International Journal of Computer Science Engineering Techniques – Volume 9 Issue 5, September - October - 2025

SELECTING AN EFFECTIVE MICROSERVICES DECOMPOSITION APPROACH: A DECISION FRAMEWORK

*Md. Abdul Momin[1], M.M. Musharaf Hussain[2], and Md. Ezharul Islam[3]
[1]Department of Computer Science Engineering, Jahangirnagar University
[2]Department of Computer Science Engineering, Jahangirnagar University
[3]Department of Computer Science Engineering, Jahangirnagar University

Email: momin99cse@gmail.com

Abstract: This research presents a comprehensive exploration of diverse microservices decomposition techniques. This research identifies the sequential steps integral to each decomposition method through a meticulous study and analysis of multiple techniques. Moreover, the paper integrates insights gleaned from a select group of experts. These experts offer valuable perspectives on software characteristics and elucidate the types of example software ideally suited for distinct decomposition types. They also validate the time and cost implications associated with each decomposition technique. Drawing from these multifaceted insights, the paper culminates in creating an algorithm. This algorithm is intricately designed based on collective knowledge and discussions surrounding software traits, such as suitability, time, and cost considerations linked to various decomposition techniques. This algorithm helps developers choose the most effective decomposition approach for microservices.

Keywords: *Microservice, Monolithic, SOA, Decomposition, Domain Driven Design (DDD), AOP, DBSCAN*

1. INTRODUCTION

In this section, the authors describe the problem, the main question, and try to find the answer. Microservices are a service-oriented architecture pattern that breaks applications into small, independent service units, promoting efficient development and scalability. SOA and MSA are service-based architectures that break down systems into modular and self-contained services. Microservices focus on goals and replace ability, allowing new technologies to be tried in isolated services. Smaller components make applications easier to build and maintain. MSA enables independent service management. Loosely coupled software makes it easier to engage with open source. SOA emphasizes sharing, while MSA minimizes it. MSA prefers choreography but aims to reduce it due to the high coupling risks. SOA has an enterprise scope, while MSA focuses on application scope [1]. Monolithic, SOA, and Microservices are architectural approaches for large-scale applications. SOA and MSA are both service-based architectures that break down systems into smaller, more manageable services. It is suggested that MSA represents the future direction of service-based architectures, while SOA becomes a legacy architecture. MSA is more flexible and scalable than SOA, and it is better applicable to the ever-changing requirements of modern applications [2]. Microservices are smaller and more specialized than traditional SOA services. They're also more independent, making them easier to build, test, deploy, and maintain in the long run. They also work well for web-based systems that have clear, separate components. SOA and microservices are both architecturally service-based, with differences in strengths and weaknesses. SOA is better suited for large, complex enterprise systems, and microservices are appropriate for smaller, more agile applications [3]. Time and cost are two of the biggest challenges facing organizations that are considering adopting a microservices architecture.

Problem Statement: The problem Statement of this research is that in the era of microservices architecture, organizations face a critical challenge in determining the most effective approach for decomposing monolithic systems into microservices. The absence of a standardized method for selecting decomposition strategies often leads to ambiguity and suboptimal architectural choices. This research aims to address this gap by developing a comprehensive decision framework that assists organizations and architects in systematically evaluating and selecting the most suitable microservices decomposition approach. The framework will consider various factors various factor related to the decomposition of monolithic applications.

Research Objective: The research objectives of this study are to conduct a comprehensive examination and analysis of various decomposition approaches, delving into their respective implementation processes and methodologies. To assess which decomposition methods are suitable for different software examples based on their specific characteristics. Ultimately, choose a decomposition approach for transitioning from monolithic software to microservices that optimizes both time and cost-effectiveness.
The study will be conducted in three phases:

i. **Literature review**: The initial stage will encompass a thorough examination of current research on microservices architecture, focusing on the efficient breakdown of monolithic systems in terms of time and cost-effectiveness.

ii. **Experts' Opinion:** During the second phase, experts' perspectives will be sought regarding the transition from monolithic to microservices across multiple parameters.

iii. **Algorithm Design:** In the third phase, an algorithm will be crafted to select an effective decomposition method efficiently.

The findings of this study will contribute to the body of knowledge on microservices architecture and time and cost-effective implementation. The study will also provide practical guidance to organizations that are considering adopting a microservices architecture.

Research Question: Here are some specific research questions that the study will address:

RQ1: Select the effective microservices decomposition approaches for different types of applications.

RQ2: Develop a decision framework to select the most appropriate microservices decomposition approach for a specific application.

RQ3: A Way to evaluate the effectiveness of different microservices decomposition approaches.

In this study, a combination of quantitative and qualitative research methods is used. Quantitative methods will be used to collect data on challenges and risks associated with microservices decomposition. Qualitative methods will be used to increase a deeper understanding of these challenges and risks.

2. LITERATURE REVIEW

A critical step in microservices adoption is decomposing applications into cohesive services. A service, as defined by [4], must encapsulate business capabilities, a clear interface, and a contract, though specifics on categorization and ownership remain underexplored. While Bogner et al. [5] explore how traditional SOA patterns can be adapted for microservices, their study falls short in one key area it doesn't fully weigh the practical trade-offs, especially when it comes to implementation costs or time investments. This leaves practitioners without clear guidance on whether repurposing an SOA pattern is truly worth the effort in their specific context. Similarly, the authors of [6] provide a useful survey of migration patterns from SOA to microservices, but their focus remains narrowly technical. Lots of studies look at how microservices stack up against traditional monoliths or SOA architectures. For instance, [11] quantifies infrastructure costs, showing microservices can reduce cloud expenses by up to 70% compared to monoliths, while serverless (e.g., AWS Lambda) further cuts overhead. Performance evaluations are explored by [13] and [14], which compare response times, scalability, and resource utilization, revealing microservices' strengths in modularity but potential latency in distributed workflows. [15] Extend this to serverless architectures, highlighting context-dependent trade-offs in cloud environments. The debate between microservices and conventional SOA-based web services is dissected by [8], which evaluates scalability, flexibility, and maintainability. Their analysis includes real-world cases to illustrate practical implications, though it acknowledges that the "right" choice depends on organizational context. Complementing this, [7] provides a foundational review of microservices architecture, clarifying its role in modern software development and underscoring its advantages in agility and independent deploy ability. Transitioning to microservices involves technical and organizational hurdles. Studies like [5] and [6] do a good job explaining how to adapt patterns for microservices, but they miss the practical challenges - like how these systems actually scale in big organizations. Luckily, we can learn from real-world examples: [16] shows how Brazilian government agencies managed to reduce deployment risks and push updates faster, while [17] reveals how companies like Amazon, Netflix and Uber used microservices to become more scalable and agile. Rainyday Grocer [18] exemplifies cost-efficient scaling using CI/CD and DevOps, though balancing operational expenses remains a consideration. Innovative applications of microservices are explored in niche domains. For example, [10] combines blockchain with microservices to enhance security and transparency in public safety systems, while [12] optimizes AI microservices on edge devices by analyzing latency-cost trade-offs. Complexity metrics, as studied by [9], offer quantitative comparisons between SOA and microservices, assessing code and interaction intricacies to guide architectural decisions. Despite extensive research, gaps persist. Many studies ([5], [6], [13]) prioritize technical feasibility over socio-technical barriers (e.g., team structure, skill gaps). Cost-time trade-offs in migration ([5], [11]) and empirical validations of scalability ([16]) warrant deeper exploration. Future work could integrate organizational factors with technical metrics to offer holistic guidance for practitioners. Most of the research out there on microservices gets way too caught up in the technical details—how to chop up an app, optimize performance, all that nerdy stuff. But what about the real-world messiness? Nobody's talking about the hidden costs, the team drama, or whether switching to microservices is even a smart move for most companies. Sure, you'll find plenty of glossy case studies about Amazon and Netflix pulling it off flawlessly—but where are the stories about the migrations that blew up budgets, took years longer than planned, or just crashed and burned? It's like reading Yelp reviews where everyone's either five stars or radio silence—nobody's admitting when things went sideways. Others compare microservices to monoliths but only look at performance, not security or maintenance nightmares. Plus, there's barely any solid advice on how to actually break down big systems without causing chaos, and almost no one talks about doing it step by step instead of all at once. Bottom line? We need more practical, no-BS guidance that covers not just code, but people, money, and risks—because right now, it's too easy for companies to jump in and regret it later.

