RADF: Architecture Decomposition for Function as a Service

Lulai Zhu[a,*], Damian Andrew Tamburri[b], Giuliano Casale[a]

[a] *Department of Computing, Imperial College London, UK*
[b] *Jheronimus Academy of Data Science, Eindhoven University of Technology, The Netherlands*

Abstract

Serverless computing encourages completely delegating operational tasks to the cloud provider. As the most successful realization of serverless, function as a service (FaaS) brings in a novel cloud computing paradigm that can save operating costs, reduce management effort, enable seamless scalability and augment development productivity. Migration of an existent application to the serverless architecture is however an intricate task as a plethora of decisions need to be made along the way. We propose in this paper RADF[1], a semi-automatic approach for decomposing a monolith into serverless functions. The proposed approach adopts a two-stage refactoring strategy, where the decomposition process is divided into coarse- and fine-grained stages. As such, the refactoring procedure is largely simplified and adaptable to produce a solution at either microservice or function level. We have implemented RADF in a DevOps methodology compliant with OASIS TOSCA[2] and evaluated its capability for microservice identification and feasibility for code refactoring. In the evaluation experiments, RADF achieves lower coupling and relatively balanced cohesion, compared to previous decomposition approaches.

Keywords: architecture decomposition; microservices; function as a service; TOSCA

1. Introduction

The last decade has witnessed the spread of cloud computing in companies of all sizes and at a great scale [1]. To further improve the utilization of physical resources and hide the complexity of heterogeneous infrastructure, new cloud computing paradigms such as container as a service (CaaS) and function as a service (FaaS) emerge, thanks to the recent advances in virtualization and containerization technologies [2]. The rise of CaaS and FaaS triggers an observable shift of cloud applications from the traditional monolithic architecture hosted on virtual machines toward a more fine-grained one, composed of microservices or serverless functions, running on light-weight containers [3].

*Corresponding author
Email addresses: lulai.zhu15@imperial.ac.uk (Lulai Zhu), d.a.tamburri@tue.nl (Damian Andrew Tamburri), g.casale@imperial.ac.uk (Giuliano Casale)

[1] RADF stands for RADON architecture decomposition for FaaS. RADON is a research project whose objective is to unlock the benefits of serverless computing for the software industry: https://radon-h2020.eu/.

[2] https://www.oasis-open.org/tosca-implementation-stories/

Monolithic applications combine everything, including user interface, business logic, data access and service integration, within a single program. Despite being simple to get started with, this architectural pattern suffers several drawbacks whose severity grows as the features of the application evolve over time [4, 5]. Firstly, it is hard for developers to keep a detailed insight into a large monolith, causing the maintenance of source code to be tedious and error-prone. Secondly, applications designed in a monolithic architecture may only be scaled as a whole, which is a costly response to typically unbalanced and fluctuating workloads. Thirdly, the development of a monolithic application becomes cumbersome once its size reaches a certain level, since the tight coupling among various modules prevents multiple teams from working independently.

All these issues are well addressed in the serverless architecture, where an application consists of stateless functions that implement the main business logic, together with backend-as-a-service (BaaS) components, for example object storages, databases and message queues, that fulfill specific product needs [6, 7]. In most cases, one function is associated with a definite and unique role, thus being very easy to understand and maintain. The scaling of functions is conducted separately according to their own workloads, which is more efficient than simply duplicating instances of the whole application. Additionally, different development teams—or even individual developers—can merely focus on the bundle of functions assigned to themselves without concerning much about the others.

Migrating monolithic applications to the serverless architecture is generally difficult, as exemplified by case studies reported in [8], [9] and [10]. The major challenge lies in how to decompose a monolith into serverless functions that satisfy the specified functional and non-functional requirements. As a common functional requirement, the behavior of the application should remain invariant after the migration. The non-functional requirements define qualities expected of a sound solution, such as coupling, cohesion, performance, cost, security and privacy. Software designers have to make a wide range of decisions during the decomposition process in order to meet both functional and non-functional requirements. Unfortunately, there is a lack of systematic approaches to this problem, and the qualities of the final solution are largely subject to human experience.

