


From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis

Imen Trabelsi¹  | Manel Abdellatif¹ | Abdalgader Abubaker^{2,3} | Naouel Moha¹ | Sébastien Mosser⁴ | Samira Ebrahimi-Kahou^{1,2,5} | Yann-Gaël Guéhéneuc⁶

¹Department of Software Engineering and Information Technology, École de Technologie Supérieure, Montreal, QC, Canada

²Mila - Quebec Artificial Intelligence Institute, Montreal, QC, Canada

³Department of Pure Mathematics, University of Khartoum, Khartoum, Sudan

⁴McMaster University, Hamilton, ON, Canada

⁵CIFAR AI Chair, Toronto, ON, Canada

⁶Department of Computer Science and Software Engineering, Concordia University, Montreal, QC, Canada

Correspondence

Imen Trabelsi, Department of Software Engineering and Information Technology, École de Technologie Supérieure, Montreal, Canada.
 Email: imen.trabelsi.1@ens.etsmtl.ca

Abstract

The microservices architecture (MSA) style has been gaining interest in recent years because of its high scalability, ability to be deployed in the cloud, and suitability for DevOps practices. While new applications can adopt MSA from their inception, many legacy monolithic systems must be migrated to an MSA to benefit from the advantages of this architectural style. To support the migration process, we propose *MicroMiner*, a microservices identification approach that is based on static-relationship analyses between code elements as well as semantic analyses of the source code. Our approach relies on machine learning (ML) techniques and uses service types to guide the identification of microservices from legacy monolithic systems. We evaluate the efficiency of our approach on four systems and compare our results to ground-truths and to those of two state-of-the-art approaches. We perform a qualitative evaluation of the resulted microservices by analyzing the business capabilities of the identified microservices. Also a quantitative analysis using the state-of-the-art metrics on independence of functionality and modularity of services was conducted. Our results show the effectiveness of our approach to automate one of the most time-consuming steps in the migration of legacy systems to microservices. The proposed approach identifies architecturally significant microservices with a 68.15% precision and 77% recall.

KEYWORDS

decomposition, microservices, migration, ML, monolith, semantic analysis

1 | INTRODUCTION

The microservice architecture (MSA) has become a prevailing architectural style in the industry. Several major organizations, such as Netflix, Amazon, and eBay, have already adopted this architectural style in their enterprise systems by refactoring their monolithic systems.

MSA-based systems contain groups of self-contained microservices, each running as a separate process designed for a specific function. These microservices communicate with each other through lightweight mechanisms, such as Representational State Transfer Application Programming Interfaces (REST APIs), and are managed by a single team.¹ MSA is popular mainly due to the dynamic and distributed nature of microservices, which offers greater agility and operational efficiency, and reduces the complexity of handling applications scalability and deployment cycles with respect to monolithic systems.²

* Equally contributing authors.

Such characteristics make microservices particularly convenient for migrating and refactoring monolithic software systems by integrating and composing reusable, distributed, and relatively independent microservices.

There are three strategies for an organization to migrate monolithic software systems to microservices. The organization can follow a *top-down*, forward-engineering strategy by (1) performing an analysis of high-level domain artifacts of the monolith (e.g., business processes, use cases, and activity diagrams), (2) decomposing these high-level domain artifacts, (3) modeling the needed microservices that will take part of the targeted microservice-based system, and (3) implementing the defined microservices.

An organization may also adopt a *bottom-up* strategy and re-engineer its monolithic software systems into microservices by (1) extracting and analyzing the dependencies of the monolith, (2) extracting the reusable components (or functionalities) from the existing monoliths that could qualify as microservices, (3) packaging the identified components as microservices to enable their reuse and to remove their dependencies to the legacy infrastructures, and (4) rewriting some existing applications to use the newly created microservices.

Finally, an organization can adopt a *hybrid* strategy by (1) identifying reusable components of its monolithic software systems, (2) mapping these components to available microservices, (3) replacing these components with calls to the appropriate microservices, and (4) implementing the missing parts of the microservices according to defined specifications.

The identification of microservices is a central activity to the three aforementioned migration strategies. Microservices identification is the most challenging step of the overall migration process, especially because the identified microservices must meet a range of expectations regarding their capabilities, the fitness of purpose, quality of service, the efficiency of use, and so forth.^{3–5}

Several approaches have been proposed in the literature to identify microservices in monolithic software systems.^{6–10} However, these approaches have limitations. Most proposed approaches rely on coupling and cohesion metrics to cluster related components in the monoliths and create corresponding microservices. However, coupling and cohesion are insufficient as other aspects of the components are essential to building “proper” microservices. For example, one of the main principles of microservices is the single responsibility principle: A microservice must fit in and stay within a bounded context, within a boundary of the domain model,¹¹ which is a concern orthogonal to cohesion and coupling. Also, many approaches require types of inputs that may not be available for most monolithic software systems, for example, business process models, use cases, and activity diagrams. All these limitations lower the accuracy of these approaches in terms of precision and recall.

To overcome these limitations, we propose an approach called *MicroMiner*, a type-based microservices identification approach that relies on ML and semantic analyses to decompose monolithic software systems into microservices. The resulting microservices respect two main principles: The single responsibility principle and the loose coupling principle. The main point in which our study is different from the existing literature is that *MicroMiner* is guided by the identification of specific types of services (i.e., type-based), which are predicted using a ML classification model. These services are then clustered according to the application domain to form the final microservices. The clustering is based on the analysis of the static and semantic relationships between the monolith's components to form architecturally-relevant microservices (i.e., each microservice belongs to a bounded context and is responsible for a single business functionality).

We validate our approach on four monolithic software systems, build independent ground-truths, and show that *MicroMiner* identifies architecturally-relevant microservices with a precision of 68.15% and a recall of 77%. We further measure and validate the quality of *MicroMiner* using five quality metrics.¹² In addition, we compare our results with two static-based microservices identification approaches: ServiceCutter,¹³ which is an heuristic-based microservices identification approach, and another state-of-the-art microservices identification tool proposed by Brito et al.¹⁴ The results shows that microservice candidates produced by *MicroMiner* are by far the best in terms of functional independence and modularity.

The rest of this paper is as follows. Section 2 summarizes previous works and their limitation. It introduces the taxonomies used in our approach. Section 3 presents our approach in details. Section 4 shows the validation of our approach while Section 5 discusses the threats to the validity of our approach and the recommendations. Section 6 concludes with future work.

2 | BACKGROUND AND RELATED WORK

We now first explain the principles on which we build our approach and then summarize the related work and present their limitations.

2.1 | Software decomposition patterns

2.1.1 | Layered architecture pattern

The layered architecture pattern is one of the earliest architecture models, also referred to as the n-tier architecture model. It is a standard architecture for most OOP architects and developers. Layers are abstract parts of a software system. Each layer plays a specific role within the system and represents a different level and type of abstraction. Layers are built on top of one another.

Various multilayer architectures exist. For example, the classical three-layer architecture¹⁵ consists of the presentation layer, the business layer, and the data layer, as well as the four-layer architecture¹⁶ consisting of the presentation layer, the business layer, the persistence layer, and the database layer. In this article, we assume a four-layer architecture,¹⁷ which includes utility classes, as shown by Figure 1. This architecture divides into the following:

- **Presentation layer:** This layer presents the content to the end-user through a graphical interface. It contains software elements and technologies to interact with the user. Presentation models such as MVVM or MVC can be part of this layer.
- **Utility layer:** This layer provides some cross-cutting functionalities required by other layers. Logging and authentication are examples of the functionalities that can be found in this layer.
- **Business layer:** It is the layer in which the business logic of the application is executed. It is responsible for controlling the functionality of an application by performing detailed business processing.
- **Persistence layer:** This is the lowest layer of this architecture and is primarily concerned with the storage and retrieval of application data.

In the following, we only consider the persistence, business, and utility layers because we focus on services and the presentation uses services but does not provide services to others in itself.

2.1.2 | Service oriented architecture

Layered architectures cannot cope with the complexities of modern large-scale Web applications or enterprise systems because it does not sufficiently partition the system, thus limiting the scalability, flexibility, and distributability of the system. Thus, a finer decomposition emerges. Service-oriented architecture (SOA) is an architectural style by which systems are composed of reusable and platform-independent services. Therefore, each layer would be divided into services. According to the taxonomy provided by Abdellatif et al,¹⁸ we consider four domain-specific services categories:

- **Enterprise services:** They provide generic business functionalities reused across different applications.
- **Application services:** These services provide functionalities specific to one application. Services that implement a business function, activity or task. They are designed to support one or more specific process activities. These services serve as entities. They exist to support reuse within one application or to enable business process services. Such services tend to have less reuse potential than Entity services.

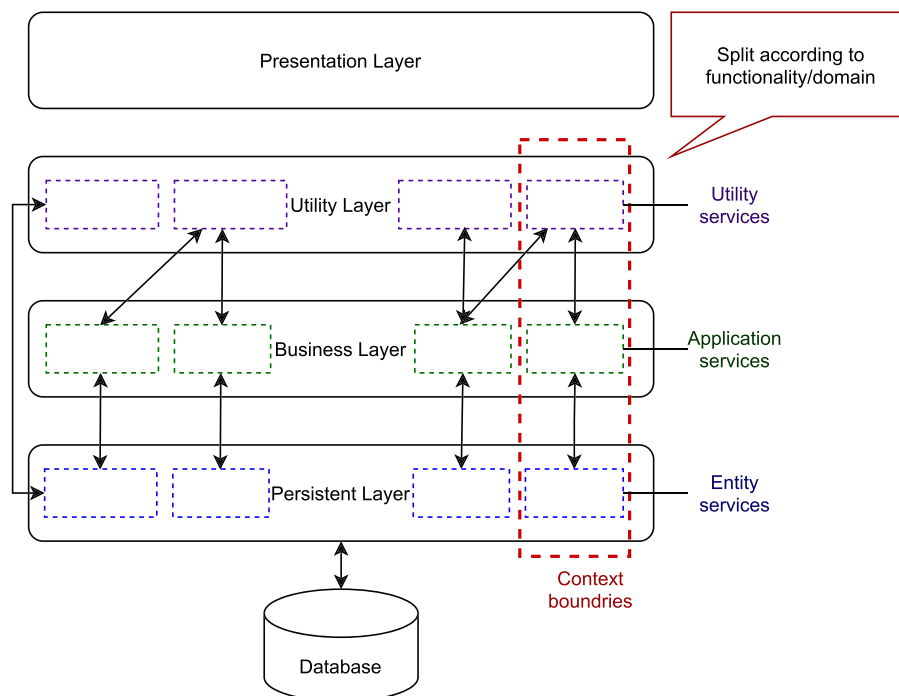


FIGURE 1 Software decomposition

- **Entity services:** or data services, they provide access to and management of the persistent data of legacy software systems. These services provide information of the business process stored in the databases and handle the business entities. Many of its capabilities are related to the traditional CRUD (create, read, update, delete) methods. It is considered a highly reusable service because it can be agnostic to many business processes.
- **Business services:** They correspond to business processes or use cases. These services generally compose/use the Enterprise, Application, and Entity services.

