BOAM: a Business Oriented identification Approach of Microservices within legacy systems

Brahim Mahmoudi, Imen Trabelsi, Dalila Tamzalit, Naouel Moha, Yann-Gaël Guéhéneuc

BOAM: a Business Oriented identification Approach of Microservices within legacy systems

Mahmoudi Brahim[1], Trabelsi Imen[1], Tamzalit Dalila[2], Moha Naouel[1], and Guéhéneuc Yann-Gaël[3]

[1] École de technologie Supérieure, Montréal, QC, Canada
brahim.mahmoudi.1@ens.etsmtl.ca
[2] Laboratoire des Sciences du Numérique de Nantes , Nantes, France
[3] Concordia University, Montréal, QC, Canada

Abstract. The microservices architecture (MSA) is highly popular for its scalability, deployability in the Cloud and compatibility with DevOps practices. Many companies are migrating their legacy systems to an MSA. They need to rely on automatic approaches to ease their migration while taking into account their business features. Existing migration approaches to an MSA often focus on technical features but neglect functional ones, which are essential for appropriate MS granularity. To address this lack, we introduce BOAM (Business Oriented identification Approach of Microservices), a hybrid approach that focuses on business decomposition by leveraging not only technical features, such as source code, but also business oriented artifacts, especially *use cases*. BOAM thus leverages static and semantic analyses of source code using *nanoentities* (data, operations or artifacts), followed by a semantic analysis of use cases to capture business features. For that, BOAM leans on machine learning, particularly clustering methods, to identify microservices through technical (source code) and business (use cases) artifacts. The goal is to ensure that identified microservices are technically sound and meet specific business features of the company. Our evaluation shows that BOAM outperforms other literature approaches to identify microservices, achieving an average precision of 74.51% and recall of 77.93%.

Keywords: Microservices · Migration · Legacy systems · business features · Use cases · Static analysis · Semantic analysis · Clustering · Nanoentities.

1 Introduction

Legacy systems are vital to many organisations, embedding valuable knowledge and executing critical business logic. However, these systems often consolidate all functionalities into a single monolith, leading to challenges like high maintenance costs, limited scalability, and portability issues [18]. While new systems can be designed with a microservices architecture (MSA) from the start, migrating legacy systems is complex. This complexity includes identifying tightly coupled

components, addressing data migration, and reorganizing processes. The primary challenge is identifying microservices within the legacy system [13, 14, 20], which involves delineating functional components based on business features, domain boundaries, or technical concerns to enhance scalability, flexibility, and maintainability. Business features are functional capabilities which are specific to a given company. They are defined to achieve the company's objectives and comply with various requirements, like for instance performance requirements or regulatory compliance. For example, a business feature for a streaming service might include high-performance video delivery, ensuring smooth playback without buffering but also ensure to respect clients' privacy. These features are generally tied to business artifacts, such as use cases. These lasts detail user-system interactions and ensure that identified microservices align with both technical and business requirements.

Most existing approaches of microservices identification focus on technical features (source code), often neglecting the specific functionalities and responsibilities to identify microservices that should align with business features [5,23]. This strategy of identifying microservices with a purely technical approach without considering business features fails to integrate use cases as business oriented artifacts, potentially leading to technically sound but business-misaligned microservices [27, 28]. Taking business features into account ensures that microservices are aligned with an organization's strategic goals and operational requirements, making them more valuable and effective. Ignoring these features can lead to microservices that fail to support the business needs and may even hinder the organization's objectives. For example, if a business feature requires real-time data processing for its e-commerce platform, a microservices that handles order processing must meet performance constraints to ensure fast transaction times. Ignoring this constraint could result in slow order processing, leading to poor customer experience and potential loss of sales. Most approaches for identifying microservices focus only on source code, with very few incorporating use cases [13,14]. We propose BOAM (Business Oriented identification Approach of Microservices) to fill this gap by considering use cases since they can play a critical role in aligning microservices with business features. It integrates business oriented artifacts with source code for microservices identification, ensuring they meet both business and technical requirements [25]. BOAM employs use cases to represent business features, leading to business oriented microservices that better meet operational needs [17]. Even when use cases are unavailable, they can be derived from source code to bridge the gap between technical implementation and business requirements as explained in Section 4.3. BOAM also leverages machine learning and clustering of source code, integrating semantic analysis of both source code and use cases to improve microservices identification precision. We evaluate BOAM through experiments with eight systems of varying complexity and size, showing an average precision of 74.51% and recall of 77.93%. We compare these results with another literature approach, MicroMiner [23], which does not use business artifacts but has been shown to outperform other approaches in microservices identification. This comparison highlights the im-

pact of incorporating business oriented artifacts into the process of identifying microservices.

The paper is organized as follows: Section 2 presents related works and their limitations. Section 3 presents our approach. Section 4 describes the empirical evaluation. Section 5 discusses the results and future work. Finally, Section 6 concludes the paper.

