} \text{if}_{n_j}$

Non-Extreme Distribution (NED) [28], [36] measures how evenly distributed the sizes of the recommended microservice is. In general, it is preferred to have a microservice that has too many or too few classes.

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

where $n_i$ 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 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 depends 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 $Ce / (Ca + Ce)$.

Relational Cohesion (RC) - It is a measure of the number of internal relations that represent class inheritance, method

[12] http://eclipse.hello2morrow.com/doc/standalone/content/java_metrics.htm
invocations, access to class attributes etc. Higher values of relational cohesion suggest more cohesion.

V. RESULTS
We evaluated GreenMicro’s performance for three benchmark applications and one proprietary 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 four baselines on five evaluation metrics. Table II, III, IV and V present the comparison of our results for all four applications across two sets of metrics. For all the 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 II. GreenMicro’s better results in terms of BCP point 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 indicated 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 III. 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 a 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 IV.

For TFWA, we achieve three microservices namely Authentication, Feedback, and Analytics. Table V (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.

V. RESULTS
We evaluated GreenMicro’s performance for three benchmark applications and one proprietary 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 four baselines on five evaluation metrics. Table II, III, IV and V present the comparison of our results for all four applications across two sets of metrics. For all the 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 II. GreenMicro’s better results in terms of BCP point 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 indicated 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 III. 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 a 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 IV.

For TFWA, we achieve three microservices namely Authentication, Feedback, and Analytics. Table V (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.
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.

| TABLE VI | 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                                 |

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 proposed 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. 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.

**REFERENCES**

[1] I. K. Aksakallii, 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,” Proc. - 2017 IEEE 24th Int. Conf. Web Serv. ICWS 2017, pp. 524–531, 2017, 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., pp. 817–852, 2021.

[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 Proceedings of the International Workshop on Requirements Engineering for Product Lines, 2002, pp. 19–25.

[7] M. Kaiske, “Transforming monolithic architecture towards microservice architecture from a department of Computer Science,” p. 72, 2017.

[8] R. Chen, S. Li, and Z. Li, “From Monolith to Microservices: A Dataflow-Driven Approach,” Proc. - Asia-Pacific Softw. Eng. Conf. APSEC, vol. 2017-Decem, pp. 466–475, 2018, doi: 10.1109/APSEC.2017.53.

[9] S. Li et al., “A dataflow-driven approach to identifying microservices from monolithic applications,” J. Syst. Softw., vol. 157, p. 110380, 2019, doi: 10.1016/j.jss.2019.07.008.

[10] M. J. Amiri, “Object-Aware Identification of Microservices,” Proc. - 2018 IEEE Int. Conf. Serv. Comput. SCC 2018 - Part 2018 IEEE World Congr. Serv., pp. 253–256, 2018, doi: 10.1109/SCC.2018.00042.

[11] M. Ahmadavand and A. Ibrahim, “Requirements reconciliation for scalable and secure microservice (de)composition,” Proc. - 2016 IEEE 24th Int. Req. Eng. Conf. Work. REW 2016, no. September, pp. 68–73, 2017, doi: 10.1109/REW.2016.14.

[12] L. Baresi, M. Garriga, and A. De Renzi, “Microservices identification through interface analysis,” Lect. Notes Comput.
Ms. Deepali Bajaj is Associate Professor in 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 is currently doing her research in the area of Cloud and Distributed Computing. Her key research areas are Microservices and Function as a service (FaaS) of serverless technology. She has authored several national and international research publications. She has served as organizing secretary of national conferences NCAECA 2016.

Dr. Anita Goel is Professor in Department of Computer Science, Dyal Singh College, University of Delhi, India. She has received her Ph.D. in Computer Science and Masters in Computer Applications from Jamia Millia Islamia and Department of Computer Science (University of Delhi), respectively. She has a work experience of more than 30 years. She is a visiting faculty to Delhi Technological University and NIIT University. From 2009-10, she was Fellow in Computer Science, at Institute of Life Long Learning (ILLL) in University of Delhi. She has served as member of program committee of International conferences like IEEE BigData Congress 2015 and ICWI 2015. She has guided several students for their doctoral studies and has travelled internationally to present research papers. She has authored books in Computer Science and has several national and international research publications.

Dr. SC Gupta is B.Tech. (EE) from IIT Delhi and has worked at Computer Group at Tata Institute of Fundamental Research and NCSDCT (now C-DAC Mumbai), Till recently, he worked as Deputy Director General, Scientist-G and Head of Training at National Informatics Centre, New Delhi and 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. Currently he is a Visiting Faculty at Dept of Computer Science and Engineering, IIT Delhi. His research interests include Software Engineering, Data Bases and Cloud Computing. He has been teaching Cloud Computing at IIT Delhi, which includes emerging disruptive technologies like SDN and SDS. He has guided many M.Tech & PhD Research students in these technologies and has many publications in Software Engineering and Cloud Technology in National and International Conferences and Journals.