For domain neutral services, we are interested on the Utility services that provide some cross-cutting functionalities required by domain-specific services. These services contain small, closely packed services. They are strictly related to the technological platform supporting the application and business services. Logging and authentication services are examples of Utility services.

As shown in Figure 1, the Persistent layer is broken down into Entity services, the Business layer into Application services, and the Utility layer into Utility services.

In the following, we consider only the identification of three types of services: Utility, Entity, and Application services because we cannot distinguish between Application services and Enterprise services using only source code. Indeed, these two types of services differ only in terms of the scope of reuse: Within a single system or across multiple systems.

2.1.3 | Microservice architecture

Two of the main principles for microservice architecture are the *single responsibility*, and *loose coupling* principles.¹⁹ The microservice architectural styles impose explicit requirements for the number of business responsibilities/capabilities of the services, which is not the case with SOA.²⁰ A microservice should have one and only one responsibility. Therefore, we rely on a domain-related service decomposition²¹ or vertical decomposition (as shown in Figure 1 to generate isolated functional microservices.²²

2.2 | Related work

The identification of services from existing software systems has been the topic of many research works. Several approaches have been proposed to identify services from monolithic software systems, but only a few of them have considered service types.

For example, Abdellatif et al²³ proposed a type-based service identification approach that extracts Utility, Entity, and Application services from legacy software systems. Through static analyses of the source code, they extract several types of interclasses relationships to build the call graphs of the systems and generate clusters that are considered as potential services. They filter and classify each cluster using code metrics and a hierarchical detection rules-based system to identify services and their types. We extend the work of Abdellatif et al²³ and use the typed services as a basis to identify the microservices inside monolithic systems. Different from the approach of Abdellatif et al., we rely on an ML classifier and a clustering algorithm to identify the typed services that will be then aggregated to form the final microservices.

Huergo et al²⁴ also introduced an approach to identify services based on their types. Based on UML class diagrams of object-oriented systems, they started by manually identifying data that played a central role in the operation of an organization and considered them as master data. Each piece of master data is defined as a candidate entity service. Then, still using UML class diagrams, they derived state-machine diagrams related to the identified master data. They analyzed the transitions on the state-machine diagrams to identify Task and Process services. Our work is different from this approach as (1) we identify different types of services, (2) we identify both services and microservices in monolithic systems, and (3) we rely on both static and semantic analysis of the monolithic components to build the final microservices.

Since Microservice architecture and SOA show considerable differences in terms of service granularity, governance, and communication protocol,²⁰ the Legacy-to-SOA migration approaches are not suitable for legacy-to-microservice migration unless various adaptations are performed on them.

Several approaches related to microservices identification have been proposed in the literature. Fritzsc et al⁶ conducted a study on 10 different approaches for refactoring a monolithic application into microservices that use four distinct decomposition strategies. The first strategy is static code analysis, which analyzes the code to find the dependencies between different system units (classes, functions, variables) and eventually derive a decomposition of the system units to identify microservices.^{25,26} The second strategy is based on meta-data: It depends on UML diagrams,⁸ use cases,^{7,8} interfaces,²⁷ or version-control data.²⁸ The purpose is to capture the relationship between users and system functionalities, to show interactions between systems and functional dependencies, and so forth. The third strategy includes workload-data aided approaches.^{10,29} These approaches measure application operational data, such as communications and performance, at the module or function level, and use these data to determine appropriate decomposition and granularity of microservices. The last strategy is the dynamic composition in which the goal is to provide a runtime environment for microservices, so that the set of microservices changes continuously each time to obtain the best-fit composition.

A recent work proposed by Brito et al¹⁴ outlines the use of topic modeling to assist in the migration of monolithic systems to microservices. They started by extracting lexical/textual terms and structural dependencies from the source code. Afterwards, they adopt the latent Dirichlet allocation (LDA) classifier to identify the topics and their distributions for each component of the monolithic system. Finally, they combine the topic distribution and structural information to cluster the monolithic system components that form microservices.

We observed that microservices identification approaches that use semantic analysis either rely on basic techniques such as Tf-IDF (term frequency-inverse document frequency)²⁸ or use local identification techniques such as topic modeling.¹⁴ Different from existing approaches, we rely on a pretrained Word2Vec model based on Google News to study the semantic relationships between the components inside monolithic systems. Although typed services provide significant information about the nature and capabilities of the identified services, none of these works leveraged the benefits of type-aware services identification as a basis for identifying microservices. Consequently, we propose in this paper a type-based microservices identification approach that relies on static and semantic analysis of the source code of legacy monolithic systems to assist their migration to MSA. As mentioned earlier, we adopted a more refined semantic analysis method that uses the pre-trained Word2Vec model based on Google News, which produces more accurate results on the semantic similarity between different components of the monolithic project and ensures consistency in the context of microservices. Our approach is guided by a taxonomy of service types and performs a vertical decomposition over the system's layers according to the application domains to ensure loose coupling and single responsibility paradigms in MSAs.

3 | PROPOSED APPROACH

We want to decompose monolithic software systems into microservices that are based only on analyses of the source code of object-oriented monolithic software systems. Figure 2 shows our proposed approach for identifying microservices in object-oriented monolithic software systems. It is divided into three main phases:

- The first phase is *Class Typing*, which decomposes a system horizontally into three layers by classifying each class in the system. We propose a method based on ML to predict/assign a label to each class. Labels are *Application* for classes belonging to the Business layer, *Entity* for the persistence layer, and *Utility* for the Utility layer. The layers and their classes are input to the next phase of our approach. We details this first phase in Section 3.1.
- The second phase is *Typed-Service Identification*, which identifies *Application*, *Entity*, and *Utility* services from the classes in each layer. We rely on a graph clustering technique that takes into account the static relationships among classes. We describe this phase in Section 3.2.
- The third phase is *Services to Microservices Mapping*, with generates the microservices. It groups the typed services using soft clustering, which is a type of clustering in which each element can belong to more than one cluster. In our context, an element is a service while a cluster corresponds to a microservice. We identify then the microservices according to the analyses of (1) the relationships between the services and (2) the application domain. Each microservice is composed by one or many *Application*, *Entity*, and *Utility* services. The details of this phase are in Section 3.3.

3.1 | Phase 1: Class typing

This section describes the first phase of our microservices identification approach, which relies on ML to classify each class in a system. At the end of this phase, the model will classify each class of a monolithic software system as *Application*, *Entity*, or *Utility*.

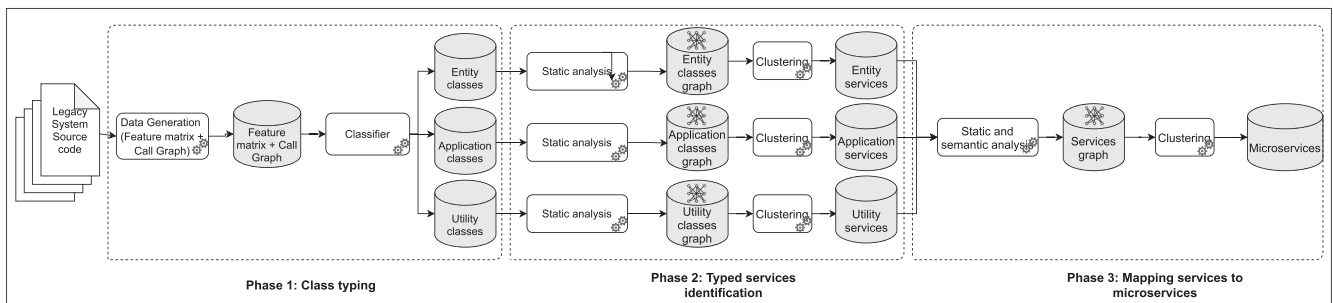


FIGURE 2 Overview of MicroMiner

Mathematically, we looked for the probability distribution $\mathcal{P}(\mathcal{Y}|\mathcal{I}, \mathcal{D})$ of missing labels of classes, where \mathcal{Y} is a class label, \mathcal{I} is the input of the classifier, and \mathcal{D} is the training dataset. We represent our dataset as a direct graph structure $\mathcal{G} = (\mathcal{V}, \mathcal{X}, \mathcal{E})$ where all the classes of the system form the set of the nodes indexes \mathcal{V} , the node feature matrix \mathcal{X} , and the calls among classes form the set of edges \mathcal{E} .

3.1.1 | Call graph generation

As a data generation preprocessing step on the source code, we construct our graph $\mathcal{G} = (\mathcal{V}, \mathcal{X}, \mathcal{E})$ by generating the call graph \mathcal{E} . We parse the source code of the system and build its model using the OMG *Knowledge Discovery Metamodel* (KDM),³⁰ which was defined to represent (legacy) systems at different levels of abstraction and regardless of the used languages and technologies. We use MoDISCO,³¹ an open-source Eclipse plugin that provides an extensible framework, (1) to obtain KDM models from source code in different languages and (2) to visit the KDM models and generate the call graphs. For each system, MoDisco generates the corresponding KDM file in an XML Metadata Interchange (XMI) format⁴, an OMG standard for exchanging metadata information via eXtensible Markup Language (XML). Figure 3 illustrates an example of a KDM file. It contains a representation of the software code elements (e.g., packages, classes, methods, and attributes) and their associations (e.g., inheritance, aggregation, association, and the relationship of containment between packages).