2 Related Work

The identification of microservices within legacy software systems has been a significant research focus, with various studies aiming to automate this process. A common limitation is the minimal use of business-oriented artifacts in their approach. We categorize these studies into two main categories of microservices identification: technical-centric and business-oriented-centric. Technical-centric approaches focus primarily on technical aspects such as code structure, performance, and scalability. Business-oriented-centric approaches emphasize the use of business artifacts such as use cases, business processes or domain models to guide the migration.

Several studies of the first categorie focus on the identification of microservices using diverse technical-centric approaches. Escobar et al. [8] proposed a rule-based algorithm to extract microservices from JEE systems. Their method involves static code analysis, associating each session bean with a cluster, which are then aggregated based on a threshold criterion. This approach is efficient for systems with well-defined session beans but may struggle with more loosely coupled architectures where session beans are not as prominent.

Trabelsi et al. [23] introduced MicroMiner, a three-step approach that integrates static and semantic code analyses. It uses an SVM algorithm for class classification and clustering techniques to form initial typed services, followed by soft clustering to refine microservices. They later enhanced this approach with MicroMatic [24], making the process more automated. While both methods are technically robust, they share a common limitation: minimal use of business-oriented artifacts such as use cases or business processes, which are crucial for aligning microservices with business goals. Unlike Escobar et al.'s rule-based approach, these methods incorporate semantic analysis for better contextual understanding of the code but remain focused on technical aspects.

Gysel et al. [16] proposed ServiceCutter, a tool for segmenting monolithic systems into microservices by analyzing code dependencies and data flows. It optimizes service boundaries to reduce inter-service coupling and enhance cohesion. However, its strong emphasis on technical dependencies can overlook business logic and user requirements, potentially resulting in microservices that are well-structured technically but misaligned with business needs.

Dehghani et al. [6] proposed a reinforcement learning (RL) framework for microservices decomposition. It uses the ServiceCutter tool to identify services, which are then processed by an RL algorithm to map methods to microservices. This innovative approach leverages the adaptability of RL to optimize microser-

vices boundaries dynamically. However, the complexity and computational requirements of RL models might pose challenges for practical implementation in large-scale systems, requiring substantial expertise and resources.

While several approaches for identifying microservices based on code dependencies and technical metrics exist, the need for business oriented artifacts in the microservices identification process has been often overlooked. Only a few studies attempt to bridge this gap:

Tyszberowicz et al. [25] focused on analyzing use case specifications and decomposing functional requirements to derive microservices. This method relies on manual effort and tools for phrase extraction and clustering visualization, emphasizing understanding system functionalities. However, it does not include source code analysis, which is crucial for ensuring that the identified microservices are compatible with the company's existing system.

Zougari et al. [29] developed an automated approach to identify microservices from business process models by analyzing control, information, and semantic dependencies. Unlike Tyszberowicz et al.'s manual method, Zougari et al.'s approach is fully automated but is similarly limited in generalizability, having been tested on a single case study that may not represent the complexity of real-world business processes.

In contrast, Gouigoux et al. [13, 14] highlighted the importance of the functional dimension in enterprise systems, advocating for the inclusion of functional and semantic analysis in microservices identification. However, they offer high-level guidance without detailed technical methods or tools.

Li et al. [19] developed RM2MS, which automates microservices identification from requirements models by analyzing dependencies between functionalities and data structures. Unlike Gouigoux et al., this approach offers a detailed framework and technical methods but relies heavily on UML models, which are not always available, and does not specify how to create these models, making it difficult to implement in practice.

Our approach, BOAM, aims to bridge this gap integrating business oriented artifacts directly into the microservices identification process. By combining static and semantic code analyses with machine learning and business oriented artifacts (use cases), BOAM aims to ensure that identified microservices are aligned with both technical structure and business features. BOAM takes into account both the structure of the system and its business objectives, focusing on business functionalities to provide business oriented microservices.

3 Approach

We organize our approach BOAM in four steps illustrated in Fig. 1 and explained hereafter.

3.1 Step 1: Use Cases Preparation

Input: Source Code.
Output: Use Cases.