3. RESEARCH METHODOLOGY

In this section, we examine various decomposition approaches tailored to specific characteristics and types of monolithic software. Additionally, we outline the steps involved in decomposing monolithic software, including estimated timeframes and costs based on expert opinions. All of these are detailed using three tables. Effectively migrating to a microservice from a monolithic application is crucial for adopting a microservice architecture. A well-structured decomposing method can facilitate this process. This study develops a comprehensive procedure for effective microservice decomposition. Various factors such as application architecture, business requirements, and technical limitations are considered here. The proposed methodology will be evaluated through real-world case studies to assess its effectiveness in practical scenarios. The findings of this research will provide valuable insights for organizations considering microservice adoption and guide their decomposition efforts. This research will contribute to the advancement of microservice architecture research by introducing a systematic and practical approach to microservice decomposition. Our methodology encompasses three distinct phases. In the initial phase, we identify and analyze various decomposition processes, meticulously outlining their decomposition steps. Subsequently, we conduct a comprehensive survey and engage in insightful interviews with a diverse group of software professionals to carefully select the most suitable decomposition process for a specific set of software characteristics and types. Additionally, we consider the time and cost implications associated with each decomposition process step. Finally, we meticulously select the most appropriate decomposition process tailored to the specific requirements.

In our preliminary stage, we examined 33 distinct approaches to the decomposition process, delineating their steps. These various approaches are cataloged in Table 1, while the specific steps associated with each approach are elaborated upon in Table 2**.**

TABLE 1. Decomposition Type

#Decomposition Type	Description
1: Decomposition by software component	This approach focuses on identifying cohesive, self-contained components and transforming them into individual microservices. Each microservice encapsulates a specific software component's functionality and operates

	independently [19].
2: Decomposition by Software event log collection	A decomposition technique for monolithic applications that is based on process mining. Process mining is a technique for extracting knowledge from event logs, which are records of the activities that have taken place in a system [20].
3: Decomposition by business functionalities	Decomposition by business functionalities, also known as functional decomposition, is a critical step in transitioning from a monolithic architecture to a microservices architecture. This process involves breaking down a large, monolithic application into smaller, independently deployable microservices that are organized around specific business functionalities or capabilities [21].
4: Decomposition by Analysis involving both static and dynamic aspects.	This decomposition process considers both the static, structural aspects of the application (such as codebase, data schema, and dependencies) and the dynamic aspects (such as runtime behavior and interactions) [22].
5: Decomposition based on business capabilities	This approach involves identifying and isolating distinct business capabilities within the monolith and then creating microservices around them [23].
6: Decomposition by Domain-driven design (DDD)	Domain-driven design is an approach that focuses on modeling a system's domain (its core business logic) and breaking it down into bounded contexts, aggregates, and entities. When transitioning from a monolithic architecture to microservices [24].
7: Decomposition by API-first development	An API-first development approach is another strategy for achieving a more modular and scalable architecture. API-first development focuses on designing and implementing well-defined APIs as a foundation for building and integrating services [24].
8: Decomposition by Technical debt	Decomposing a monolithic application into microservices while addressing technical debt is a challenging but essential process for ensuring the long-term maintainability and scalability of your system. Technical debt refers to the accumulation of suboptimal or hasty technical decisions made during the development of your monolithic application [25].
9: Data-oriented decomposition	Data-oriented decomposition is an approach to transitioning from a monolithic architecture to a microservices architecture that focuses on breaking down the monolithic data store and its associated dependencies into smaller, more manageable data services [25].
10: Process Oriented decomposition	Process-oriented decomposition is an approach to transitioning from a monolithic architecture to a microservices architecture that focuses on breaking down the monolithic application's business processes into smaller, more manageable microservices [25].
11: User-oriented decomposition	User-oriented decomposition is an approach to transitioning from a monolithic architecture to a microservices architecture that focuses on breaking down the monolithic application's user-facing functionality into smaller, more specialized microservices [25].
12: Technology-oriented decomposition	A technology-oriented decomposition is an approach to transitioning from a monolithic architecture to a microservices architecture that focuses on breaking down the monolithic application based on the underlying technologies or technical components it uses [25].
13: hierarchical DBSCAN algorithm	Hierarchical DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an extension of the traditional DBSCAN clustering algorithm that hierarchically organizes clusters [26].
14: Extensible Multiple Strategy Tool approach	Creating an extensible multiple-strategy tool for decomposing a monolithic application into microservices involves developing a flexible software system that can accommodate various strategies and methods for decomposition [27].
15: decomposition based on the application's development history	Analyzing the development history of a monolithic application can provide valuable insights for identifying potential candidates for microservices decomposition. By examining the historical context, code changes, and architectural decisions, you can make informed decisions about which parts of the monolith should be extracted into microservices [28].

16: By Keyword Extraction and BFS Combination Method to Cluster Monolithic Classes	Decomposing a monolithic application's classes into clusters of microservices using a combination of keyword extraction and breadth-first search (BFS) is a complex but powerful approach. This method involves extracting keywords from class names and code comments to identify related classes and then using BFS to create clusters based on those relationships [29].
17: A Classification of Refactoring Approaches	Refactoring approaches for transitioning from a monolithic architecture to a microservices architecture can be classified into several categories based on their focus and objectives [30].
18: Decomposition using Distributed Representation of Source Code	Decomposing a monolithic application into microservices using the distributed representation of source code involves breaking down the large, monolithic codebase into smaller, more manageable services that can operate independently. This is a complex and multifaceted process that requires careful planning and execution [31].
19: Decomposition by Aspect-Oriented Programming	Aspect-oriented programming (AOP) is a programming paradigm that allows you to modularize cross-cutting concerns, such as logging, security, and transactions, which often span multiple modules in a monolithic application [32].
20: Based on a semi-automatic approach	A semi-automatic approach to decomposing a monolithic application into microservices combines human expertise with automated tools and processes [33].
21: Decomposition by an extensible multiple strategy tool, called Mono2Micro	The general idea behind such tools (Mono2Micro) is to automate the process of identifying potential microservices, determining their boundaries, and assisting with the extraction and refactoring of code [34].
22: Decomposition by MicroValid: A validation framework	MicroValid is a validation framework for automatically decomposed microservices from monolithic applications. It is a tool that can help developers to assess the quality of their microservices and to identify any potential problems before they go into production. MicroValid uses a variety of static analysis techniques to assess the quality of microservices [35].
23: Decomposition based on Problem Frames	Decomposition based on Problem Frames is a systematic approach to decomposing monolithic applications into microservices. It is based on the idea that a monolithic application can be viewed as a collection of problem frames, each of which represents a distinct business or technical problem [36].
24: Dependencies-based decomposition method	Decomposition based on Problem Frames (PFs) is a systematic approach to decomposing monolithic applications into microservices. It is based on the idea that a monolithic application can be viewed as a collection of PFs, each of which represents a distinct business or technical problem [37].
25: Decomposition based on topic modeling	Decomposition based on topic modeling is a relatively new approach to decomposing monolithic applications into microservices. It is based on the idea that the different parts of a monolithic application can be identified and grouped together based on the topics to which they are related [38].
26: Decomposition uses evolutionary search	It is a type of optimization algorithm that copies the process of natural selection and finds optimal solutions to problems. It is often used in software engineering to find optimal designs, architectures, and configurations. Evolutionary search, often associated with genetic algorithms, can be applied to find the best-fit microservice boundaries over time [39].
27: Decomposition using code vectorization and sequence of accesses	Decomposition using code vectorization and a sequence of accesses from monolithic to microservice is a promising approach for automating the decomposition of monolithic applications into microservices. Decomposition using code vectorization and sequence of accesses is a promising approach for automating the decomposition of monolithic applications into microservices [40].
28: Decomposition using Quality Driven Framework	A Quality Driven Framework for Decomposing Legacy Monolith Applications to Microservice Architecture proposes a quality-driven framework for decomposing monolithic applications into microservices. The framework is designed to help organizations decompose their monolithic