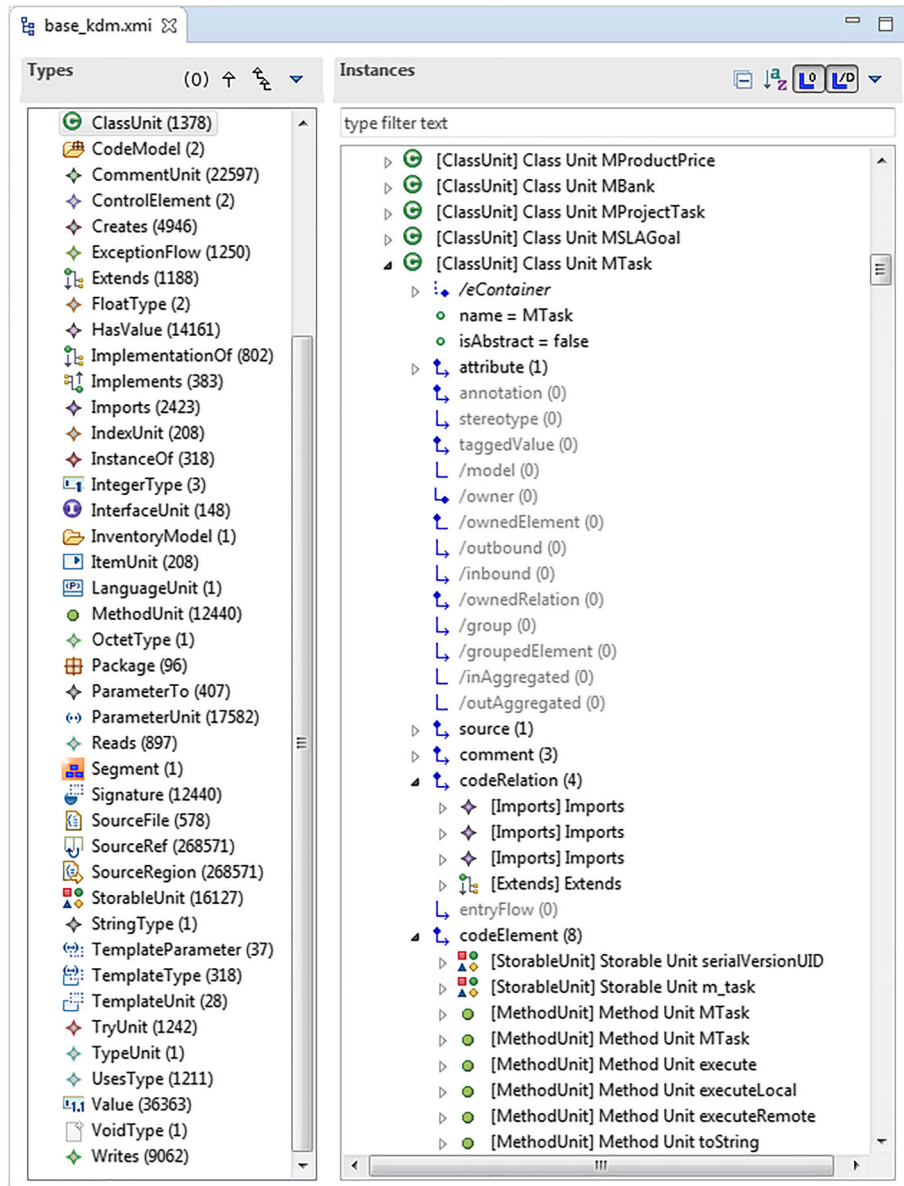


FIGURE 3 Example of a KDM file

Running example.

In the following, we use a running example to illustrate our work. We use a simplified version of the *FXML-POS* monolithic system. *FXML-POS* is an open-source[†] inventory management system in Java. It includes 56 classes. It follows the model-view-controller architecture and provides several features related to ERP, such as purchase management, sales management, and supplier management. We exclude some classes and consider only the 13 classes related to sales, purchases, and products to simplify this running example. Figure 4 shows an excerpt of the running example call graph. This excerpt contains four classes and four relationships. *ProdcutModel* implements *ProductDao*. *ProductController* instantiates *ProdcutModel* and invokes three of its methods. Finally, *ProdcutModel* instantiates *Product*.

3.1.2 | Feature matrix generation

We generate the feature matrix \mathcal{X} by one of the following methods:

- **Metrics and relations:** Our first initiative was to use the same class-level metrics and method-level metrics \mathcal{M} , which were used in a rule-based service-type detection approach,³² to obtain the feature matrix $\mathcal{X}_{|\mathcal{V}| \times |\mathcal{M}|}$. The method-level metrics were used to compute some class-level metrics. For example, we considered the number of incoming/outcoming calls to/from each method in the system to compute the fan-in (the number of incoming calls) and fan-out (the number of outcoming calls) related to each class. The class-level metrics also include the McCabe cyclomatic complexity, the number of “try-catch” in each class and the number of database queries. However, later experimental results, in Table 5, showed that we could not achieve good accuracy for the classification. To improve the accuracy of the results, we added extra features related to the relationships between the classes, such as aggregation, inheritance, and implementation. Therefore, we define the matrix $\mathcal{X}_{|\mathcal{V}| \times |\mathcal{V}|}$ as

$$\begin{cases} \mathcal{X}(i,j) = \mathcal{X}(j,i) = 1, & \text{if classes } i \text{ and } j \text{ are related by an inheritance, aggregation, or implementation relationship} \\ 0, & \text{Otherwise} \end{cases}$$

which improved the results. We further increased accuracy by concatenating the two feature matrices and obtaining $\mathcal{X}_{|\mathcal{V}| \times |\mathcal{M}| + |\mathcal{V}|}$, whose results we detail in Section 4.5.1.

- **CodeBERT:** Our second initiative was to feed the source code into the pre-trained model for programming languages, CodeBERT³³ to generate the node features of the graph. We can think of CodeBERT as a function that maps source-code classes into a feature embedding within a n -dimensional space, where each class has its corresponding n -dimensional vector representation that can be easily fed into the ML classifier.

We rely on CodeBERT to generate the representation of the source code because (1) it has been shown to achieve superior performance in many NLP tasks, (2) it supports several programming languages, such as Go, Java, Python, and Ruby, and (3) the pretrained model of CodeBERT is open source⁸ and easily integrated into our approach.

CodeBERT generates the node feature matrix \mathcal{X} of size $|\mathcal{V}| \times n$, where each row i represents the node feature of the source-code class i with dimension n .

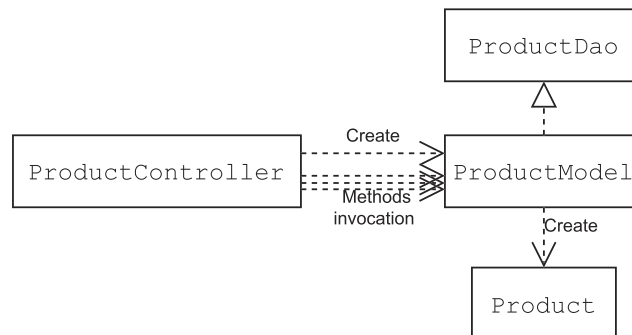


FIGURE 4 Excerpt from the running example call graph

3.1.3 | System classes classification

Given the labeled node feature matrix \mathcal{X} and the call graph \mathcal{E} , we train supervised ML models to classify the source-code classes as belonging to *Application*, *Entity*, or *Utility* services.

The supported classifiers of our approach are:

- **Support vector machine (SVM)** is a supervised classical ML algorithm for classification or regression tasks.³⁴ It is a widely used and relatively simple. The classifier separates data points using hyperplanes with the largest margin. It finds optimal hyperplanes in multidimensional space to separate different classes and classify new data points (an SVM classifier is also called a discriminant classifier).
- **Graph convolutional network (GCN)** is one of the most well-known deep graph neural networks (GNNs).³⁵ It provides a powerful neural network architecture for learning features from graphs by inspecting neighboring nodes. In our approach, we use GCN as a classification model that uses the generated embedding node features \mathcal{X} and the call graph \mathcal{E} to learn how to classify the source-code class to *Application*, *Entity*, or *Utility* service.

In addition to SVM, *MicroMiner* supports different classical ML classifiers such as *decision tree*,³⁶ *K-nearest neighbor (KNN)*,³⁷ *logistic regression*, and *naive Bayes*.³⁸ However, SVM gives the best accuracy among all these algorithms (see Table 6).

Running example

Figure 5 shows the results of the system classes classification. We assigned a color to each type and colored the classes with the appropriate color according to their type. In the running example, there is a single *Utility* class, six *Entity* classes, and six *Application* classes.

3.2 | Phase 2: Typed-service identification

We now identify the services in the monolithic software system by analyzing the static relationships between the classes of the same layer, that is, of the same type. Then, we apply the Louvain community detection algorithm³⁹ on each class layer to group and create *Application*, *Entity*, and *Utility* services.

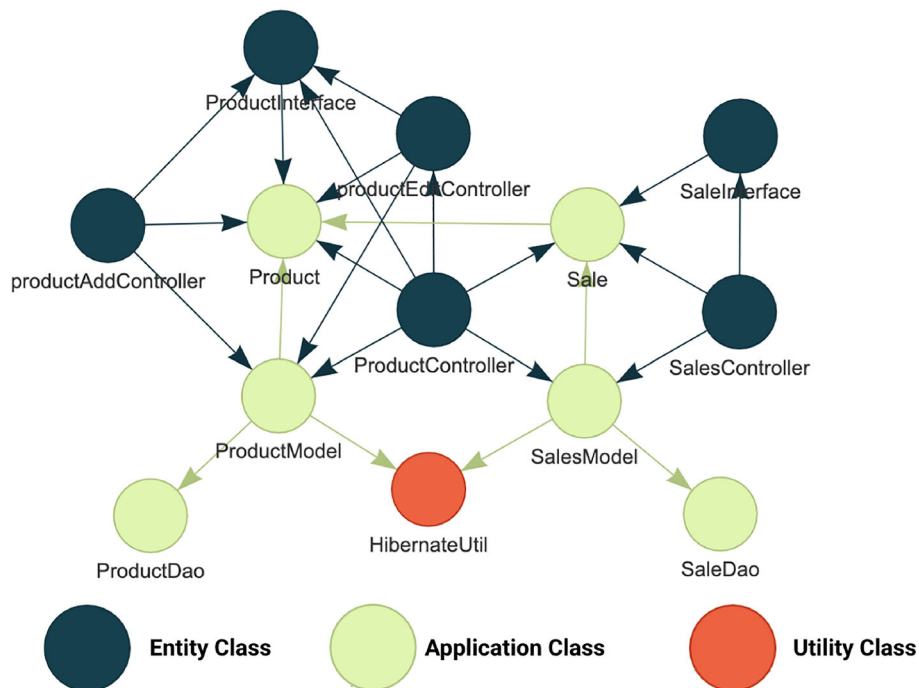


FIGURE 5 Running example: Class typing results

3.2.1 | Static relationship computation

We analyze the static relationships among the constituents of each class in each layer (methods, fields, etc.). A relationship could be a *generalization*, an *aggregation*, or an *association* between classes, for example. We assign a weight to each of them according to their relative importance. The class relationship weights are assigned as follows. We first rank the relationships based on their strength or relative importance. The strength of a class relationship is based on how dependent the classes involved in the relationship are. More precisely, two classes that are strongly dependent on one another are considered tightly coupled and should be in the same cluster. Consequently, if we change one class, this will most likely affect the other class. We classify the class relationships into three types according to their importance and assign a weight between 5 and 100 according to the strength of the relationship. The values of the assigned weights are depicted in Table 1. First, we assign the weight of 100 for highly dependent (strong) class relationships such as Generalization. Second, we assign the weight of 25 to relationships with medium dependencies such as Association. Finally, we assign the weight of 5 to class relationships with low dependencies, such as method invocation between two classes. The total weight between a pair of related constituents is

$$Weight(C_i, C_j) = \sum_{t=1}^T W_t \times NR_t$$

where C_i and C_j are the constituents, T is the number of relationships, W_t the weight of a relationship of type t , and NR_t the number of such relationship between C_i and C_j . We thus obtain three graphs representing each layer.