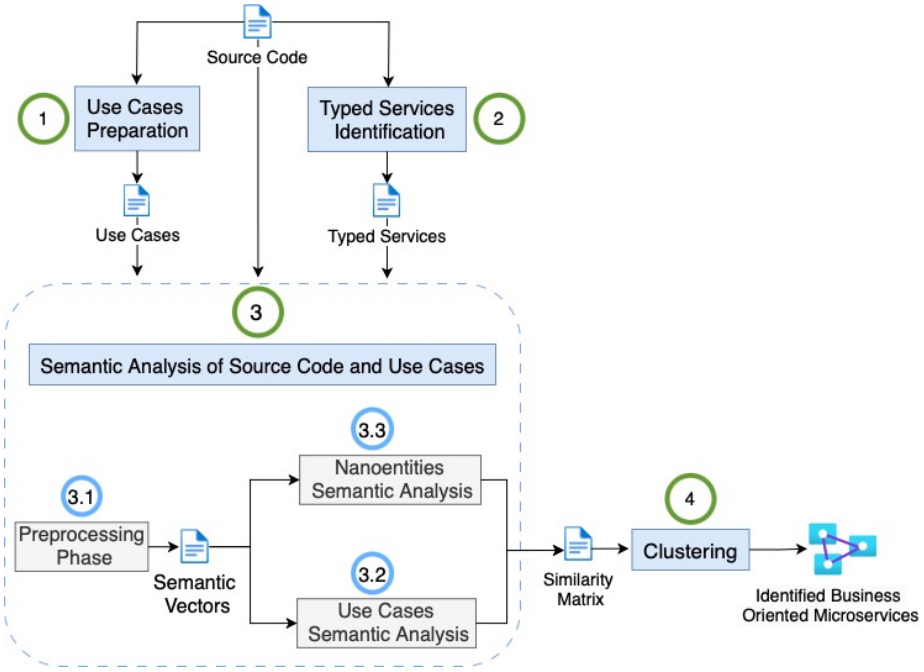
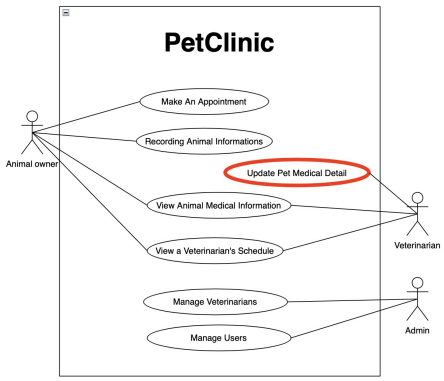


Fig. 1: Workflow of BOAM

Description: Use cases serve as essential blueprints for designing and developing microservices tailored to business features. They provide a clear representation of the system's functionality and user interactions. Traditionally, UML use cases are widely recognized and adopted for their simplicity and higher-level abstraction. These use cases formalism, as illustrated in Fig.2a, illustrate interactions between actors and the system, highlighting various use cases for each actor. However, Gysel et al. [15, 16] proposed a different formalism, the *nanoentity* format as illustrated in Fig.2b , which offers a more granular representation than UML. Each use case in this format includes information on attributes that are read (*nanoentitiesRead*) and modified (*nanoentitiesWritten*). This formalism allows for detailed analysis of use cases for the identification of microservices. The nanoentity format provides advantages over traditional UML use cases. It simplifies the analysis by explicitly showing the attributes involved in each use case, making it easier to identify microservices. Additionally, it facilitates more straightforward automation of the analysis process.

Implementation: Given these benefits, we adopt the nanoentity format for our approach to analyze the system to identify key business features and user scenarios, ensuring the use cases are consistent and align with our approach's standards. To validate and refine the use cases, we conduct manual inspections and reviews, ensuring they accurately reflect the system's intended behavior and business logic. These iterative refinements help us understand the system's



(a) UML use cases

```
{
    "useCases":
    [
        {
            "name":"UpdatePetMedicalDetail",
            "nanoentitiesRead":
                    ["Pet.birthDate","Pet.type",
                     "Pet.owner","Pet.visits",
                     "NamedEntity.name"],

            "nanoentitiesWritten":
                    ["Pet.owner","Pet.visits",
                     "NamedEntity.name"]
        },...
    ]
}
```

(b) Nanoentity use cases

Fig. 2: Comparison between UML and nanoentity use cases

requirements better, ensuring that the resulting microservices effectively fulfill the identified business features. This formalism is a systematic representation of UML use cases, and the detailed procedure for translating UML use cases into nanoentities is available in the replication package[4].

3.2 Step 2: Typed Services Identification

Input: Source Code.
Output: Typed Services.
Implementation: Typed services serve as a foundational element, guiding the microservices identification process. [1]. These types include Entity Services, which are responsible for accessing and manipulating data for a specific entity; Utility Services, which provide common, reusable functionality; and Application Services, which coordinate calls to multiple services and implement application-specific business logic. Identifying these types is crucial for understanding the role and responsibility of each service. We use the first two phases of MicroMiner [23] (*R2.7 Phase 1: Class typing and Phase 2: Typed services identification*) to generate typed services because this approach outperforms other literature approaches and typed services align closely with the system's existing structure and business logic, facilitating a smoother and simpler transition from a monolithic to a microservices architecture.

3.3 Step 3: Semantic Analysis of Source Code and Use Cases

We perform a semantic analysis of the source code and use cases using a standard Natural Language Processing (NLP) method based on semantic distance [7].