	applications into microservices in a way that preserves quality [41].
29: Decomposition Applying Microservice Refactoring	This is a promising approach for refactoring legacy object-oriented systems to microservices. It is based on some principles and has been evaluated on a real-world system. Refactoring is a complex process, so no single approach fits all systems. A specific approach will be used depending on the specific characteristics of the legacy system [42].
30: Break down a Monolithic based on cohesion, coupling, and size.	Breaking down a monolithic application into microservices based on cohesion, coupling, and size is a sound approach to achieving a well-structured and maintainable microservices architecture [43].
31: Decomposition by detection of service boundary	The technique for automating the detection of service boundaries is a promising approach for breaking down monolithic applications into microservices. It is based on sound principles and has been evaluated on a number of real-world applications. The technique has been evaluated on a number of real-world monolithic applications and is effective in detecting service boundaries [44].
32: decomposing ORM-based monolithic web applications	Decomposing ORM (Object-Relational Mapping)-based monolithic web applications into microservices is a complex process that requires careful planning and execution. Decomposing ORM-based monolithic web applications from monolithic to microservices can be a challenging task, but it can offer a number of benefits, such as improved scalability, reliability, and maintainability [45].
33: Decomposition based on the Architecture of the Monolithic system	A process for converting a monolithic application to a microservices-based architecture. Converting a monolithic application to a microservices-based architecture can be a complex task, but it can offer a number of benefits, such as improved scalability, reliability, and maintainability [46].

In the subsequent stage, we aimed to extract these motivations by carrying out an empirical study through a survey. This study involved interviewing 16 practitioners who have embraced a microservices-based architectural style for a minimum of two years. As for the profiles of our interviewees, they exhibit differences across multiple aspects. In terms of their roles within their companies, our participants consisted of 30% software architects, 25% project managers, 25% senior developers, 10% agile coaches, and 10% company CEOs. Each interviewee possessed a minimum of five years of experience in software development, including the CEOs. In terms of the organizational domain, our interviewees were distributed as follows: 28.57% in banking, another 28.57% in companies exclusively creating and selling their software as a service (such as website builders, mobile app generators, and similar services), 23.81% in consultancies focused on microservices migration, 9.52% in public administration IT departments, and the remaining 9.52% in telecommunication companies. Table 2 serves as a comprehensive summary curated by expert professionals, detailing the inherent characteristics of software alongside specific examples that aptly correspond to distinct decomposition types. This collation encapsulates the collective insights and expertise of these professionals, showcasing how different software attributes harmonize with particular approaches to decomposition.

TABLE 2. Monolithic Software Characteristics and Type (Described in Appendix A)

Once more, we've outlined various steps for decomposing a monolithic system into microservices based on different decomposition types. A crucial initial step for all types involves comprehending the monolithic system itself. The effort and expenses involved in understanding this system depend on its scale, complexity, and the expertise of those involved. Table 3 presents the decomposition steps derived from the research paper listed in Table 1. Additionally, we've included time and cost estimations based on our expertise. However, these estimations are ideal scenarios and may differ depending on the size and complexity of the monolithic system.

TABLE 3. Steps applied and time, and cost requirements of the decomposition process (Described in Appendix B)

After analyzing the three aforementioned tables, we identified various decomposition types and their corresponding software characteristics. Additionally, we pinpointed specific software types best aligned with particular decomposition methods. Drawing insights from research papers and expert opinions, we inferred the time and cost involved in breaking down monolithic software into microservices. Consequently, it's now imperative to introduce an algorithm outlining the process of selecting suitable decomposition types for a monolithic system.

4. PROPOSED ALGORITHM AND EVALUATION

To decompose a monolithic system into microservices here we design an algorithm. Decomposing means breaking down a large system into several smaller ones. Smaller systems enhance scalability, agility, and easy maintenance. With the following steps large system can be effectively analyzed. After analyzing based on time and cost can identify potential decomposition types. Finally, a microservice is generated.

4.1 Proposed Algorithm

Step 1: Find the features of monolithic software as well as its corresponding types.

Step 2: Go through Table 2 iteratively and attempt to correlate its contents with characteristics and types. When a match is found, categorize and save it as a potential decomposition type.

Step 3: If a single match is identified, classify it as a decomposition type. Should no match be found, return to step 2 to identify closely related characteristics or types. When multiple features or types align, proceed to step 4.

Step 4: Using the decomposition types selected in Step 3, assess the time and cost details from Table 3 related to these choices to conclusively determine the viable decomposition types.

Step 5: Generate microservices from the monolithic system by applying the chosen decomposition types.

4.2 Formal Representation of the Algorithm

This algorithm defines a strategy for identifying a suitable decomposition approach from monolithic to microservice

Input:

- MonolithicSoftware: A monolithic application description, with features and types.

Output:

- DecompositionTypes: Suitable microservice decomposition strategies for the provided MonolithicSoftware.

Algorithm:

1. **Identify Monolithic Features and Types:**
    - Function: GetMonolithicDetails (MonolithicSoftware)
        - Analyzes MonolithicSoftware to extract its features (Features) and types (Types).
    - Return value: (Features, Types)
2. **Iterate Through Decomposition Table:**
    - Function: FindPotentialDecompositions (Features, Types, DecompositionTable)
        - DecompositionTable is a data structure containing information about decomposition types.
        - Iterates through each decomposition type (DecompositionType) in DecompositionTable.
            - Compares the features and types of DecompositionType and monolithic applications.
            - If completely match, then add DecompositionType to a list of potential decompositions (PotentialDecompositions).
            - If a partial match (closely related features or types) is found, add DecompositionType to a separate list for further consideration (PartialMatches).

3. **Select Decomposition Type:**
    - Function: ChooseViableDecomposition (PotentialDecompositions, PartialMatches, CostTimeTable)
        - CostTimeTable is a data structure containing information about the cost and time associated with each decomposition type.
            - If PotentialDecompositions has a single entry (DecompositionType), return it as the chosen decomposition.
            - Otherwise, if PotentialDecompositions is empty and PartialMatches is not empty:
                - Iterate through each DecompositionType in PartialMatches.
                - Use CostTimeTable to retrieve the estimated cost and time for each step involved in DecompositionType.
                - Based on cost and time constraints, select a single DecompositionType as the chosen decomposition and return it.
            - If PotentialDecompositions and PartialMatches are empty, return "No suitable decomposition found."
4. **Generate Microservices:**
    - Function: GenerateMicroservices (MonolithicSoftware, DecompositionType)
        - Uses the chosen DecompositionType to guide breaking down the MonolithicSoftware into individual microservices.
5. **Output:**
    - The algorithm outputs the DecompositionTypes returned by ChooseViableDecomposition or "No suitable decomposition found" if no viable option exists.