To facilitate migration to the serverless architecture, we propose a semi-automatic approach called RADF[1] for refactoring a monolithic application into one compatible with the FaaS paradigm. The proposed approach divides the entire process into two stages, i.e. coarse- and fine-grained decompositions. It identifies candidate microservices by clustering API operations in the first stage and composes each identified microservice with serverless functions in the second stage. We have implemented RADF as part of a DevOps methodology, which aims to offer holistic support for developing and operating serverless applications based on OASIS TOSCA.[2] An evaluation of the proposed approach has been carried out in terms of its capability for microservice identification and

feasibility for code refactoring. The evaluation results indicate that the decomposition solution given by RADF has lower coupling and relatively balanced cohesion against baselines found in prior art.

The rest of the paper is organized as follows. Section 2 reviews the related work to show the motivation behind RADF. Section 3 describes a running example to be used afterward. An overview of the refactoring strategy and the reference workflow is present in Section 4. The similarity analysis and cluster discovery of API operations are elaborated in Sections 5 and 6, respectively. Section 7 outlines a practical procedure for code refactoring. Section 8 introduces the implementation of the proposed approach. The evaluation experiments and results are reported in Section 9. Section 10 is dedicated to the conclusions.

2. Related Work

Automatic decomposition of a monolith into microservices has gained extensive attention in the past few years. Gysel et al. [11] offered a tool named Service Cutter, which constructs a weighted undirected graph capturing the domain model and coupling information of a software system, and derives service decomposition from the graph with the Girvan-Newman [12] or the epidemic label propagation algorithm [13]. Baresi et al. [14] introduced the idea of identifying microservices by means of interface analysis. Specifically, they generate candidate microservices by mapping operations defined in the API specification onto the concepts in a reference vocabulary. Mazlami et al. [15] suggested three coupling strategies, namely logical, semantic and contributor, to obtain a graph representation of an application based on its revision history or source code as well as a minimum spanning tree clustering algorithm to extract candidate microservices from the graph. De Alwis et al. [16] presented function splitting heuristics for identifying consumer-oriented parts of an enterprise system that could be suitably re-engineered as microservices, with respect to business object subtypes or common execution fragments. A data flow-driven approach was proposed by Li et al. [17]. This approach takes a well-defined diagram of processes and data stores depicting the business logic of an application and finds candidate microservices by grouping the processes and closely related data stores into individual modules. Jin et al. [18] contributes a service identification approach that discovers functional atoms from the execution traces of different test cases and resorts to the non-dominated sorting genetic algorithm II [19] to search for a partition of the functional atoms with the optimal structural and conceptual intra- and inter-connectivities. A graph neural network with the ability to dilute outliers was applied by Desai et al. [20] to decomposing monoliths. They create the call graph of an application through static analysis of the source code and populate it with attributes that incorporate execution traces of the API entry points and inheritance dependencies among the classes. Kalia et al. [21] developed Mono2Micro. This tool acquires candidate microservices by performing hierarchical clustering on the classes

according to direct and indirect call relations and patterns observed in the execution traces of different use cases. A multi-model-based approach to microservice identification was proposed by Daoud et al. [22], who retrieve control, data and semantic dependencies from the business process model of an application and put highly-dependent activities into the same microservice using a collaborative clustering algorithm.

There are a couple of publications devoted to automatic decomposition of a monolith into serverless functions. To convert Java code into Lambda[3] functions, Spillner and Dorodko [23] devised a FaaSification pipeline comprising six steps, specifically analysis, decomposition, translation, compilation, upload and verification, and implemented a tool called Podilizer to support automation of this pipeline. They also offered Lambada [24], another tool that follows a similar idea but operates on Python code. De Carvalho and de Araújo [25] developed NodeJS2FaaS for migrating Node.js code to various FaaS compute services including AWS Lambda,[3] Google Cloud Functions[4] and Microsoft Azure Functions.[5] NodeJS2FaaS performs the migration in five steps: extraction, normalization, assembly, compression and publication. The aforementioned three approaches have two common weaknesses. Code dependencies are computed via static analysis. Advanced language features, for example reflection, dynamic class loading and dependency injection, are consequently left aside, leading to discrepancies between the detected and the actual dependencies. Besides, these approaches essentially carry out mechanical transformation without taking into account the business logic of the application, which is an important factor in making decomposition decisions. Noticing the current gap, we propose RADF as an alternative approach for decomposing a monolith at the function level. RADF adopts a two-stage refactoring strategy, where the decomposition process is divided into coarse- and fine-grained stages. This simplifies the refactoring procedure into smaller steps. Moreover, RADF produces decomposition solutions based on business logic inherent in the interface of the application, leveraging the state-of-the-art techniques for natural language processing and dynamic tracing.