[4] `https://github.com/Brahim-Mahmoudi/BOAM_Repo/tree/main/BOAMWorkshops/UseCaseGeneration/UseCaseGeneration.jpg`

Each use case is defined with associated nanoentities and names, which facilitates the semantic analysis, as outlined in Steps 3.1, 3.2, and 3.3. The goal is to transform the source code and use cases into a vector representation, allowing us to analyze their semantic proximity.

Step 3.1: Preprocessing Phase.
Input: Typed Services, Source Code.
Output: Semantic Vectors *(of each class of the system).*
Implementation: We parse, tokenize, and clean elements of the system for example *class names, method names, variable names, code comments*, which are terms associated with programming languages, and we standardize terms through lemmatization, which is often more precise than stemming, as it takes into account the grammatical context of words [3]. Each term is then transformed into a numerical representation using the Google Word2Vec model[5]. This standardization results in a numerical representation that captures the semantic and syntactic context. For each element, we average the embeddings of all its associated terms to combine their semantic and syntactic information into a representative vector. This is done because averaging allows us to combine the semantic information of all terms into a vector that represents the entire element.

Step 3.2: Use Cases Semantic Analysis.
Input: Semantic Vectors from Step 3.1, Use Cases.
Output: Similarity Scores between Use Cases and Typed Services.
Implementation: We calculate the similarity between the the two inputs using cosine similarity, a method well-suited for capturing semantic similarity between terms [21].

$$\text{Cosine Similarity}(S, UC) = \frac{\sum S_i \cdot UC_i}{\|S\| \cdot \|UC\|} \tag{1}$$

where S and UC represent the lemmatized service and use case names, S_i and UC_i are their respective vector terms, and $|S|$ and $|UC|$ are their magnitudes. The final score is the average of all cosine similarities between the use case name and the preprocessed classes of the typed service identified in Step 2. This score ranges from 0 to 1, with higher scores indicating stronger similarity.

Step 3.3: Nanoentities Semantic Analysis.
Input: Semantic Vectors from Step 3.1, Use Cases.
Output: Similarity Matrix between Use Cases and Typed Services.
Implementation: As detailed in Algorithm 1, we conduct a semantic analysis for each nanoentity of the use case and each semantic vectors from Step 3.1. We assign a higher semantic weight to the *write* nanoentities, using a designated weight distribution (0.7 for nanoentitiesWritten and for nanoentitiesRead). This is based on Tyszberowicz et al.'s approach [25], which introduces a differential weighting factor for *write* and *read* nanoentities, emphasizing the greater impact

[5] https://github.com/mmihaltz/word2vec-GoogleNews-vectors

of *write* operations on system functionality and cohesion. To achieve a comprehensive semantic analysis, we combine the results from Step 3.2 and Step 3.3 to obtain a semantic similarity matrix between each use case name and each service identified in Step 2.

Algorithm:

Algorithm 1: Nanoentities Semantic Analysis

Input: $\mathcal{V}$: Semantic Vectors from Step 3.1, $\mathcal{U}$: Use Cases
Output: $\mathcal{S}$: Similarity Matrix between Use Cases and Typed Services

begin
  foreach $u \in \mathcal{U}$ **do**
    $\mathbf{v}_u^{(3.3)} \leftarrow \mathbf{0}$;
    foreach $n \in u$ **do**
      foreach $\mathbf{v}(c) \in \mathcal{V}$ **do**
        temp_sim $\leftarrow \cos(\mathbf{v}(n), \mathbf{v}(c)) \cdot (0.7 \cdot \mathbb{I}(n.\text{type} = \text{"write"}) + 0.3 \cdot \mathbb{I}(n.\text{type} = \text{"read"}))$;
        $\mathbf{v}_u^{(3.3)} \leftarrow \mathbf{v}_u^{(3.3)} + \text{temp\_sim}$;
    $\mathbf{v}_u^{(3.3)} \leftarrow \frac{\mathbf{v}_u^{(3.3)}}{|u|}$;
    $\mathbf{v}_u^{combined} \leftarrow \alpha \cdot \mathbf{v}_u^{(3.2)} + \beta \cdot \mathbf{v}_u^{(3.3)}$;
  foreach $u \in \mathcal{U}$ **do**
    foreach $s \in \mathcal{V}$ **do**
      $\mathbf{v}_s \leftarrow \frac{1}{|s|} \sum_{c \in s} \mathbf{v}(c)$;
      $\text{final\_sim}_{u,s} \leftarrow \cos(\mathbf{v}_u^{combined}, \mathbf{v}_s)$;
      $\mathcal{S}[(u, s)] \leftarrow \text{final\_sim}_{u,s}$;
  return $\mathcal{S}$;