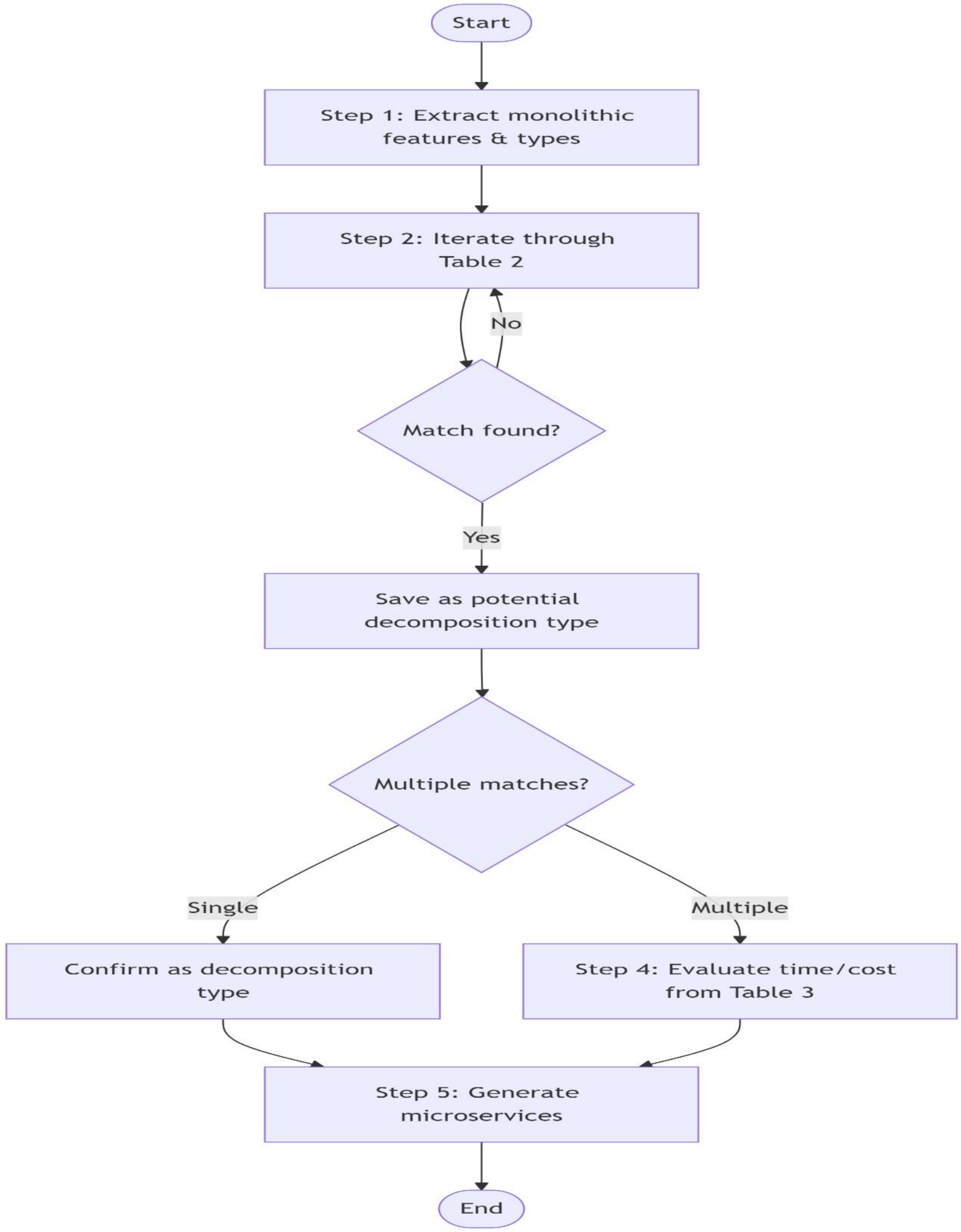

Fig. 1. Flowchart of the algorithm

4.3 Algorithm Evaluation and Complexity

The proposed algorithm offers a structured approach to breaking down a monolithic system into microservices. Its effectiveness hinges on accurate initial analysis in Step 1 and the comprehensiveness of information in Tables 2 and 3. Success also relies on the implementer's expertise, particularly in assessing time and cost details in Step 4, and on the algorithm's accuracy in identifying relevant features and decomposition choices. Scalability depends on input data complexity and clarity, as well as decision-making efficiency. With organization, documentation, and automation, the algorithm can handle larger systems. Optimality is influenced by initial analysis accuracy, input data relevance, and decision-making effectiveness. Continuous refinement improves optimality. Robustness is enhanced through thorough feature identification, clear input data, and flexible decision-making. Overall, the algorithm efficiently selects decomposition techniques, as proven mathematically. The algorithm's performance depends heavily on system size. In the worst case, the time it takes grows quadratically ($O(M \times N)$) - meaning if you double the number of features (M) and rules (N), the runtime could quadruple. This makes it less ideal for massive systems with ens of thousands of features. Memory usage grows linearly ($O(M+N+P+S)$), where P includes extra processing data and S is your final microservice count, though you'll need substantial temporary storage. Generating the actual microservices (Step 5) is efficient ($O(S)$), but complex system dependencies can create unexpected slowdowns. For larger implementations, we recommend speed boosts like parallel processing or machine learning shortcuts, though the sweet spot remains medium-sized systems where the approach works most efficiently.

4.4 Recommendation and Discussion

Migrating to microservices from monolithic decomposition is a crucial part. It enhances scalability by scaling each service to scale independently. Isolation is also provided by microservices, which limit the failures or issues within specific services, minimizing a full system down. This approach encirclements technology diversity, which encourages suitable technology for each service to optimize performance and development cost. It also enhances updates, maintenance, and bug fixing for individual services without impacting the entire system. By decomposing into several services, parallel work can be done. Different teams work on different services that increase adaptability, innovation with specific services, reduce overall complexity, and improve the system's comprehensibility, debugging, and extensibility. So, the decomposition phase is important, and it is the foundation for a more modular, scalable, and adaptable system. To gain this advantage, it requires careful planning, dependency consideration, data management, and full architecture to confirm successful migration of the system

4.5 Limitations and Future Directions

This research helps developers break monoliths into microservices, but it has limits. The algorithm gets slow with very large systems. Matching features manually can be subjective. Time and cost estimates are helpful, but real projects may differ. It also focuses mostly on technical issues, not team or skill challenges. Future improvements could make it better. Machine learning could automate feature matching. The framework should address team and process problems too. More real-world testing would make it more practical. Combining code analysis with runtime data might improve results. These changes could make the framework even more useful for developers.

5. CONCLUSION

Within this research paper, our focus revolves around an in-depth analysis of various decomposition processes pivotal for the transition from a monolithic architecture to microservices. This approach aims to determine the most fitting decomposition process for this migration. To ensure a clear perspective, we engage multiple software experts, tapping into their expertise. As a conclusion of this collaborative effort, we meticulously designed an algorithm to select the most suitable decomposition method. This research paper makes a strong contribution by offering a thorough analysis, consulting with experts, developing a practical algorithm, and focusing on efficiency. Its impact extends beyond academic discourse, aiming to provide actionable insights for practitioners in the field of software architecture transition. The research paper focuses on numerous decomposition approaches, acknowledging that it doesn't encompass all existing types. Additionally, it recognizes the vast spectrum of decomposition systems available. The paper assesses software characteristics based on input from various experts. It suggests the potential for involving additional experts to enhance result accuracy, providing a direction for further research. Furthermore, the inclusion of more software professionals could refine time and cost estimations, especially when supplemented by practical examples.

References

[1] L. Rushani and F. Halili, “Differences between service-oriented architecture and microservices architecture,” Int. J. Natural Sciences: Current and Future Research Trends (IJNSCFRT), vol. 13, no. 1, pp. 30-48, 2022.

[2] T. Cerny, M. J. Donahoo, and J. Pechanec, "Disambiguation and comparison of SOA, microservices and self-contained systems," in Proc. Int. Conf. Research in Adaptive and Convergent Systems (RACS), 2017.

[3] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: a systematic mapping study," in Proc. 8th Int. Conf. Cloud Computing and Services Science (CLOSER), Funchal, Portugal, Mar. 2018, pp. —.

[4] M. Richards, Microservices vs. service-oriented architecture. Sebastopol, CA: O'Reilly Media, 2015, pp. 22-24.