Running example

After analyzing the relationships among classes in the running example, we compute their weights. Figure 6 shows the call graph with the assigned weights. For example, class *ProductController* instantiates class *ProductModel* and invokes three of its methods. Thus, using the weights in Table 1, we obtain

TABLE 1 Assigned weights of the static relationships

Relationship	Generalization	Aggregation	Implementation	Association	Instantiation	Method invocation
Weight	100	100	100	25	25	5

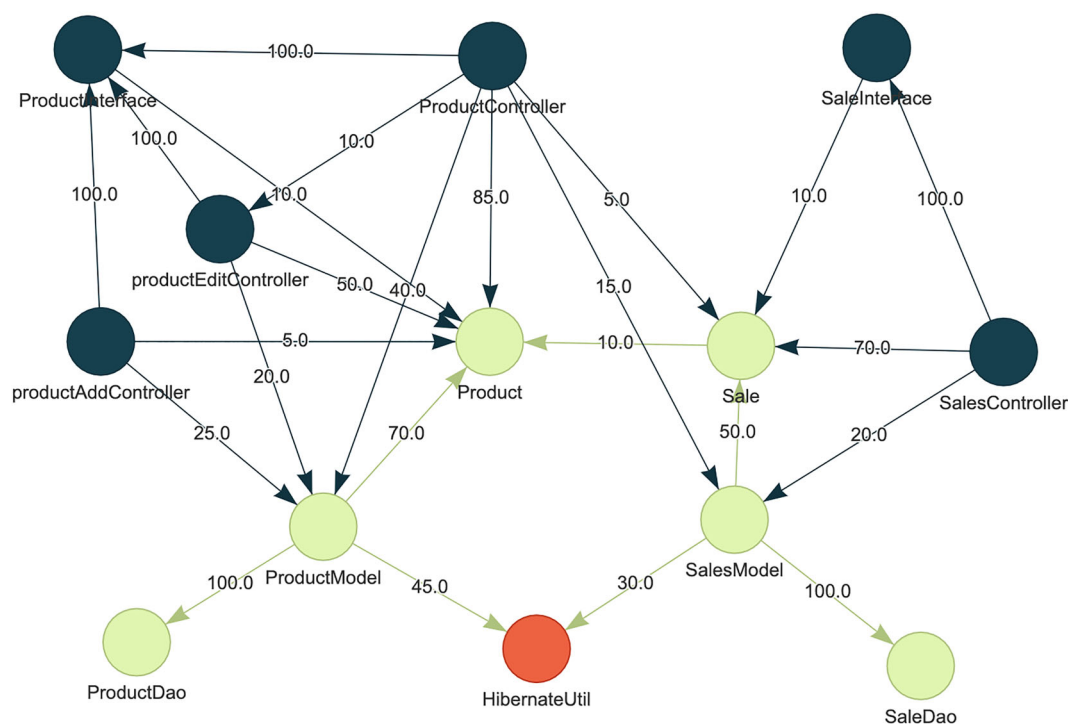


FIGURE 6 Running example call graph with static weights

$$\text{Weight}(C_{\text{ProductController}}, C_{\text{ProductModel}}) = 25 \times 1 + 5 \times 3 = 40$$

3.2.2 | Type-specific services clustering and identification

In each graph of each layer, we group classes into candidate services. The grouping algorithm considers the weights $w(e)$ on the edges when searching for the communities. A large weight $w(e_k)$ on an edge e_k between two nodes A and B implies that the classes A and B belong to the same candidate service.

We rely on the Louvain community detection algorithm³⁹ to derive communities from the graphs. It is an unsupervised algorithm divided into two steps: Modularity optimization and community aggregation. For modularity optimization, it initializes the communities randomly. Then, it relocates each node into a different community until there is no significant increase in modularity. Nodes of each community are collapsed into one node, and the same process is repeated.

To improve the clustering results achieved by the Louvain community detection algorithm, we aggregate modules/communities that are only accessible from some other modules in a same cluster. For example, if a class A is only accessible from classes B and C , with B and C in a same cluster CL , then we put A in the cluster CL and obtain $CL = [A, B, C]$.

Running example

Table 2 shows the results of the second step of our approach on the running example. After applying the clustering algorithm and the refinement process, we obtained five typed services.

3.3 | Phase 3: services to microservices mapping

We generate microservices based on the typed services obtained in the previous phase by analyzing the static and semantic relationships among identified services. We choose each Application service as the core business component of each microservice and merge related Entity and Utility services to form the final microservices, as follows.

3.3.1 | Static relationship computation

We consider the typed services that have been identified in the previous phase as our unit of work, which are no longer the source-code classes. We are now interested in the relationships between services, which we compute in two steps.

First, we compute the weight of the direct relationships between two services. We consider all calls between the services by computing the sum of the weights of the incoming and outgoing calls between the services using the previously generated call graphs. We create an undirected, edge-weighted graph $\mathcal{G} = (\mathcal{E}, \mathcal{V})$, in which each node $vi \in \mathcal{V}$ corresponds to a service $si \in \mathcal{S}$ and each edge $e_i \in \mathcal{E}$ represents the relationships between two services. Each edge e_i has a weight that shows how strong the link between two services is.

Second, we apply Floyd-Warshall algorithm⁴⁰ to create the adjacency matrix between all services of the graph \mathcal{G} by finding the shortest path between each node of the graph. Thus, we obtain the static distance between two services.

Running example

Figure 7 shows the weights calculated between the typed services resulting from this step.

TABLE 2 Running example: Typed-services

Service	Utility Service 1 (US1)	Entity Service 1 (ES1)	Entity Service 2 (ES2)	Application Service 1 (AS1)	Application Service 2 (AS2)
Classes	HibernateUtil	Product	Sale	ProductController	SaleInterface
		ProductDao	SalesModel	ProductInterface	SalesController
		ProductModel	SaleDao	productEditController	
				productAddController	

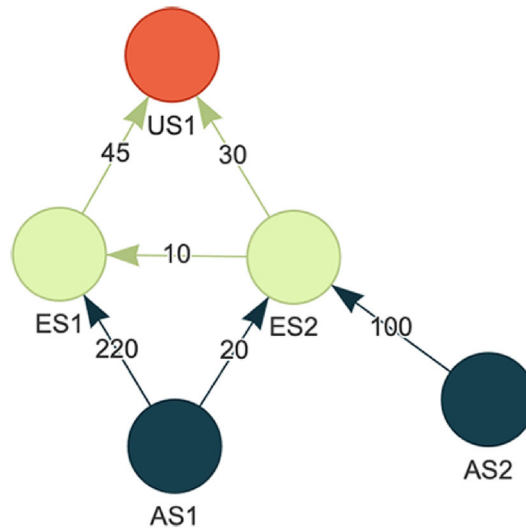


FIGURE 7 Running example: Static weights between services

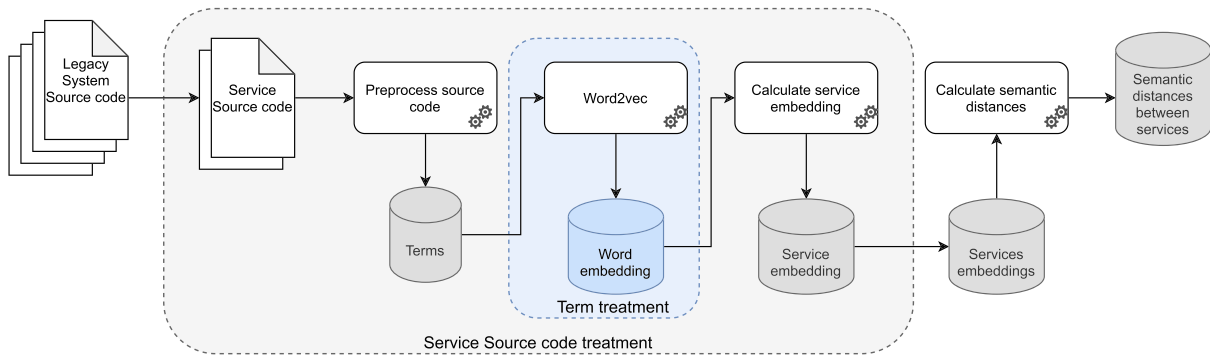


FIGURE 8 Semantic analysis pipeline

3.3.2 | Semantic relationship computation

Each microservices must have a single responsibility. Consequently, we perform a domain-related service decomposition to generate isolated functional microservices that belong to a specific bounded context.²² We analyze the semantic relationships between the identified services to group them according to their domain. Figure 8 shows that the semantic analysis of the source code is used to extract the semantic coupling between the services in four steps.

First, we perform a preprocessing step, which involves: (1) Tokenizing the source code in which the text is separated at each blank space, (2) removing any terms related to the programming language (`public`, `private`, etc. in Java), (3) separating the composed terms using naming conventions like camel case and underscore,⁴¹ and (4) lemmatizing terms into their base forms.

Second, we encode each term in a numerical representation to facilitate their manipulation. We use word embedding, which is one of the most popular representations of words. It transforms terms into a numerical vector and captures the context of a word in a document, the semantic and syntactic similarity, the relationship with other words, etc. We use the pre-trained, Google-news-based Word2Vec that contains three million terms.⁴

Third, we determine the vector representation of the services, that is, their embeddings. We use a simple yet efficient technique to obtain the mean of all term vectors related to a service. Indeed, a study showed that, for the sentiment text analysis task, using the average of term embeddings to obtain a document embedding yields similar results compared to more complex techniques.⁴²

Finally, we compute the cosine distance between the service embeddings to obtain the semantic distances between each pair of services.

3.3.3 | Microservices generation

To assure both the contextual consistency and high cohesion of the microservices, we rely on both static and semantic weights. To find the final weight to qualify how strong the relationship between the services/clusters is, we combine both weights and perform a unit-based normalization on the static and semantic weights to adjust their values to $[0, 1]$. Then, we compute the final weight w_{ij} by balancing the other two weights $w_{Static_{ij}}$ and $w_{Semantic_{ij}}$ using two parameters α and β :

$$w_{ij} = \alpha \times w_{Static_{ij}} + \beta \times w_{Semantic_{ij}}$$

An expert must specify these parameters as they depend on the systems, for example, in the case of a system in which components are poorly named, β could be reduced in favor of α to reduce the dependence on the semantic analysis in favor of the static relationships/analysis.