3.4 Step 4: Clustering

Input: Similarity matrix from Step 3.
Output: Identified Business Oriented Microservices.
Implementation: We use the Fuzzy C-Means algorithm and the semantic similarity matrix from the previous step to determine the degree of membership of each service (composed of classes identified in Step 3) in each cluster. Unlike K-means, Fuzzy C-Means allows for flexible assignment of services to clusters, with each service potentially belonging to multiple clusters, which is useful when services are associated with multiple business features [4]. We perform a single iteration of the Fuzzy C-Means algorithm to fix cluster centers, corresponding to each use case, maintaining a link between use cases and services to align with business needs [25]. This clustering assigns each service identified in Step 3 of BOAM to a cluster and quantifies the degree of service membership within that cluster. The membership matrix U is calculated as follows:

$$U_{ij} = \frac{1}{\sum_{k=1}^{M} \left(\frac{d_{ij}}{d_{ik}}\right)^{\frac{2}{m-1}}} \tag{2}$$

where U_{ij} represents the degree of membership of the i^{th} typed service in the j^{th} cluster, d_{ij} is the distance between the i^{th} typed service and the center of the j^{th} cluster (the use case), M is the total number of clusters, and m is the fuzziness exponent that controls the transition between true and false values, affecting the membership functions and tolerance for ambiguity. We chose $m = 2$, chosen drawing on relevant works [10, 23].

4 Evaluation

In this section, we evaluate BOAM by comparing it with the ground truth and an other literature approach MICROMINER [23]. We introduce the case studies used in our evaluation and discuss the creation of ground truths, which are crucial for the quantitative assessment of microservices identification. We then present the metrics used to measure the accuracy of our approach and the quantitative results obtained. The tool and most of the data (as explained in Section 4.1) are available online[6].

4.1 Case Studies

To gather our set of case studies, we established specific criteria: (1) open-source (OS), (2) publicly available online, (3) written in Java, (4) have monolithic architecture and above all (5) have use cases available. Despite our efforts, finding truly complex systems that meet all criteria and provide corporate system access is challenging. Consequently, we identified only one system (Cargo) that fully meets all criteria. Additionally, for our evaluation, we included three other open-source systems commonly used in the literature for microservices identification [1,16,23]. We also used four projects from university students enrolled in a Master's degree analysis and modeling course. These students manually constructed small systems from use cases to implementation. In total, we have eight systems, as shown in Table 1, providing a diverse range for assessing the scalability and accuracy of our approach. Due to confidentiality requirements imposed by national regulations, we cannot share data related to these systems. Access to the code requires signing a confidentiality agreement, as it constitutes intellectual property. However, these systems are available upon request.

4.2 Ground truth: Expected Microservices

We created a ground truth for each system using two groups of three PhD students who were not previously involved with the projects. Each student independently analyzed the source code, class diagrams, and use cases to identify microservices. They then cross-validated their findings to minimize bias, resolving discrepancies through mutual agreement to strengthen the reliability of the ground truth. Details of the systematic creation process are available online[7]. In the final step, the established ground truth was reviewed by the authors. This additional analysis provided a broader perspective, further enhancing the validity of the ground truth. Despite meticulous efforts, manually defining ground truth remains difficult. Identifying exact microservices in legacy systems is challenging, as there can be various valid interpretations. Despite rigorous analysis and cross-validation to reduce bias, the established ground truth is an approximation of reality. Nonetheless, following a systematic process ensures accurate identification of microservices.

[6] https://github.com/Brahim-Mahmoudi/BOAM_Repo.git

[7] https://github.com/Brahim-Mahmoudi/BOAM_Repo/tree/main/BOAMWorkshops/GroundTruth

Table 1: Overview of Case Studies

System	Size		Use Cases Ground Truth		#Microservices
	#Classes	LOC	Preparation	Preparation	
JForum (OS)	271	33837	7,5h	5h	20
Cargo (OS)	99	7336	Already Available	4h	9
POS (OS)	55	4037	2h	3h	5
PetClinic (OS)	52	3521	2h	2h	7
Powerlifting	41	3013	Already Available	2h	4
PoolPro	29	2795	Already Available	1,5h	4
MovieNight	24	1604	Already Available	1,5h	5
DestopWarold	18	3285	Already Available	1h	3

4.3 Required Use Cases

We designed our approach to gathering and creating use cases to reflect common industry practices [2]. In real-world scenarios, use cases often derive from various sources, including existing documentation, team discussions, and hands-on development. We begin by searching for online resources, as we did for Cargo, mirroring industry practice where companies rely on existing documentation to understand system requirements. Secondly, we organized a workshop for PetClinic and POS, reflecting industry brainstorming and discussion sessions [11, 12]. Two teams of three students, not previously involved with the projects, collaboratively developed use cases based on available resources. They then cross-validated their findings. Details of the workshop are available online[8]. These sessions facilitate the exchange of insights, leading to a more comprehensive understanding of the system's requirements. Thirdly, for JForum, the first two authors individually created use cases, following the same process as the second point to ensure thorough analysis and validation. This method incorporated multiple viewpoints for a comprehensive perspective on the system's requirements. They took this approach due to the substantial size of JForum, as shown in Table 1, which was too demanding for volunteer students. Lastly, we included student projects, reflecting the industry's reliance on prototyping and iterative development. These projects, varying in size and complexity, provide valuable hands-on experience applicable to real-world software development.