[5] J. Bogner, A. Zimmermann, and S. Wagner, "Analyzing the relevance of SOA patterns for microservice-based systems," in ZEUS 2018: 10th Central European Workshop on Services and their Composition, Dresden, Germany, Feb. 2018, CEUR Workshop Proc., vol. 2072, pp. 9-16.

[6] V. Raj and R. Sadam, "Patterns for migration of SOA based applications to microservices architecture," J. Web Eng., vol. 20, no. 5, pp. 1229-1246, 2021.

[7] D. Shadija, M. Rezai, and R. Hill, "Towards an understanding of microservices," in Proc. 23rd Int. Conf. Automation and Computing (ICAC), Huddersfield, UK, Sept. 2017, pp. 1-6.

[8] V. Raj and S. Ravichandra, "Microservices: A perfect SOA-based solution for enterprise applications compared to web services," in Proc. 3rd IEEE Int. Conf. Recent Trends in Electronics, Information & Communication Technology (RTEICT), May 2018, pp. 1531-1536.

[9] V. Raj and R. Sadam, "Evaluation of SOA-based web services and microservices architecture using complexity metrics," SN Comput. Sci., vol. 2, pp. 1-10, 2021.

[10] R. Xu, S. Y. Nikouei, Y. Chen, E. Blasch, and A. Aved, "Blendmas: A blockchain-enabled decentralized microservices architecture for smart public safety," in Proc. IEEE Int. Conf. Blockchain, July 2019, pp. 564-571.

[11] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures," Service Oriented Comput. Appl., vol. 11, pp. 233-247, 2017.

[12] C. Wu, Q. Peng, Y. Xia, Y. Jin, and Z. Hu, "Towards cost-effective and robust AI microservice deployment in edge computing environments," Future Gener. Comput. Syst., vol. 141, pp. 129-142, 2023.

[13] F. Tapia, M. Á. Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, "From monolithic systems to microservices: A comparative study of performance," Appl. Sci., vol. 10, no. 17, p. 5797, 2020.

[14] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," IEEE Access, vol. 10, pp. 20357-20374, 2022.

[15] C. F. Fan, A. Jindal, and M. Gerndt, "Microservices vs serverless: A performance comparison on a cloud-native web application," in Proc. CLOSER, 2020, pp. 204-215.

[16] W. Luz, E. Agilar, M. C. de Oliveira, C. E. R. de Melo, G. Pinto, and R. Bonifácio, "An experience report on the adoption of microservices in three Brazilian government institutions," in Proc. XXXII Brazilian Symp. Software Eng., Sept. 2018, pp. 32-41.

[17] TS2.Space, "Microservices case studies: Success stories from leading companies," Available: https://ts2.space/en/microservices-case-studies-success-stories-from-leading-companies/. [Accessed: Jul. 30, 2024].

[18] The Open Group, "Microservices architecture working paper," Available: https://www.opengroup.org/soa/source-book/msawp/p5.htm. [Accessed: Jul. 25, 2024].

[19] D. Kuryazov, D. Jabborov, and B. Khujamuratov, "Towards decomposing monolithic applications into microservices," in Proc. 14th IEEE Int. Conf. Application of Information and Communication Technologies (AICT), Oct. 2020, pp. 1-4.

[20] D. Taibi and K. Systä, "From monolithic systems to microservices: A decomposition framework based on process mining," 2019.

[21] J. Kazanavičius and D. Mažeika, "Migrating legacy software to microservices architecture," in Proc. Open Conf. Electrical, Electronic and Information Sciences (eStream), Apr. 2019, pp. 1-5.

[22] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger, "Microservice decomposition via static and dynamic analysis of the monolith," in Proc. IEEE Int. Conf. Software Architecture Companion (ICSA-C), Mar. 2020, pp. 9-16.

[23] J. Kazanavičius and D. Mažeika, "Analysis of legacy monolithic software decomposition into microservices," 2020.

[24] N. Ivanov and A. Tasheva, "A hot decomposition procedure: Operational monolith system to microservices," in Proc. Int. Conf. Automatics and Informatics (ICAI), Sept. 2021, pp. 182-187.

[25] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in Proc. DEVOPS 2018, Chateau de Villebrumier, France, Mar. 2018, Revised Selected Papers 1, Springer, pp. 128-141.

[26] K. Sellami, M. A. Saied, and A. Ouni, "A hierarchical DBSCAN method for extracting microservices from monolithic applications," in Proc. 26th Int. Conf. Evaluation and Assessment in Software Engineering (EASE), June 2022, pp. 201-210.

[27] T. Lopes and A. R. Silva, "Monolith microservices identification: Towards an extensible multiple strategy tool," in Proc. 20th IEEE Int. Conf. Software Architecture Companion (ICSA-C), Mar. 2023, pp. 111-115.

[28] J. Lourenço and A. R. Silva, "Monolith development history for microservices identification: A comparative analysis," arXiv preprint arXiv:2212.11656, 2022.

[29] S. Rochimah and B. Nuralamsyah, "Decomposing monolithic to microservices: Keyword extraction and BFS combination method to cluster monolithic's classes," J. RESTI, vol. 7, no. 2, pp. 263-270, 2023.

[30] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in Proc. DEVOPS 2018, Chateau de Villebrumier, France, Mar. 2018, Revised Selected Papers 1, Springer, pp. 128-141.

[31] O. Al-Debagy and P. Martinek, "A microservice decomposition method through using distributed representation of source code," Scalable Comput.: Pract. Exp., vol. 22, no. 1, pp. 39-52, 2021.

[32] A. F. A. Freire, A. F. Sampaio, L. H. L. Carvalho, O. Medeiros, and N. C. Mendonça, "Migrating production monolithic systems to microservices using aspect-oriented programming," Softw.: Pract. Exp., vol. 51, no. 6, pp. 1280-1307, 2021.

[33] A. Selmadji, A. D. Seriai, H. L. Bouziane, R. O. Mahamane, P. Zaragoza, and C. Dony, "From monolithic architecture style to microservice one based on a semi-automatic approach," in Proc. IEEE Int. Conf. Software Architecture (ICSA), Mar. 2020, pp. 157-168.

[34] T. Lopes and A. R. Silva, "Monolith microservices identification: Towards an extensible multiple strategy tool," in Proc. 20th IEEE Int. Conf. Software Architecture Companion (ICSA-C), Mar. 2023, pp. 111-115.

[35] M. Cojocaru, A. Uta, and A. M. Oprescu, "MicroValid: A validation framework for automatically decomposed microservices," in Proc. IEEE Int. Conf. Cloud Computing Technology and Science (CloudCom), Dec. 2019, pp. 78-86.

[36] Z. Li, Y. Bo, and H. Xiao, "PF4Microservices: A decomposition scheme for microservices based on problem frames," arXiv preprint arXiv:2207.04586, 2022.

[37] O. Al-Debagy and P. Martinek, "Dependencies-based microservices decomposition method," Int. J. Comput. Appl., vol. 44, no. 9, pp. 814-821, 2022.

[38] M. Brito, J. Cunha, and J. Saraiva, "Identification of microservices from monolithic applications through topic modelling," in Proc. 36th Annu. ACM Symp. Applied Computing, Mar. 2021, pp. 1409-1418.

[39] K. Sellami, A. Ouni, M. A. Saied, S. Bouktif, and M. W. Mkaouer, "Improving microservices extraction using evolutionary search," Inf. Softw. Technol., vol. 151, p. 106996, 2022.

[40] V. Faria and A. R. Silva, "Code vectorization and sequence of accesses strategies for monolith microservices identification," in Proc. Int. Conf. Web Eng., Cham, Switzerland: Springer, June 2023, pp. 19-33.

[41] M. H. Hasan, M. H. Osman, N. I. Admodisastro, and M. S. Muhammad, "A quality-driven framework for decomposing legacy monolith applications to microservice architecture," 2023.