After calculating the final weights between the typed services, we cluster these services to compose the microservices. In this step, we consider each application service as the central element of each microservice c_i , because they “own” the functionality. We use the fuzzy C-means clustering (FCM) algorithm⁴³ because it is one of the most widely used fuzzy clustering algorithms. The FCM algorithm returns membership scores representing the degrees of membership of data points x_i to each cluster c . This membership score ms_{ij} is calculated by

$$ms_{ij} = \frac{1}{\sum_{k=1}^c \left(\frac{\|x_i - c_j\|}{\|x_i - c_k\|} \right)^{\frac{2}{m-1}}}$$

where m is the fuzziness parameter and k is the number of clusters.

We apply the FCM algorithm with one iteration because we do not want to change the cluster centers. Then, we must specify a threshold at which we consider whether a service belongs or not to a microservice. Again, an expert should specify this threshold, especially because it may vary from one system to another.

Our approach respects the loose coupling principle of microservices. If two microservices are dependent on a same service, we duplicate this service into each microservice to prevent any dependency between them. For example, in the case of a retail management software, the Entity service *Product*, which manages product data, is used by the microservices *Sales* and *Product Restocking*. We include this Entity service in both microservices.

Running example

Finally, we obtain two microservices for our running example. $MS1 = \{US1, ES1, AS1\}$ and $MS2 = \{US1, ES1, ES2, AS2\}$. The Utility service *US1* is duplicated in both microservices. The *Product* Entity service *ES1* is also duplicated in both the *Sales* and *Product* microservices, due to its strong coupling with the two services.

4 | EMPIRICAL EVALUATION

The overall quality of the generated results of our approach has been first evaluated by the fifth author, who is an expert in microservice-based systems. He thoroughly analyzed the systems and the generated microservices decomposition by our approach to making sure that the identified microservices embed cohesive classes and belong to bounded contexts. Despite the positive feedback of our expert regarding the generated results, we further validate our approach by (1) comparing our results with two state-of-the-art microservices identification approaches and (2) relying on qualitative metrics to further evaluate the quality of the identified microservices.

This section presents the experiments that we conducted on four case studies to validate our approach both quantitatively against a ground-truth and qualitatively to use evaluation metrics. We provide further details on the setups and outputs of the three phases of our approach: the Class Typing phase, the Typed-service Identification phase, and the Microservices Mapping phase.

4.1 | Case studies

To evaluate approach, we sought legacy systems that (1) are open source and available online, (2) whose initial architecture is monolithic so we discarded service-oriented systems, and (3) are object-oriented because we rely on classes as the basic entities in our approach. However, we faced a lack of availability of systems that meet these three criteria. Therefore, we conducted our studies on only four systems of different sizes that we could find in the literature and that were used to validate similar state-of-the-art approaches. Thus, we considered Compiere, a large ERP

system that contains 1,042 classes; JForum, a medium-sized system that contains 271 classes; and PetClinic and POS that contain less than 100 classes.

4.1.1 | Compiere

Compiere is one of the few large, Java, open-source *legacy* ERP systems. It was first introduced by *Aptean* in 2003.[#] It provides businesses, government agencies, and nonprofit organizations with flexible and low-cost ERP features,^{||} such as business partners management, warehouse management, purchasing and sales order management (quotes, book orders, etc.). We use *Compiere* v3.3 because (1) it is the first stable release of the system, (2) it was released more than 15 years ago, and (3) it is not based on microservices.

4.1.2 | FXML-POS

This system was presented as our running example in Section 3.1.1 as well as used as a case study to validate our approach. We use *FXML-POS* because it offers several features and it is not microservice-oriented.

4.1.3 | PetClinic

PetClinic is an open-source** Java-based veterinary clinic management system that allows veterinarians to manage information about pets and their owners. It is based on the model-view-controller architecture and has 52 classes. This system provides several features such as pet management, owner management and, visits management. We chose to use *PetClinic* because there is also a new version of the system built using the microservices architecture that serves as the basis for creating our ground-truth.

4.1.4 | JForum 3

JForum is an open-source[†] discussion board system implemented in Java. We worked with the last version that includes nearly 300 classes. It is based on the model-view-controller architecture and provides several features, such as users management, message system, topic management, etc. We use *JForum* because (1) it is not microservice-oriented, and (2) it was previously used to validate several state-of-the-art monolith-to-service migration approaches.^{12,44,45}

4.2 | Ground-truths

To assess the reliability of our approach, we need two kinds of ground-truths for each system. The first type of ground-truth is related to the services and their types. This type of ground-truth will be used to validate the two first phases of our approach (class typing and typed-service identification). Also, we need a second type of ground-truth which is related to the microservices in the monolithic systems. It will be finally used to evaluate the microservice mapping. We asked two independent PhD students to identify services and microservices in *Compiere* and *FXML-POS* systems. They relied on several artifacts to build the ground-truth architectures manually by (1) analyzing and understanding the systems and (2) extracting the reusable parts that could become services/ microservices. To recover their designs and to visualize class dependencies, they used *Understand*^{‡‡} integrated development environment. Additionally, they generated views of their call graphs that we make available online.^{§§} They also reviewed extensively the system documentation as well as their source code to have the best possible understanding and accurately identify both services and microservices that can be integrated into the targeted SOA-based system and in a microservice architecture. Finally, they annotated the services manually according to their types. Table 3 shows the statistics of the ground-truth decomposition.

4.3 | Evaluation metrics

To evaluate the quality of the identified microservices, we follow the work of Jin et al¹² and consider the following metrics that are related to functionality independence and modularity aspects.

TABLE 3 Overview of the ground-truths

Legacy system	# System-classes	# Entity services	# Application services	# Utility services	# Microservices
Compiere	1,042	358	30	85	92
FXML-POS	55	9	10	3	9
PetClinic	52	7	7	4	7
JForum 3	271	31	73	19	61

4.3.1 | Independence of functionality

The functionality independence refers to the external independence, that is, the independence and consistency of the functionality that the microservice provides to its external users, as defined by the single responsibility principle (SRP). These metrics are calculated using the interfaces of a microservice. Typically, an interface is a class that exposes functionality as an endpoint. The methods for each interface are considered as operations. **IFN (Interface Number)** *ifn* measures the number of published interfaces of a microservice. The smaller the *ifn*, the more likely the microservice has a single responsibility. The *IFN* is the average of all *ifn*.

CHM (Cohesion at Message level) *chm* measures the cohesion of interfaces published by a microservice at the message level. This is achieved by calculating the similarity between two message-level operations based on parameters and return values. A higher *chm* represents a higher cohesiveness of the microservice. The *CHM* is the average all *chm*.

CHD (Cohesion at Domain level) *chd* measures the cohesion of interfaces published by a microservice at the domain level. This metric measured very similarly to *chm*, but instead of using only the message terms, all domain terms contained in the operation signature are taken into account. A higher *chd* represents a higher cohesiveness of the microservice. The *CHD* is the average all *chd*.

4.3.2 | Modularity

Modularity evaluates the cohesion of microservices in their internal interactions and the looseness of interactions between microservices. To evaluate the modularity of service candidates, we use the following metrics Quality of structural modularity (SMQ) and Quality of conceptual modularity (CMQ). **SMQ (Structural Modularity Quality)** measures the quality of modularity from a structural point of view. The higher the SMQ, the more modular the service is. The SMQ calculation is composed of two terms: The first measures the structural cohesion of a service (intraconnectivity) using the number of edges within a service, and the second measures the coupling between services (inter-connectivity) based on the number of edges between services.

CMQ (Conceptual Modularity Quality) measures the quality of modularity from a conceptual perspective, like the SMQ. CMQ is composed of two terms, the only difference is that in the CMQ definition, an edge between two entities exists if the intersection between the set of textual terms of the entities is not empty. The higher the CMQ, the better.

4.4 | Experimental setup

In the following, we present the experimental settings at each phase of the *MicroMiner* architecture.

4.4.1 | Class typing

For the datasets, as we mentioned in Section 3.1, we consider the graph representation of the four legacy systems as our dataset form. For all the classification models, we use 80% of the graph nodes for training (\mathcal{V}_{train}). The rest 20% of the nodes are used for testing (\mathcal{V}_{test}), except for GCN model is splitted into 10%, for validation (\mathcal{V}_{valid}) and 10% for testing. Now, we present the setup of the experiment for our classification models categories: **Deep GNNs**. We considered a two-layer graph convolutional network (GCN) for the semi-supervised classification of the nodes of a graph. At the training, the GCN classifier takes as input: (1) The adjacency matrix \mathcal{A}_{train} of the training graph \mathcal{G}_{train} of the legacy system to migrate. The nodes of the call graph $v_i \in \mathcal{V}_{train}$ correspond to classes $c_i \in \mathcal{C}$ from the system to analyze. (2) The feature matrix \mathcal{X}_{train} that embed information about the system-classes. We experimented with several representations for the feature matrix (e.g., CodeBERT embeddings and Method-level and class-level metrics) as described earlier in Section 3.1.2. We validate the GCN classifier using the validation components (\mathcal{V}_{valid} , \mathcal{E}_{valid} , \mathcal{X}_{valid}),

and we test with the test components ($\mathcal{V}_{test}, \mathcal{E}_{test}, \mathcal{X}_{test}$). Table 5 summarizes the results of GCN experiments with different feature representations on all systems. Given GCN shows the best performance with CodeBERT embedding on *Compiere* and *FXML-POS*, we state the performance results of GCN on *PetClinic* and *JForum 3* when using CodeBERT embeddings only.

Classical ML. As we showed in Section 3.1.3, we used several classical ML classification models. To classify the graph nodes (i.e., system-classes) based on their features. All these classical models rely on the code embeddings that CodeBERT generates. With SVM, we use the linear kernel function. For *KNN classifier*, we set $k=5$ for the number of nearest neighbors to be considered in the voting process. We also experienced with *Decision Tree* and set the tree's maximum depth to 2. For *Logistic Regression*, we used *lbfgs* as the optimization algorithm. The final algorithm we applied was Gaussian Naive Bayes. In Table 6, we present the performance measures of all classifiers in our four use cases. We report the accuracy that gives the overall correctness of the models, that is, the fraction of total samples that were correctly classified by the classifier and the F-measure for each class (Utility, Entity, and Application).