4.4 Evaluation Metrics

In our evaluation of the microservices identification process, we use precision (P), recall (R), and F1-score (F_1). **Precision (P)** measures the accuracy of correctly identifying microservices: $P = TruePositives(TP)/(TP + FalsePositives(FP))$; **Recall ($R$)** assesses the ability to capture all true microservices: $R = TP\ /\ (TP + FalseNegatives(FN))$; **F1-score ($F_1$)** combines precision and recall: $F_1 = (2PR)\ /\ (P + R)$. We computed these metrics using the ground truths to quantitatively assess the performance of our approach.

[8] https://github.com/Brahim-Mahmoudi/BOAM_Repo/tree/main/BOAMWorkshops/UseCaseGeneration

4.5 Quantitative Results

Comparison We compare our results with MicroMiner [23], focusing on identifying business oriented microservices. Direct comparisons with other literature approaches are challenging due to differing goals, such as business vs. domain-oriented microservices. However, MicroMiner already outperformed other literature approaches like ServiceCutter and Topic Modeling[9], making it a relevant benchmark. The results reveal that our approach outperforms MicroMiner in terms of precision and recall, with an average value of 74.51% and 77.93% respectively, as detailed in Table 2. Our primary objective is to achieve higher precision in microservices identification compared to approaches that do not consider business oriented artifacts. Emphasizing precision is crucial for accurately identifying true microservices without excessive false positives, where false positives refer to classes being assigned to a microservice when they should not be. MicroMiner tends to be more inclusive, capturing a broader range of services related to use cases, which results in lower precision as some identified services may not closely align with the business oriented focus of our approach. BOAM, with its balanced precision and recall, shows better performance for our research objectives. This indicates that using business oriented artifacts in microservices identification improves their quality.

Results Results show that incorporating business oriented artifacts improves microservices identification, achieving an average precision of 74.51%, recall of 77.93%, and F1-score of 76.10% (Table 2). For POS (55 classes, 5 microservices), precision is 69.63% and recall is 93.33%, indicating a high rate of correctly identified classes belonging to the right microservice but also more false positives. The smaller size and complexity of POS may make it easier to identify microservices, but this also increases the risk of including irrelevant ones. For JForum (271 classes, 20 microservices), precision is 77.67% and recall is 60.40%, showing very precise identification with fewer true positives. The larger size and complexity of JForum likely make it more challenging to identify all relevant microservices, resulting in a lower recall. Generally, high recall identifies most true positives but may decrease precision, while high precision identifies fewer false positives but may decrease recall.

Table 2: BOAM quantitative results compared with Microminer

Approach	Metric	Cargo	POS	PetClinic	JForum	Powerlifting	DesktopWarold	MoovieNight	PoolPro	Average
BOAM	**Precision** *(%)*	73.80	69.63	71.59	77.67	85.42	70.08	75.64	72.22	74.51
	Recall *(%)*	68.35	93.33	75.78	60.40	54.40	83.64	100.00	87.50	77.93
	F1-score *(%)*	71.01	79.76	73.63	66.46	66.47	76.26	85.55	79.64	76.10
MicroMiner	**Precision** *(%)*	68.15	72.01	71.41	53.89	81.50	76.30	72.80	68.90	70.75
	Recall *(%)*	63.40	87.98	71.43	76.13	51.30	80.10	95.50	84.60	76.30
	F1-score *(%)*	65.66	79.26	71.44	62.99	62.83	78.15	82.33	76.15	72.35

[9] https://github.com/miguelfbrito/microservice-identification

5 Discussion and Future Work

We present *Qualitative Results* of our approach, followed by *Threats to Validity* of our results. We conclude with a discussion on envisioned *Future Works*.

5.1 Qualitative Analysis

Business Feature Coherence: BOAM tends to produce microservices that are more coherent in terms of business features, as demonstrated by our evaluation across multiple systems. For example, in the *Cargo* system, BOAM has shown the ability to encapsulate business-oriented artifacts, clustering classes like *Itinerary* and *RouteSpecification* into coherent microservices for the *ViewCargos* use case. This contrasts with MicroMiner, which sometimes groups classes based on syntactic or technical similarities without consideration of business features.
Class Distribution and Business Artifacts: The weighting method used by BOAM, which assigns higher weights to classes associated with nanoentities marked as *written*, has enabled more precise identification of microservices. Conversely, MicroMiner has shown a tendency to include additional classes that are not directly functional in some microservices, as observed in the *HandleCargoEvent* use case in the *Cargo* system.
Balance Between Business Relevance and Functional Utility: The observations show that BOAM achieves a better balance between business relevance and functional utility of microservices. For instance, in the *Powerlifting* system, the microservices identified by BOAM have a class distribution that better reflects the functional needs of business artifacts, unlike MicroMiner which may sometimes sacrifice functional coherence for broader business features coverage.