[42] J. Zhao and K. Zhao, "Applying microservice refactoring to object-oriented legacy system," in Proc. 8th IEEE Int. Conf. Dependable Systems and Their Applications (DSA), Aug. 2021, pp. 467-473.

[43] S. T. Ali, J. Long, V. K. Khatri, and M. A. Khuhro, "An approach to break down a monolithic app into microservices," —.

[44] R. X. C. de Jesus, "From monoliths to microservices: automating service boundary detection," 2021.

[45] F. Freitas, A. Ferreira, and J. Cunha, "A methodology for refactoring ORM-based monolithic web applications into microservices," J. Comput. Lang., vol. 75, p. 101205, 2023.

[46] T. C. K. Arachchi, "Process of conversion of monolithic application to microservices-based architecture," Ph.D. dissertation, 2021.

Appendix A

SL#	Monolithic Software Characteristics	Monolithic Software Example
1	Modular Architecture. Component-Based Design Clear Component Boundaries Component Reusability Improved Maintainability Component Ownership	E-commerce applications Content management systems Customer relationship management (CRM) systems Enterprise resource planning (ERP) systems Social networking applications Online gaming applications
2	Event-Driven Architecture Complex Event Processing Audit and Compliance Needs Real-time Data Processing Distributed System Needs	E-commerce platforms Customer relationship management (CRM) systems Enterprise resource planning (ERP) systems. Financial trading systems.
3	Well-Defined Business Domains Complex and Diverse Functionality Isolated Business Logic Independent Business Processes	E-commerce applications Customer relationship management (CRM) systems Enterprise resource planning (ERP) systems Banking and financial applications
4	High Complexity Performance Bottlenecks Resource Intensive High Dependency Redundancy and Duplication	Social networking applications Online gaming applications Banking and financial applications Healthcare applications Manufacturing applications
5	Business Domain Complexity Well-Defined Business Capabilities Independent Business Units Scalability and Performance Isolation Clear Service Boundaries	E-commerce platforms Online banking systems Airline reservation systems Social media platforms Content management systems
6	Well-defined domains Modular structure Clear domain boundaries Domain expertise Team collaboration	Large-scale E-commerce Platforms Financial Systems Enterprise Resource Planning (ERP) Systems Healthcare Information Systems Supply Chain Management Systems
7	Well-Defined Functional Modules Separation of Concerns Third-Party Integrations Integration Points Consumed by multiple channels	E-commerce platforms Online banking systems Content management systems Enterprise resource planning (ERP) systems Customer relationship management (CRM) systems
8	High Technical Debt Outdated Technologies Complex Legacy Code High Technical Debt Impact Lack of Documentation Regulatory Compliance	Enterprise Legacy Systems E-commerce Platforms Content Management Systems (CMS) Financial Systems Healthcare Information Systems Government Software
9	Data-Intensive Applications Complex Data Models High Data Volume Data Integration Data Processing Workflows Isolation of Data Access Data Security and Compliance	Real-time analytics systems Machine learning systems Fraud detection systems Financial trading systems Reservation systems Order management systems Logistics and Supply Chain Management Systems
10	Complex Business Workflows Modular Business Logic Clearly Defined Business Processes Highly Interconnected Components Isolation of Critical Processes	Manufacturing Execution Systems (MES) Workflow Management Systems Project Management Software Supply Chain Management Systems Customer Service Ticketing Systems

11	Has well-defined user roles Has complex user interfaces Diverse user requirements Evolving user experiences Cross-functional teams Integration with third-party systems User-driven features	Banking applications Content management systems (CMS) Customer Relationship Management (CRM) Healthcare Information Systems Human Resources Management Systems (HRMS) Booking and Reservation Systems E-learning and Education Systems
12	Complex and Tightly Coupled Components Technology Stack Migration Built using different technologies Has well-defined technical layers Has complex technical dependencies	Legacy Enterprise Systems Systems with Multiple Integration Points Monolithic Web Applications Complex Data Processing Systems Systems with Multiple Communication Protocols
13	Complex Interactions and Dependencies Non-Trivial Functional Separation Data-Intensive Systems Need for Data-Driven Decomposition Gradual Modernization	Geospatial Information Systems (GIS) Location-based Services Environmental Monitoring Systems Infrastructure Planning Systems Logistics and Supply Chain Management
14	Diverse Functionality Parallel Development Selective Modernization Gradual Transition to Microservices Has well-defined business capabilities	Financial Software Business Intelligence and Analytics Tools Predictive Maintenance Systems Simulation and Modeling Software E-commerce Platforms with Varied Sales Strategies
15	Applications with Modular Features Applications with Frequent Updates Applications with High Availability Requirements Applications with Multiple Technology Stacks Applications with Diverse User Bases Well-documented development history Clear separation of concerns	Legacy systems E-commerce application Banking application Healthcare application Enterprise Systems
16	Lack of Clear Modular Structure Keyword-Rich Codebase Interconnected Classes Limited Prior Documentation Incremental Decomposition Complex Business Logic Well-defined class structure Strong relationships between classes No external dependencies Complex and Large Codebases Lack of Clear Separation of Concerns Heterogeneous Functionalities Poor Code Documentation Codebase is Not Easily Refactored	E-commerce platforms Content management systems (CMS) Enterprise resource planning (ERP) systems Customer relationship management (CRM) systems Order management systems (OMS) Inventory management systems (IMS) Billing and payment systems Fraud detection systems Product recommendation systems Content delivery networks (CDNs) API management systems Integration platforms as a service (iPaaS) Business intelligence (BI) systems Machine learning (ML) systems
17	Monoliths with Poor Code Quality Systems with Clear Functional Boundaries Systems with Frequent Updates Interconnected Modules Complex Data Models High Availability and Fault Tolerance	Order management systems (OMS) Inventory management systems (IMS) Billing and payment systems Fraud detection systems Product recommendation systems Content delivery networks (CDNs)
18	Codebases with Unclear Modularization Applications with Diverse Functionality Applications with Multiple Modules Applications with Limited Documentation Codebases with High Technical Debt Applications with Evolving Business	E-commerce platforms Content management systems (CMS) Enterprise resource planning (ERP) systems Customer relationship management (CRM) systems Order management systems (OMS) Inventory management systems (IMS)

	Requirements Systems with Extensive Interdependencies No external dependencies	Billing and payment systems Content delivery networks (CDNs)
19	Monolithic Systems with Cross-Cutting Concerns Security and Compliance Requirements Applications with Frequent Updates Interconnected Modules Reducing Code Duplication	Enterprise resource planning (ERP) systems Customer relationship management (CRM) systems Content management systems (CMS) E-commerce platforms Real-time systems
20	Applications with Diverse User Bases Applications with Evolving Business Requirements Applications with Poor Maintainability Applications with Performance Challenges	Enterprise resource planning (ERP) systems Content delivery networks (CDNs) Integration platforms as a service (iPaaS) Machine learning (ML) systems
21	Applications with Poor Scalability Diverse Functionalities Complex Business Logic Frequent Updates and Changes Applications with Poor Maintainability Resource-Intensive Operations	Content management systems (CMS) Enterprise resource planning (ERP) systems Order management systems (OMS) Inventory management systems (IMS) Billing and payment systems Fraud detection systems
22	Diverse Functionalities Complex Business Logic Frequent Updates and Changes Resource-Intensive Operations	Enterprise resource planning (ERP) systems E-commerce platforms Content management systems (CMS) Social media platforms
23	Multi-Functional Applications Resource-Intensive Operations Applications with Poor Modularity Custom Business Applications Safety-Critical Systems Tightly Coupled Modules	Enterprise resource planning (ERP) systems E-commerce platforms Content management systems (CMS) Social media platforms Enterprise software Embedded systems
24	Well-defined class structure Strong relationships between classes No external dependencies Multi-Functional Applications Applications with Scalability Challenges Complex Business Logic Applications with Poor Modularity	Web applications Desktop applications Microservices Large Enterprise Applications Legacy Systems Custom Business Applications Safety-Critical Systems
25	The large corpus of text Well-defined topic structure No external dependencies Large codebase Well-defined domain Heterogeneous functionality Incremental refactoring	Enterprise Resource Planning (ERP) Systems Content Management Systems (CMS) E-commerce Platforms Healthcare Information Systems Banking and Financial Systems Manufacturing and Supply Chain Systems Customer Relationship Management (CRM) Software
26	Complex dependencies Well-defined fitness function Limited Technology Stack Unclear boundaries Evolving requirements Incremental improvement Limited documentation Cross-functional teams	Enterprise resource planning (ERP) systems E-commerce platforms Content management systems (CMS) Social media platforms Legacy Systems Scientific and Research Software Custom Business Applications Financial Systems
27	Well-defined domain Complex dependencies Sequential Execution Lack of Parallelism	Large and Complex Enterprise Applications Legacy Systems Scientific and Research Software Custom Business Applications