4.4.2 | Typed-service identification

After predicting the type of each class on our different case studies, we constructed three graphs per system: the Utility graph, the Entity graph, and the Application graph, as explained in Section 3.2.1. Then, we applied the Louvain algorithm for community detection to cluster and identify Utility, Entity, Application services. For the Louvain algorithm, the only parameter to specify was the *resolution* that controls the communities sizes. We have set it to its default value of 1 for all experiments.

4.4.3 | Microservices mapping

As mentioned in Section 3.3.3, we calculate the final weight to qualify the strength of the relationship between the services/clusters by combining the static and semantic weights based on two parameters α and β . Table 4 shows the considered values of these parameters for the different systems. For *Compiere*, the code components, variables and method names are poorly named, so we reduced the value of β in favor of α to reduce the dependence of the semantic analysis on the static relation. For the other three systems, we gave equal weights to semantic and static relationships as the naming employed in the code was expressive and meaningful.

In order to generate microservices, we cluster the typed services using Fuzzy C-means clustering algorithm. To calculate the membership score, we need to specify the number of clusters and the fuzziness m parameter. In our case, the number of clusters is equal to the number of *application services*. We used $m=2$ for all systems because it is the most used value.⁴⁶ Moreover, by adopting this value, we avoided getting large microservices and thus increased the cohesion of microservices while maintaining a low coupling value.

4.5 | Results and evaluation

In the following, we present the results of each phase of the *MicroMiner* architecture.

4.5.1 | Class typing results

We present in Table 5 the GCN accuracy with respect to different feature matrix formalisms. The results show that the most accurate classification result is achieved by using the code embeddings generated through CodeBERT with an average accuracy of 74%.

As shown in Table 6, the SVM classifier achieved the highest accuracy across the four systems with 86% for *Compiere*, 92% for *FXML-POS*, 87% for *PetClinic* and 82% for *JForum*. However, we observe that the F-measure of some classes is slightly higher when using other classifiers, albeit with a small difference. Based on these results, we selected SVM in the classification phase of *MicroMiner* to predict the class types of the system because it outperformed GCN and other classifiers and showed overall better results.

TABLE 4 Total weight parameters

System	Compiere	FXML-POS	PetClinic	Jforum
α	0.8	0.5	0.5	0.5
β	0.2	0.5	0.5	0.5

TABLE 5 GCN experiments results for all systems with respect to different generated feature matrix

Feature matrix	Feature matrix dimension	Compiere Accuracy	FXML-POS Accuracy	PetClinic Accuracy	JForum Accuracy
Method-level and class-level metrics	$ V \times 6$	51%	59%	N/A	N/A
Relations	$ V \times V $	65%	62%	N/A	N/A
Method-level and class-level metrics+relations	$ V \times V + 6$	58%	62%	N/A	N/A
CodeBERT embeddings	$ V \times 768$	72%	75%	76%	73%

TABLE 6 Source-code class classification results with the classical ML models using CodeBERT embedding of the four legacy systems

Legacy system	Quality metrics	Decision tree	SVM	KNN (k = 5)	Logistic regression	Naive Bayes
Compiere	Accuracy	73%	86%	81%	83%	52%
	Class Application F1-score	54%	68%	69%	61%	47%
	Class Entity F1-score	85%	93%	91%	92%	64%
	Class Utility F1-score	13%	62%	37%	64%	35%
FXML-POS	Accuracy	84%	92%	66%	84%	84%
	Class Application F1-score	86%	93%	75%	86%	86%
	Class Entity F1-score	87%	97%	58%	90%	87%
	Class Utility F1-score	50%	50%	0%	0%	50%
PetClinic	Accuracy	79%	87%	74%	77%	69%
	Class Application F1-score	67%	81%	58%	81%	77%
	Class Entity F1-score	82%	89%	71%	73%	77%
	Class Utility F1-score	50%	50%	0%	0%	0%
JForum	Accuracy	79%	82%	69%	76%	62%
	Class Application F1-score	69%	83%	75%	66%	53%
	Class Entity F1-score	81%	87%	58%	80%	69%
	Class Utility F1-score	10%	30%	18%	22%	18%

4.5.2 | Typed-service identification results

We used our ground-truth architecture for each system to quantitatively validate the typed services. We measured precision, recall, and F-measure for the identification of each service type and reported the results in Table 7. We obtained architecturally significant typed services with a total precision of 68.3%, a recall of 83.9%, and an F-measure of 75.2% for *Compiere*. We also achieved a precision of 86.3%, a recall of 86.3%, and an F-measure of 86.3% for *FXML-POS*. For *PetClinic*, we got a precision of 70%, a recall of 66%, and an F-measure of 68%. For our fourth system *JForum*, we obtained a precision of 83.1%, a recall of 60.1%, and an F-measure of 69.7%.

The performance of this phase depends on the results of the previous classification phase because we consider the classes of each type independently. For example, we had in *Compiere* a precision of 49.2% for the identification of Application services. We missed some Application services because, in the previous phase of class typing, we had a classification accuracy of 68%. We missed some classes of type Applications to cluster into Application services. Also, in *FXML-POS*, we could not identify some Utility services for the same reasons.

To obtain better results for identifying typed-service, developers could refine the classification when applied to their systems.

4.5.3 | Microservices mapping results

In this section, we analyze and validate the generated microservices. We used our ground-truth architectures to calculate the precision, recall, and F-measure of our approach. We also measure and validate the quality of *MicroMiner* using five quality metrics. In addition, we compare our results with two static-based microservices identification approaches: *ServiceCutter*,¹³ which is a heuristics-based microservices identification approach, and another state-of-the-art microservices identification tool proposed by Brito et al.,¹⁴ which is based on topic modeling techniques.

TABLE 7 Overview of typed-service identification results

Legacy system	Service type	Precision	Recall	F-measure
Compiere	Application	49.2%	81.3%	61.4%
	Entity	74.3%	90.7%	81.7%
	Utility	68.2%	83.9%	75.2%
	Total	68.2%	83.9%	75.2%
FXML-POS	Application	83.3%	100%	90.9%
	Entity	88.8%	88.8%	88.8%
	Utility	100%	33.3%	50%
	Total	86.3%	86.3%	86.3%
PetClinic	Application	75%	85%	80%
	Entity	83.3%	71.4%	76.9%
	Utility	50%	75%	60%
	Total	70%	66%	68%
Jforum	Application	89.8%	72.6%	80.2%
	Entity	77.2%	54.8%	64.1%
	Utility	50%	21%	29.5%
	Total	83.1%	60.1%	69.7%

TABLE 8 Comparison results of microservices identification approaches

Legacy system	Approach	Precision	Recall	F-measure
Compiere	ServiceCutter	5%	21.7%	8.1%
	Topic modeling-based	22.5%	29.3%	25.4%
	MicroMiner	75.2%	72.8%	73.9%
FXML-POS	ServiceCutter	7.8%	33.3%	12.6%
	Topic modeling-based	29.4%	55.5%	38.4%
	MicroMiner	72%	88%	80%
PetClinic	ServiceCutter	8.5%	42%	14.2%
	Topic modeling-based	36.3%	57.1%	44.3%
	MicroMiner	71.4%	71.4%	71.4%
Jforum	ServiceCutter	4.7%	11.9%	6.7%
	Topic modeling-based	44.1%	45.2%	44.6%
	MicroMiner	54%	76.1%	63.1%

4.5.4 | Quantitative evaluation

We applied *MicroMiner* on *Compiere*, *FXML-POS*, *PetClinic* and *Jforum* to show its practical accuracy in identifying microservices in existing systems. We also applied on these systems the two state-of-the-art microservices identification approaches, *ServiceCutter* and the *Topic modeling-based* approach. We measured the precision, recall, and F-measure for each system and reported the results in Table 8. We found that *MicroMiner* identified architecturally relevant microservices with 68.15% precision, 77% recall, and 72.1% F-measure.

Also, while we identified 92 microservices in the ground-truth microservice-based architecture of *Compiere*, *MicroMiner* identified 89 microservices, among which 67 were correctly composed. We also obtained for the same system a precision of 75.2%, a recall of 72.8%, and an F-measure of 73.9%. For *FXML-POS*, while we identified nine microservices in the ground-truth, *MicroMiner* identified 11 microservices, with nine of them correctly identified. We obtained a precision of 72%, a recall of 88%, and an F-measure of 80%. Table 9 shows the number of duplicated services. We noticed that this number is not large because we selected a relatively small fuzziness parameter.

In general, the correctly-identified microservices are built around accurately classified Application services because we choose to consider Application services as the central component of each microservice. The correctly identified microservices in *Compiere*, for example, are related to bank-statement management, account management, partners, warehouses, orders, invoice management, etc.

TABLE 9 Duplicated services per system

System	Compiere	FXML-POS	PetClinic	Jforum
# duplicated services	22	5	4	12

TABLE 10 Overview of the generated microservices quality

Legacy system	Approach	IFN	CHM	CHD	SMQ	CMQ
Compiere	ServiceCutter	4.1	0.23	0.21	−0.17	−0.21
	Topic modeling-based	2.1	0.46	0.54	−0.02	0.01
	MicroMiner	3.4	0.73	0.53	0.11	0.09
FXML-POS	ServiceCutter	2.9	0.54	0.33	−0.12	0.01
	Topic modeling-based	2.5	0.47	0.79	−0.03	0.03
	MicroMiner	2.3	0.69	0.72	0.09	0.05
PetClinic	ServiceCutter	2.3	0.42	0.53	0.02	−0.03
	Topic modeling-based	2.1	0.43	0.72	0.03	0.02
	MicroMiner	1.9	0.75	0.67	0.05	0.03
Jforum	ServiceCutter	3.1	0.34	0.24	−0.01	−0.03
	Topic modeling-based	2.3	0.43	0.62	0.03	0.05
	MicroMiner	2.8	0.67	0.56	0.04	0.06

The incorrectly identified microservices were mainly coarse-grained: they contained a large number of classes. When we analyzed these microservices, we found that the related Application services contained tightly coupled classes but covered more than one domain/business functionality. Thus, poor identification of Application services inside a system led to poor identification of the corresponding microservices. For example, one of the identified Application services from *Compiere* aggregated classes related to both projects and payments, which led to the generation of a large microservice that did not respect the single responsibility principle.

Furthermore, when services were not correctly typed, the quality of the identified microservices suffered. For example, in *Compiere*, some classes related to *data migration* should have been labeled as Utility-related classes. However, *MicroMiner* classified these classes as Application-related, which led to the identification of incorrect Application services and, therefore, to incorrect/poor microservices.