5.2 Threats To Validity

We now discuss threats to the validity of our evaluation.
Internal Validity. Our microservices identification approach and its validation pose potential threats to internal validity due to reliance on metrics and thresholds. To mitigate these concerns, we incorporated threshold values sourced from previous studies. The qualitative evaluation included four authors and one PhD student specializing in microservices development. These participants were actively involved in general work discussions and meetings related to our approach, potentially introducing bias. We acknowledge this bias for two main reasons: (1) the authors are experts in microservices development and (2) sourcing an independent expert for validation across multiple systems proved challenging.
External Validity. Our use of use cases presents a potential threat to external validity. The process described in Section 4.3 was driven by the scarcity of accessible use cases in open-source monolithic legacy systems. To enhance our validation, we expanded our research beyond publicly available sources by generating additional use cases from existing code and documentation of other systems. This approach involved two strategies: initially producing two use cases independently through a workshop, and subsequently creating a use case for one additional system by the authors. We followed the nanoentity format, which focuses on nanoentity interactions for specific functions. This format is simplified compared to traditional use cases but is designed for automated service identification. Traditional UML use cases can include information

like business processes and user interactions, which were not our main focus here. Our study, conducted with a select pool of suitable open source systems, lays a foundation for further research in microservices identification. Recognizing the need to broaden the range of systems studied and to innovate methodologies for deeper analysis, we offer our data for future research in the replication package[10]. This provides a valuable resource for others to expand on our findings and further progress in this field.

5.3 Future Work

Now that we can identify microservices aligned with specific use cases, our goal is to integrate Domain-Driven Design (DDD) principles [9] as advocated in [13, 14], alongside Bounded Contexts. These are widely used techniques for transitioning from monolithic architectures to microservices [26]. Our challenge lies in establishing a clear relationship between use cases and DDD principles within our microservices identification process. Another potential direction is exploring whether descending to method-level granularity could enhance the precision and utility of identified microservices. This shift towards finer granularity at the method level might offer a more precise and functionally cohesive representation. We will automate the threshold selection process for microservices identification with machine learning algorithms to predict optimal thresholds based on historical data and system characteristics. This automated approach, adaptable to each system, will save time and effort for software experts. Additionally, we aim to generate use cases for a system such as Compiere[11], a Java-based open-source ERP system with over 1000 classes. This endeavor would involve rigorously following the steps outlined in Section 4.3. Finally, we aim to broaden our approach by using established models in standard languages like UML [22], and possibly leveraging the RM2MS tool [19] as mentioned in Section 2, this tool automates the identification and visualization of microservices from requirement models that include classes, use cases, sequence diagrams, and OCL contracts..

6 Conclusion

In this paper, we introduced BOAM, a novel approach for microservices identification that performs business oriented analysis of microservices in legacy systems. BOAM uses static and semantic analyses of source code and use cases, incorporating machine learning clustering methods to effectively identify microservices based on technical and business oriented artifacts. We evaluated BOAM on eight systems—four real-world legacy systems and four student legacy systems—comparing results with independently established ground-truths. Results show that BOAM achieves an average precision of 74.51%, recall of 77.93%, and F1-score of 75.14%, outperforming MicroMiner with average precision of 70.75%, recall of 76.30%, and F1-score of 72.35%.

Acknowledgment

We sincerely thank the professor for providing the student systems that greatly enhanced our dataset.