	Access Patterns Performance Profiling Compatibility Considerations	Content Management Systems (CMS) E-commerce Platforms Financial Systems
28	Large codebase Well-defined quality attributes Complex dependencies Maintenance Challenges Technology Stack Heterogeneity Performance Bottlenecks	Enterprise Resource Planning (ERP) Systems Customer Relationship Management (CRM) Systems Healthcare Information Systems E-commerce Platforms Financial Systems Manufacturing and Supply Chain Systems
29	Large codebase Well-defined domain Loosely coupled components Clear boundaries between components Isolation of Faults Rapid Development and Deployment Resource Efficiency Service Reusability	Enterprise resource planning (ERP) systems Content management systems (CMS) Banking and financial systems Telecom systems Large Enterprise Systems E-commerce Platforms Content Management Systems (CMS) Manufacturing and Supply Chain Systems
30	Modular Structure Clear Functional Boundaries Technological Diversity Isolation of Faults Improved Maintainability Rapid Development and Deployment Evolving Requirements	Enterprise resource planning (ERP) systems E-commerce platforms Content management systems (CMS) Social media platforms Banking and financial systems Healthcare systems Telecom systems
31	Modular Structure Clear Functional Boundaries Business Logic Separation Limited Cross-Cutting Concerns Well-Defined Interfaces Maintainability Challenges Technological Diversity	Enterprise resource planning (ERP) systems E-commerce platforms Content management systems (CMS) Social media platforms Banking and financial systems Healthcare systems Telecom systems
32	Complexity and Size Separation of Concerns Modularization Potential Limited Cross-Cutting Concerns Outdated Technology Stack	Content management system (CMS) Enterprise resource planning (ERP) Customer relationship management (CRM) Social media platform E-learning platform
33	Modular Monoliths Layered Architectures Service-Oriented Monoliths Separation of Concerns Microservices-Ready Monoliths Large and Complex Monoliths	Enterprise resource planning (ERP) systems E-commerce platforms Content management systems (CMS) Social media platforms Banking and financial systems Telecom systems

Appendix B

Sl#	Steps Applied in Decomposition Type	Time Required	Cost Required
1	1. Recognize the fundamental components.	From Days to Month	Employee remuneration
	2. Streamline and improve the components.	From Week to Months	Cost related to Refactoring and regression testing.
	3. Determine the dependencies between components.	From Days to Week	Employee remuneration
	4. Categorize components into groups.	From Days to Week	categorization effort and expertise cost

2	1. Gather event logs.	From a few days to several weeks	Cost of log collection and log storage and cost of used tools
	2. Extract insights from the event logs.	From week to months	Cost of log mining, analysis tools, or software.
	3. Determine microservices.	From Several weeks to months	Cost of software architect, documentation effort, and tool and technology
3	1. Recognize the key business functions.	From a few weeks to a couple of months	Cost related to remuneration of analysts, domain experts, and stakeholders
	2. Break down these business functions into microservices.	From several months to a year	Cost related to development resources, new technology adoption, and infrastructure changes
	3. Pinpoint the interrelationships among microservices.	From a few months to a year or more	Cost related to analysis, design, and development of APIs
4	1. Static examination	From a few days to months	Cost related to Tools and expertise
	2. Dynamic assessment	From week to several months	Cost related to hardware, software setting up test environments, automated testing tools, and personnel to design, execute, and analyze the tests.
	3. Manual improvement	From months to year	the salaries and benefits of the development team
5	1. Discover business capabilities.	2-4 weeks	the salaries and benefits of the development team
	2. Break down business capabilities into microservices.	4-8 weeks	the salaries and benefits of the development team
	3. Recognize interdependencies among microservices.	4-8 weeks	the salaries and benefits of the development team
6	1. Bounded contexts with ubiquitous language.	Several weeks to several months	the salaries and benefits of the development team and the cost of external resource
	2. Identify and define Domains	Several weeks to several months	the salaries and benefits of the development team and the price of tools
	3. Aggregates, entities, and value objects	several months	the salaries and benefits of the development team and the price of tools
7	1. Identify the logical components of the monolithic system.	several weeks to a few months	salaries of the team and external consultant
	2. Flatten or refactor the components into smaller, more manageable units.	several months	salaries and cost of tools or technologies
	3. Identify the dependencies between the components.	several weeks to a few months	the salaries and benefits of developers
	4. Group the components into microservices.	several weeks to a few months	developer salaries and cost of resources or tools
8	1. Assessment and Planning	several weeks to a few	salaries of the team and cost tools or resources
	2. Define Microservice Boundaries	few weeks to a couple of months	salaries of the expertise team
	3. Prioritize Technical Debt	several weeks	Salaries of team and cost of specialized tools for code analysis

	4. Refactoring and Code Improvement	several months to years	salaries of the development team and cost tooling or resources
9	1. Identify Core Data Entities	several weeks	Salary and benefits of domain experts and developers
	2. Map Data Relationships	a few weeks to a couple of months	Salary and benefits of domain experts and developers
	3. Define Data Boundaries	several weeks	Cost-related remuneration of developer and architect hours needed.
	4. Identify Data Access Patterns	a few weeks	Salary of developer and architect hours, as well as cost of monitoring and profiling tools
	5. Group Functionality with Data Entities	a few weeks	Cost related to Developer and architect hours
	6. Prioritize and Plan Decomposition	a few weeks to a couple of months	Cost of project management and development hours.
	7. Extract Microservices	several months to over a year	Cost related to developer and QA resources, infrastructure, and new technologies or tools
10	1. Map Business Processes to Components	several weeks	Cost related to business analysts and developers
	2. Determine Dependencies	a few weeks to a couple of months	Cost related to developer and architect hours and costs for tools.
	3. Define Microservices Boundaries	take several weeks	Cost of architect and developer hours
	4. Identify Data Requirements	a few weeks	Cost related to data architects and developers
11	1. Identify User Personas and Use Cases	a few months	Cost related to research, user interviews, and documentation
	2. Create a User Flow Diagram	a few weeks to a couple of months	Remuneration of expertise of UX designers and diagramming tools
	3. Identify User-Centric Boundaries	a few weeks to a few months	Cost involvement of analysis and documentation
	4. Define Microservice Boundaries	several months to a year	Cost of architectural planning, discussions, and potentially hiring experienced architects
	5. Data Modeling and Database Splitting	several months	Cost related to database expertise, migration tools, and testing resources
	1. Identify Technology Dependencies	several weeks to a few months	Costs involve analysis and documentation resources
	2. Define Technology-Oriented Boundaries	several weeks to a few months	Cost related to architecture planning, discussions, and possibly hiring experienced architects
	3. Select Appropriate Microservice Technologies	several weeks to a few months	Cost of research, training, and potential licensing fees for new technologies
	4. Data and Database Consideration	several months to a year	Cost related to database experts, migration tools, and potential data synchronization mechanisms
	5. Refactor and Isolate Components	several months to a year or more	Cost of resources, testing, and rewriting or adapting code to fit the new microservices architecture
13	1. Data Analysis and Preparation	several weeks to a few months	Cost related to resources for system analysis, documentation, and hiring experts
	2. Feature Extraction	several weeks	Remuneration of engineering resources and data scientists