Furthermore, our approach outperformed *ServiceCutter* and *Topic modeling-based* approach in terms of identification precision, recall and F-measure. Indeed, *ServiceCutter* generated poor results on all systems that we have considered in our experiment. We observed that this approach produced a large number of services of unbalanced size. More precisely, we observed that *ServiceCutter* generates in general a few large services and many small ones containing one or two classes. This affects considerably the microservices identification performance because the generated services are not architecturally relevant. For the *Topic modeling-based* approach, the generated services were partially acceptable. However, many classes that are not directly relevant to the service context, for example, generic classes like *EntityUtils* or *BaseEntity* are always incorrectly mapped because they do not have a clear domain connection. For example, when applying the *Topic modeling-based* approach on *PetClinic*, generic classes such as *EntityUtils* and *BaseEntity* are always incorrectly mapped to their corresponding microservices as such classes provide cross-cutting functionalities that could take part in several possible service domains. Moreover, such approach highly relies on identifying microservices according to the semantic relationships between the system components. In contrast, our approach considers both static and semantic relationships between the system's classes and relies on a more structured and two-phased clustering approach that (1) group classes according to the types of their provided functionalities, and then (2) group the generated clusters according to their domain.

4.5.5 | Qualitative evaluation

To evaluate the quality of the identified microservices, we conducted a metrics-based and a content-based evaluation.

A- Metric-based Evaluation

Table 10 shows the values of the microservice quality metrics obtained using *ServiceCutter*, *Topic modeling-based approach* and *MicroMiner* across the four systems. IFN quantifies the interfaces that exposed by a given microservice, that is, any class that exposes functionality as an

endpoint. Hence, a smaller IFN represents a higher likelihood of a service having a single responsibility. The median is almost three in our case. Although our IFN values are relatively low, they are not optimal ($IFN = 1$) because the legacy systems are improperly designed. For example, in *FXML-POS*, we found *ProductController*, *ProductAddController*, and *ProductEditController* that should have been merged into a single class. The *Topic modeling-based* approach gives better IFN values for *Compiere* and *Jforum*.

CHM represents message-level cohesiveness. For *MicroMiner*, it has a mean around 0.7, which is a positive statement regarding the independence of microservices. Our approach outperforms the two others in most cases.

CHD quantifies the cohesion at the domain level. For *MicroMiner*, it has a mean around 0.7, which indicates that the generated microservices respect the bounded-context principle, thanks to the semantic analysis in our process. For *Compiere*, the value is low because the elements of the system are poorly named. The *Topic modeling-based* approach gives better CHD values for *Compiere* and *Jforum*.

SMQ and **CMQ** represent the modularity of the microservices. We mainly calculate the differences between cohesion and coupling and the values are bound between -1 and 1 . *MicroMiner* exhibits distinctively positive differences from the cohesion to coupling across all systems and medians values are roughly around 0.07 and 0.05, respectively. We therefore conclude that the generated microservices have a good modularity: They are loosely coupled and strongly cohesive. Our approach outperforms the two other approaches.

B- Content-based Evaluation

For the content-based evaluation, we detail the results of applying *MicroMiner* on *FXML-POS* to identify relevant microservices in the system. We take the example of sales and supplier management in *FXML-POS*. We qualitatively study the microservices related to these business functionalities while we detail how *MicroMiner* helps practitioners to identify such microservices.

The initial classification step of our approach predicts the layer to which each class in the system belongs (i.e., Utility, Entity, or Application). Based on this classification, we apply the Louvain community detection algorithm on each layer to identify the clusters that correspond to the typed services. Finally, based on the Application services, we perform a vertical clustering over the layers to merge Utility, Entity, and Application services that belong to the same domain to form microservices.

First, we divide the system into three layers (Utility, Entity, and Application service layers). In the Utility layer, we have some cross-cutting functionalities, like printing, logging, and Hibernate-related functionalities. In the Entity layer, we find the DAO classes that support CRUD actions on the data and the models that represent the data: classes *SaleDAO*, *SupplierDAO*, *SaleModel*, and *SupplierModel*. In the Application layer, we find the classes that provide functionalities related to supplier and sales management. Second, we perform a clustering to create the services in each layer and to get the types of these services. Third, we choose the Sales and Vendors Application services as central components of two microservices and perform a fuzzy clustering to create the microservices. Finally, we obtain two microservices: The first is related to sales management, which comprises the Serialization Utility service, Sales entity service, and Sales application service. The second microservice is composed of Hibernate Utility service, Vendor entity service, and Vendor application service.

Thus, we could identify with *MicroMiner* architecturally-relevant microservices in the different systems. We believe that our approach can assist practitioners in identifying candidate microservices because it relies only on the static analysis of the system source code to migrate and automate the microservices identification process with acceptable precision and recall.

5 | DISCUSSIONS

We will describe in this section the threats to the validity of our approach and the recommendations that we derived based on our observations.

5.1 | Threats to validity

5.1.1 | Internal validity

Our microservices identification approach and its validation depend on several metrics and thresholds that threaten the internal validity of our results. To mitigate these threats, we used different algorithms and threshold values.

The results of our approach were qualitatively evaluated by the fifth author because of his expertise in developing microservices. However, we must accept a threat to the validity of this validation because this author participated in some meetings discussing the work in general and the approach in particular. Therefore, this author is not entirely independent and could have been biased. Yet, we accept this threat because (1) he is an expert at developing and studying microservices and (2) finding an expert who is willing to validate our approach on different systems is difficult. Besides, we provide all results and validation for others to verify the validation or perform it again independently. Finally, we mitigate this validation bias by (1) comparing our results with two state-of-the-art microservices identification approaches and (2) relying on qualitative metrics to further evaluate the quality of the identified microservices.

5.1.2 | Construct validity

The quality of the legacy source code may have an impact on the results of the microservices identification. Legacy monolithic systems may embed some poor design practices that may limit the accuracy of static-based microservices identification approaches. To mitigate this threat, we rely on the analysis of both static and semantic relationships between the system's elements on different levels (in the class level and service level).

We relied on the generation of ground truths to validate quantitatively the microservices identified by our approach. Two independently PhD students extensively analyzed the systems to obtain two sets of ground-truths: The first set was related to the services and their types, while the second set was for the microservices. The generated ground-truths by both students were very similar. They reconciled the differences between the generated ground-truths through discussions and end-up with common ground-truths used to validate our microservices identification approach. We are aware that there is no single "correct" microservice-based version of the analyzed legacy systems. To reduce the bias, we may ask the projects' owners or software developers to provide the ground-truths. However, getting in touch with such experts is challenging. Furthermore, producing an accurate ground-truth microservice based architecture for a monolithic system is an arduous and time-consuming task. We do not expect that the projects' owners or software developers will accept to take considerable time away from their daily obligations to build the ground-truths on our behalf. However, we do not exclude such tasks and will try to involve software developers in our experiments as future work.

5.1.3 | External validity

In order for the classification model to classify accurately, there must be a similar distribution of data on which the model makes predictions as the data on which the model was trained. The classification model in the class typing phase was trained using only four legacy systems; hence, our results could not be generalized. A small portion of the labeled classes is requested to adapt the already trained model to mitigate this threat. Finally, the generalization of our training could be enhanced if we label and analyze a large set of monolithic systems. However, building such a dataset is challenging as we have to (1) select hundreds of systems pertaining to different domains and relying on different architectures, (2) analyze and review the source code of these systems, and (3) manually classify each of the classes inside the selected systems. Considering all these challenges, we aim as future work to train the ML classifier using a large set of monolithic systems and then use the trained ML model to predict the classes of new monolithic systems.

5.2 | Discussion and recommendations

In the first phase of our approach, the use of the ML classifier may not be necessary when the different types of classes in the system are well packaged into their respective source files. However, our target in this paper is legacy monolithic systems to be migrated to microservices. These legacy systems generally embed several poor design problems and packaging issues that may hinder their maintenance. For instance, different types of classes are not necessarily packaged into their respective source files. This is the case of *Compiere*, for example, where we found different classes pertaining to different types in some packages. Also, when the system is too large and complex, it is not easy to manually check if each class in the system is correctly packaged according to its type. To deal with all these challenges and to make the approach more generalizable, we rely on our ML classifier to detect the types of the classes that will be then mapped to their corresponding typed services. In our case, we use entity, application, and utility services, which are considered fine-grained services because they are specialized services that cover a specific domain context and one role (data management, business logic, etc.) at a time.⁴⁷ However, MSAs are composed of loosely coupled specialized components that often operate independently of each other. Therefore, a microservice should encapsulate all elements to perform a specific business functionality. In our case, a microservice is composed of different service types (i.e., entity, application, and utility) to perform a specific business functionality. Our approach works best on systems with a relatively good architecture and design. Although the quality of the architecture is not very impactful (as our approach does not consider packages, etc.), the quality of the system design indeed may impact our approach. In fact, if a system is poorly designed and has code smells that reduces the separation of concerns within/between classes, our approach would fail to identify (relevant) microservices like any typical static-based service identification approach. Our approach depends on several metrics and thresholds. For the fuzziness parameter m , we recommend using values in the range [1.5, 2.5] to allow some classes to be duplicated in several microservices. By limiting the fuzziness parameter m to 2.5, we avoid obtaining large numbers of duplicated classes. Thus, we increase the cohesion of the microservices while maintaining a low coupling value. For parameters that balance static and semantic weights, their selection depends on the quality of the initial system source code, for example, in the case of a system in which components are misnamed, the semantic weight values could be reduced in favor of the static weight values. For the weight of each type of static relationship, we recommend using the values mentioned in the paper (Table 1), because we relied on previous work to assign these values. For the last step of

our approach, an expert should specify the threshold for the desired size of microservices since this parameter depends on the system to migrate and the expert's expectations.

The availability of the source code is essential to migrate an existing system to microservices using our *MicroMiner* because we derive a bottom-up microservices identification approach. From a practical standpoint, the availability of the source code of monolithic or legacy systems to be migrated is common in many industrial contexts.¹⁸ The source code is often the most up-to-date available source of information about existing software systems.¹⁸ Furthermore, our approach is applicable on both large and small systems. For example, we validated *MicroMiner* on a relatively small system (POS). The results were promising despite the small size of such system. Furthermore, we obtained similar good results when applying *MicroMiner* on Compiere, which is considered a relatively large system.

We believe that our approach is useful for both researchers and practitioners involved in migrating legacy systems to a microservices architecture because (1) we automate one of the most labor-intensive steps in migrating such systems, which is the identification of microservices, (2) our approach yields architecturally meaningful candidate microservices that satisfy two main principles of MSA: loose coupling and single responsibility per microservice, (3) our approach offers the possibility to balance static and semantic weight according to their importance and the code naming quality of the legacy system, and (4) our approach can be applied to systems of different programming languages, thanks to the use of language-independent techniques in the different steps, such as CodeBERT which supports various languages. Finally, we recommend converting utility services into libraries (jar for java, DLL for .Net languages, etc.). This could reduce the size of the source code files, minimize class redundancy, and allow them to be accessible to external systems. Furthermore, as we have opted for fuzzy clustering, meaning that certain classes can occur in several microservices, we recommend performing a code slicing step at the end, to refine the microservices and get rid of dead codes that are never used/reached in the microservice.