[10] https://github.com/Brahim-Mahmoudi/BOAM_Repo.git

[11] https://www.compiere.com/products/capabilities/

References

1. Abdellatif, M., Tighilt, R., Moha, N., Mili, H., El Boussaidi, G., Privat, J., Guéhéneuc, Y.G.: A type-sensitive service identification approach for legacy-to-soa migration. In: Service-Oriented Computing: 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings 18. pp. 476–491. Springer (2020)
2. Annett, R.: Working with Legacy Systems. Packt (2019), ^1^
3. Balakrishnan, V., Ethel, L.Y.: Stemming and lemmatization: A comparison of retrieval performances. Lecture Notes on Software Engineering **2**, 262–267 (01 2014). https://doi.org/10.7763/LNSE.2014.V2.134
4. Bezdek, J.C., Ehrlich, R., Full, W.: Fcm: The fuzzy c-means clustering algorithm. Computers & geosciences **10**(2-3), 191–203 (1984)
5. Brito, M., Cunha, J., Saraiva, J.: Identification of microservices from monolithic applications through topic modelling. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. pp. 1409–1418 (2021)
6. Dehghani, M., Kolahdouz-Rahimi, S., Tisi, M., Tamzalit, D.: Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning. Software and Systems Modeling **21**(3), 1115–1133 (2022)
7. Elov, B., Khamroeva, S.M., Xusainova, Z.: The pipeline processing of nlp. In: E3S Web of Conferences. vol. 413, p. 03011. EDP Sciences (2023)
8. Escobar, D., Cárdenas, D., Amarillo, R., Castro, E., Garcés, K., Parra, C., Casallas, R.: Towards the understanding and evolution of monolithic applications as microservices. In: 2016 XLII Latin American computing conference (CLEI). pp. 1–11. IEEE (2016)
9. Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional (2004)
10. Gao, X.B., PEI, J.h., XIE, W.x.: A study of weighting exponent m in a fuzzy c-means algorithm. ACTA ELECTONICA SINICA **28**(4), 80 (2000)
11. Gottesdiener, E.: Requirements by Collaboration: Workshops for Defining Needs. Addison-Wesley Professional (2002), ^9^
12. Gottesdiener, E.: Use cases: best practices. Rational Software white paper (2003)
13. Gouigoux, J.P., Tamzalit, D.: From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In: 2017 IEEE international conference on software architecture workshops (ICSAW). pp. 62–65. IEEE (2017)
14. Gouigoux, J.P., Tamzalit, D.: "functional-first" recommendations for beneficial microservices migration and integration lessons learned from an industrial experience. In: International Conference on Software Architecture Companion. pp. 182–186. IEEE (2019)
15. Gysel, M., Kölbener, L.: Service cutter-a structured way to service decomposition. Ph.D. thesis, HSR Hochschule für Technik Rapperswil (2015)
16. Gysel, M., Kölbener, L., Giersche, W., Zimmermann, O.: Service cutter: A systematic approach to service decomposition. In: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Service-Oriented and Cloud Computing, September 5-7. pp. 185–200. Springer (2016)
17. Jacobson, I.: Use cases–yesterday, today, and tomorrow. Software & systems modeling **3**, 210–220 (2004)
18. Lewis, G., Morris, E., Smith, D., O'Brien, L.: Service-oriented migration and reuse technique (smart). In: 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05). pp. 222–229. IEEE (2005)

19. Li, Y., Zhang, Y., Yang, Y., Wang, W., Yin, Y.: Rm2ms: A tool for automatic identification of microservices from requirements models. In: Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems (MODELS'23). Västerås, Sweden (October 2023), demonstration Track
20. Newman, S.: Building microservices. " O'Reilly Media, Inc." (2021)
21. Rahutomo, F., Kitasuka, T., Aritsugi, M., et al.: Semantic cosine similarity. In: The 7th international student conference on advanced science and technology (ICAST). p. 1. University of Seoul, South Korea (2012)
22. Rosenberg, D., Scott, K.: Use case driven object modeling with UML. Springer (1999)
23. Trabelsi, I., Abdellatif, M., Abubaker, A., Moha, N., Mosser, S., Ebrahimi-Kahou, S., Guéhéneuc, Y.G.: From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis. Journal of Software: Evolution and Process p. e2503 (2022)
24. Trabelsi, I., Popa, B., Péreyrol, J., Beaulieu, P.O., Moha, N.: Micromatic: Fully automated microservices identification approach from monolithic systems. In: 2024 IEEE/ACM 6th International Workshop on Software Engineering Research & Practices for the IoT (SERP4IoT). pp. 7–13 (2024). https://doi.org/10.1145/3643794.3648283
25. Tyszberowicz, S., Heinrich, R., Liu, B., Liu, Z.: Identifying microservices using functional decomposition. In: 4th International Symposium, Dependable Software Engineering. Theories, Tools, and Applications. pp. 50–65. Springer (2018)
26. Vernon, V.: Implementing domain-driven design. Addison-Wesley (2013)
27. Wang, Y., Kadiyala, H., Rubin, J.: Promises and challenges of microservices: an exploratory study. Empirical Software Engineering **26**(4), 63 (2021)
28. Waseem, M., Liang, P., Shahin, M., Ahmad, A., Nassab, A.R.: On the nature of issues in five open source microservices systems: An empirical study. In: Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering. pp. 201–210 (2021)
29. Zougari, S., Daoud, M., Sabri, A., Zougari, A., Chahboun, A., Azyat, A.: Automating the recognition of microservices from business process analysis. In: 2024 International Conference on Intelligent Systems and Computer Vision (ISCV). pp. 1–8 (2024). https://doi.org/10.1109/ISCV60512.2024.10620133