	3. Hierarchical DBSCAN Clustering	several weeks to a few months	Remuneration of data science and machine learning expertise, as well as the cost of software or libraries for clustering
	4. Determining Clustering Parameters	a few weeks	Remuneration of data scientists or machine learning experts
	5. Identify Microservice Boundaries	several weeks to a few months	Cost of architectural planning and discussions
14	1. Data collection	several weeks to a few months	Cost related to resources for data collection, documentation, and hiring experts
	2. Strategy selection	few weeks to a couple of months	Cost related to resources for strategy evaluation, discussions, and potentially consulting with experts
	3. Microservices identification	several months	Costs involve resources with expertise in software architecture
	4. Evaluation	a few weeks to several months	Cost related to architectural evaluation, testing, and potentially prototyping
15	1. Collect the development history	several weeks or even months	Remuneration of developers and possibly data analysis experts.
	2. Identify the microservices	several weeks to months	cost will depend on the expertise of the team and the size of the codebase
	3. Evaluate the microservices	several weeks to months	Cost related to the analysis and decision-making process
	4. Redesign the microservices	several months	Cost of the redesign
	5. Migrate the code	several months to years	Remuneration of developers and tools or automation is used.
16	1. Identify the classes	several weeks to months	Cost related to developers and architects and tools for code analysis
	2. Extract the keywords	several weeks	Cost related to developer data analysts and automated tools
	3. Cluster the classes	several weeks	Cost related to architect and developer and tools for code analysis.
	4. Identify the microservices	several weeks to months	Remuneration of expertise team.
	5. Redesign the microservices	several weeks to months	Remuneration and benefits of developers and architect.
	6. Migrate the code	several months to years	Remuneration and benefits of developers who are responsible for rewriting and retesting code. Also, the cost of migration tools or frameworks are used.
17	1. Structural refactoring	several months to over a year	Remuneration of developers and architects as well as the cost of any tools or technologies required
	2. Behavioral refactoring	several months	Remuneration of developers, architects, and potentially domain experts. It may also involve testing and validation costs.
	3.Data refactoring	several months to complete.	Remuneration and benefit of data engineers or database administrators and transformation. As well as the cost of tools or technologies used for data management.

18	1. Distributed Representation of Source Code	several weeks to months	Cost related to data scientists or machine learning engineers as well as the cost of any tools or computational resources used.
	2. Clustering	several weeks to months	Cost related to data scientists, developers, or architects and the cost of tools or libraries used for clustering
	3. Evaluation	several weeks to months	Benefits of architects, developers, or domain experts
19	1. Identify the concerns in the monolithic application	several weeks to months	Remuneration of developers, architects, and domain experts and cost involve tools or code analysis software
	2. Create the microservices	several months to over a year	salaries of developers and the cost of tools, or technologies required for the microservices.
20	1. Analyze the monolithic application	several weeks to months	Benefits of expertise of software architects and developers and cost of tools or software
	2. Define a quality function	a few weeks to a month	Benefits of expertise of architects
	3. Identify the microservices	several months to a year or more	Benefits of expertise of software architects and developers and cost of testing and validation
21	1. Select the decomposition technique	weeks to a couple of months	Compensation of experts or consultants
	2. Configure the Mono2Micro tool	a few weeks to a couple of months	licensing or subscription fees for the Mono2Micro tool, as well as the time and expertise required to configure and customize the tool for your specific application
	3. Evaluate the microservices	several months	Remuneration of expertise of software architects and developers. It may also include testing and validation costs
22	1. Granularity checking	several weeks to a few months	compensation of experts or consultants
	2. Coupling checking	several weeks to months	cost includes the time and expertise of software architects and developers
	3. Messaging checking	several weeks to months	time and expertise required to implement messaging patterns and potentially the cost of any messaging infrastructure or tools
	4. Security Checking	several weeks to months	Remuneration and benefit of development and security teams
23	1. Problem Identification	several weeks to months	compensation of experts or consultants and involve the cost of tools or software
	2. Problem Decomposition	several months to a year or more	Remuneration of expertise of software architects and developers
	3. Microservice Identification	several months	Remuneration of expertise of software architects and developers
24	1. Identifying dependencies	several weeks to a few months	compensation of experts or consultants and the cost of tools or software for code analysis.
	2. Clustering components	several months to a year	Remuneration and benefits of the expertise of software architects and

			developers.
	3. Identifying microservices	several months	Benefits of expertise of software architects and developers and cost of tools or software
25	1. Extracting features	several weeks to several months	Benefits of the number of resources and cost of tools and software
	2. Building a topic model	take several weeks	The benefits of the expertise of the data scientists or analysts and the tools or software cost
	3. Identifying microservices	several weeks to several months	The salary of the architects and developers
26	1. Initialize the population	few weeks	The salary of architects, developers, and data scientists and the cost of specialized software or tools
	2. Evaluate the solutions	weeks to months	Salary of expertise required to assess each solution's suitability
	3. Select the parents	a few days or less.	Minimal cost is associated with selecting parents, as it's a straightforward algorithmic process
	4. Crossover	a few weeks	cost includes development time to implement crossover operations and potential computational resources for simulation or testing
	5. Mutation	a few days	cost includes development time to analyze mutation operations and potential computational resources for simulation or testing
	6. Repeat steps 2-5	As 2-5	As 2-5
27	1. Vectorize the code	several weeks to several months	Benefit of expertise required for code analysis and vectorization and cost of tools or software used for code analysis
	2. Identify microservices	weeks to months	Benefits of architects and developers cost of specialized software or tools for clustering and analysis.
	3. Consider the sequence of accesses	weeks to several months	Benefits of expertise and the cost of monitoring tools or logging infrastructure to gather data on runtime interactions.
28	1. Analyze the legacy application	several weeks to several months	Benefits of software architects, developers, and analysts who assess the legacy application and cost of tools or software for code analysis and documentation.
	2. Identify quality attributes	several weeks	Benefits of Expertise.
	3. Decompose the application	several weeks to several months	Salary of software architects and developers
29	1. Identify candidate microservices	several weeks to several months	Benefits of software architects and developers
	2. Refactor the classes	several months or more	Benefits of software developers
	3. Design the microservices	several weeks to several months	Benefits of software architects and developers and cost of tools or

			frameworks for designing microservices
30	1. Analyze the monolithic application	several weeks to months	salaries of the engineers and architects
	2. Identify microservice candidates	several weeks to months	salaries of engineers, architects, and possibly domain experts and cost of tools or software for code analysis
	3. Evaluate the microservice candidates	several weeks to months	salaries of engineers and architects involved in the evaluation process
31	1. Model the application as a graph	several weeks to months	salaries of software architects and engineers, and cost-specialized modeling tools or software
	2. Identify microservice candidates	several weeks to months	salaries of software architects, engineers, and domain experts
	3. Optimize the decomposition	several weeks to months	salaries of software architects and engineers
32	1. Identify ORM entities	several weeks to months	salaries of developers and database experts, and the cost of database analysis tools
	2. Group ORM entities	several weeks to months	salaries of developers, architects, and domain experts
	3. Refactor the application	several months to years	salaries of developers, architects, testers, and project managers
33	1. Identification of microservice candidates	weeks to a few months	salaries of software architects, developers, and domain experts, and the cost of tools for code analysis and documentation
	2. Design of the microservice architecture	several weeks to several months	salaries of software architects, designers, and developers
	3. Development of the microservices	several months to a year or more	salaries of developers, testers, and project managers