6 | CONCLUSION

We presented *MicroMiner*, a type-based approach for identifying microservices in monolithic software systems. *MicroMiner* is guided by a taxonomy of service types, which are predicted using ML classification models. It uses the source code of the systems, from which it extracts static relationships between components. It also performs a semantic analysis of the source code to obtain the semantic similarities among components to ensure a single bounded context per microservice.

We evaluated *MicroMiner* on four real-world legacy monolithic software systems and compared its results with ground-truths built independently. We showed that, on average, *MicroMiner* identifies architecturally-relevant and significant microservices with 68.15% precision, 77% recall, and 72.1% F-measure. Thus, we showed that *MicroMiner* could help practitioners identify candidate microservices in their monolithic software systems.

As future work, we aim to consider more legacy systems to validate the approach. This will also increase the automation degree of the approach by (1) enhancing the training of the classifier and (2) avoiding the manual labeling of some classes of each system to analyze. We aim to investigate as well database decomposition (when available), to enhance the identification accuracy of the microservices.

We could not find many systems fulfilling our criteria for the validation of our approach, we mitigated this threat by reusing systems found and used in the literature. Future work will also include finding industrial partners, who could share systems that match our criteria for further validation. We also plan a user study with such partners when our approach will include the concrete migration and deployment of the services.

We also aim to perform a qualitative validation of the identified microservices with developers to fully assess the reliability of the approach. We aim to study the quality of the identified microservices and use another clustering method in the approach guided by microservices patterns.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in *MicroMiner data* driver folder at <https://drive.google.com/drive/folders/1TQaS8etLr-32d0RXwC1Le-IOMVaDBcSS?usp=sharing>. We also include our approach implementation in the same folder.

ORCID

Imen Trabelsi  <https://orcid.org/0000-0001-7268-4067>

ENDNOTES

[†] <https://www.omg.org/spec/XMI/2.1.1>.

[‡] <https://github.com/sadatrafsanjani/JavaFX-Point-of-Sales>.

[§] <https://github.com/microsoft/CodeBERT>

[¶] <https://github.com/mmihaltz/word2vec-GoogleNews-vectors>

[#] <https://www.aptean.com>.

|| <https://www.compiere.com/products/capabilities/>.

** <https://github.com/spring-projects/spring-petclinic>.

† <https://github.com/rafaelsteil/jforum3>.

‡‡ <https://www.scitools.com/>.

§§ <https://si-serviceminer.com>.

REFERENCES

- Nadareishvili I, Mitra R, McLarty M, Amundsen M. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc.; 2016.
- Newman S. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc.; 2015. ISBN: 978-1491950357.
- Khadka R, Saeidi A, Jansen S, Hage J. A structured legacy to SOA migration process and its evaluation in practice. In: MESOCA; 2013:2-11.
- Abdellatif M, Hecht G, Mili H, et al. State of the practice in service identification for SOA migration in industry. ICSOC: Springer; 2018:634-650.
- Lewis G, Morris E, O'Brien L, Smith D, Wrage L. Smart: The service-oriented migration and reuse technique. DTIC Document; 2005.
- Fritzsche J, Bogner J, Zimmermann A, Wagner S. From monolith to microservices: a classification of refactoring approaches. *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*: Springer; 2018:128-141.
- Ahmadvand M, Ibrahim A. Requirements reconciliation for scalable and secure microservice (de) composition. In: 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW) IEEE; 2016:68-73.
- Gysel M, Kölbener L, Giersche W, Zimmermann O. Service cutter: a systematic approach to service decomposition. *European Conference on Service-Oriented and Cloud Computing*: Springer; 2016:185-200.
- Hassan S, Ali N, Bahsoon R. Microservice ambients: an architectural meta-modelling approach for microservice granularity. In: 2017 IEEE International Conference on Software Architecture (ICSA) IEEE; 2017:1-10.
- Klock S, Van Der Werf JanMartijnEM, Guelen JP, Jansen S. Workload-based clustering of coherent feature sets in microservice architectures. In: 2017 IEEE International Conference on Software Architecture (ICSA) IEEE; 2017:11-20.
- Dragoni N, Lanese I, Larsen ST, Mazzara M, Mustafin R, Safina L. Microservices: how to make your application scale. *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*: Springer; 2017:95-104.
- Jin W, Liu T, Cai Y, Kazman R, Mo R, Zheng Q. Service candidate identification from monolithic systems based on execution traces. *IEEE Trans Softw Eng*. 2019;47(5):987-1007.
- Jain H, Zhao H, Chintia NR. A spanning tree based approach to identifying web services. *Int J Web Serv Res (IJWSR)*. 2004;1(1):1-20.
- 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; 2021:1409-1418.
- Gat E, Bonnasso RP, Murphy R, et al. On three-layer architectures. *Artif Intell Mob Robots*. 1998;195:210.
- Richards M. *Software architecture patterns*, Vol. 4: O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA; 2015.
- Moravcik O, Petrik D, Skripcak T, Schreiber P. Elements of the modern application software development. *Int J Comput Theory Eng*. 2012;4(6):891.
- Abdellatif M, Shatnawi A, Mili H, et al. A taxonomy of service identification approaches for legacy software systems modernization. *J Syst Softw*. 2021;173:110868. <https://doi.org/10.1016/j.jss.2020.110868>
- Lewis J, Fowler M. Microservices: a definition of this new architectural term. *MartinFowler Com*. 2014;25:14-26.
- Rademacher F, Sachweh S, Zündorf A. Differences between model-driven development of service-oriented and microservice architecture. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW) IEEE; 2017:38-45.
- Microsoft. *Using domain analysis to model microservices* Microsoft, ed.: Microsoft; 2019. <https://docs.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis>
- Newman S. *Building microservices: designing fine-grained systems*: O'Reilly Media, Inc.; 2015.
- Abdellatif M, Tighilt R, Moha N, et al. A type-sensitive service identification approach for legacy-to-SOA migration. *International Conference on Service-Oriented Computing*: Springer; 2020:476-491.
- Huergo RS, Pires PF, Delicato FC. MDSCIM: a method and a tool to identify services. *IT Convergent Pract*. 2014;2(4):1-27.
- Escobar D, Cárdenas D, Amarillo R, et al. Towards the understanding and evolution of monolithic applications as microservices. In: 2016 XLII Latin American Computing Conference (CLEI) IEEE IEEE; 2016:1-11.
- Levcovitz A, Terra R, Valente MT. Towards a technique for extracting microservices from monolithic enterprise systems. arXiv preprint arXiv:1605.03175; 2016.
- Baresi L, Garriga M, De Renzis A. Microservices identification through interface analysis. *European Conference on Service-Oriented and Cloud Computing*: Springer; 2017:19-33.
- Mazlami G, Cito J, Leitner P. Extraction of microservices from monolithic software architectures. In: 2017 IEEE International Conference on Web Services (ICWS) IEEE; 2017:524-531.
- Mustafa O, Gómez JM, Hamed M, Pargmann H. Granmicro: a black-box based approach for optimizing microservices based applications. *From Science to Society*: Springer; 2018:283-294.
- El Boussaidi G, Belle AB, Vaucher S, Mili H. Reconstructing architectural views from legacy systems. In: 2012 19th Working Conference on Reverse Engineering IEEE; 2012:345-354.
- Bruneliere H, Cabot J, Dupé G, Madiot F. MoDisco: a model driven reverse engineering framework. *IST*. 2014;56(8):1012-1032.
- Abdellatif M, Tighilt R, Moha N, et al. A type-sensitive service identification approach for legacy-to-SOA migration. In: Kafeza E, Benatallah B, Martinelli F, Hacid H, Bouguettaya A, Motahari H, eds. *Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, proceedings*, Lecture Notes in Computer Science: Springer; 2020:476-491. https://doi.org/10.1007/978-3-030-65310-1_34
- Feng Z, Guo D, Tang D, et al. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155; 2020.
- Amari S, Wu S. Improving support vector machine classifiers by modifying kernel functions. *Neural Netw*. 1999;12(6):783-789.
- Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. *Int Conf Learn Represent (ICLR)*. 2016.

36. Swain PH, Hauska H. The decision tree classifier: design and potential. *IEEE Trans Geosci Electron*. 1977;15(3):142-147.
37. Peterson LE. K-nearest neighbor. *Scholarpedia*. 2009;4(2):1883.
38. Rish I. An empirical study of the naive Bayes classifier. In: *IJCAI 2001 workshop on empirical methods in artificial intelligence*, Vol. 3; 2001:41-46.
39. Blondel VD, Guillaume J-L, Lambiotte R, Lefebvre E. Fast unfolding of communities in large networks. *J Stat Mech Theory Exp*. 2008;2008(10):P10008.
40. Floyd RW. On ambiguity in phrase structure languages. *Commun ACM*. 1962;5(10):526.
41. Enslen E, Hill E, Pollock L, Vijay-Shanker K. Mining source code to automatically split identifiers for software analysis. In: *2009 6th IEEE International Working Conference on Mining Software Repositories IEEE*; 2009:71-80.
42. Iyyer M, Manjunatha V, Boyd-Graber J, Daumé III H. Deep unordered composition rivals syntactic methods for text classification. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (volume 1: Long papers)*; 2015:1681-1691.
43. Bezdek JC, Ehrlich R, Full W. Fcm: The fuzzy c-means clustering algorithm. *Comput Geosci*. 1984;10(2-3):191-203.
44. Saidani I, Ouni A, Mkaouer MW, Saied A. Towards automated microservices extraction using multi-objective evolutionary search. *International conference on service-oriented computing*: Springer; 2019:58-63.
45. Lapuz N, Clarke P, Abgaz Y. Digital transformation and the role of dynamic tooling in extracting microservices from existing software systems. *European Conference on Software Process Improvement*: Springer; 2021:301-315.
46. Gao X-B, Pei J, Xie W. A study of weighting exponent m in a fuzzy c-means algorithm. *Acta Electronica Sinica*. 2000;28(4):80-83.
47. Xiao Z, Wijegunaratne I, Qiang X. Reflections on SOA and microservices. In: *2016 4th International Conference on Enterprise Systems (ES) IEEE*; 2016:60-67.

How to cite this article: Trabelsi I, Abdellatif M, Abubaker A, et al. From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis. *J Softw Evol Proc*. 2022;e2503. doi:[10.1002/smr.2503](https://doi.org/10.1002/smr.2503)