3. Running Example

We consider the famous Cargo Shipping system[6] as a running example to demonstrate RADF in the subsequent sections. The Cargo Shipping system is a sample Java application introduced by Evans [26] to illustrate domain-driven design, a software development approach that maps software artifacts onto business domain concepts defined by human experts. Prior art such as [11], [14], [17] and [22] have used this application to evaluate their decomposition approaches, which potentially provides baselines for us to compare RADF with these approaches at the level of microservices.

[3] `https://aws.amazon.com/lambda/`
[4] `https://cloud.google.com/functions/`
[5] `https://azure.microsoft.com/en-us/services/functions/`
[6] `https://github.com/citerus/dddsample-core`

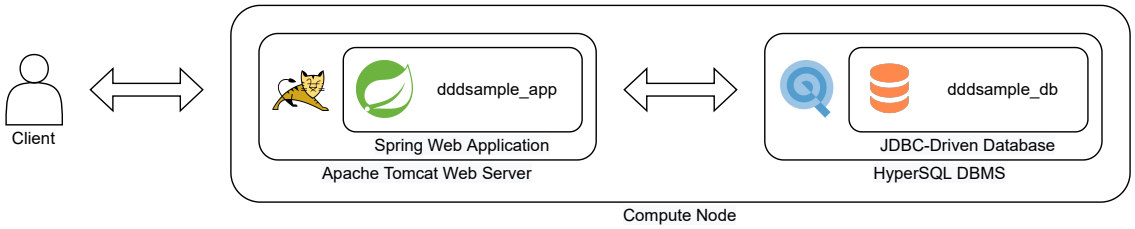

Figure 1: The architecture of the Cargo Shipping system.

As can be seen from Figure 1, the Cargo Shipping system has a typical monolithic architecture consisting of a Spring[7] web application named `dddsample_app` and a JDBC[8]driven database named `dddsample_db`. The two components are deployed respectively on an Apache Tomcat[9] web server and a HyperSQL[10] database management system (DBMS) operating on some compute node. At runtime, The `dddsample_app` web application accesses the `dddsample_db` database for persistent data storage, reading and writing information about cargoes, locations, voyages and handling events.

The Cargo Shipping system provides functionalities for managing and tracking cargoes. These functionalities are implemented conceptually by two subsystems, which we refer to as Cargo Admin and Cargo Tracking respectively. The Cargo Admin subsystem is present for the system manager to book a new cargo, show the details of the registered cargoes, pick a destination or select an itinerary for a cargo. The Cargo Tracking subsystem updates the status of every cargo according to the submitted handling reports, and allows a customer to inspect the handling history of a cargo with a given tracking ID.

Suppose that a software designer wants to migrate the Cargo Shipping system to the serverless architecture. They may ask themselves the following list of questions before taking action:

- What does the code structure of the decomposition solution look like?
- How many functions do I need to preserve the behavior of the application?
- What are the role and boundary associated with each function?

The last question is the hardest one, which involves numerous decisions on the assignment of classes. The goal of RADF is to find answers to these questions in a systematic way.

4. Approach Overview

To help with a better understanding of RADF in detail, we draw a preliminary overview of the proposed approach in this section, particularly focusing on the refactoring strategy adopted

[7] `https://spring.io/`
[8] `https://www.oracle.com/database/technologies/appdev/jdbc.html`
[9] `https://tomcat.apache.org/`
[10] `http://hsqldb.org/`

for the decomposition of a monolith into serverless functions as well as the reference workflow to complete this task.

4.1. Refactoring Strategy

The interface of an application defines a set of specific operations that interpret and execute the business logic. We think a pair of API operations correlated if they coordinate to serve the same business capability or subdomain. To produce a decomposition solution with loose coupling and tight cohesion, candidate microservices can be identified as small self-contained services that are responsible for different groups of correlated API operations. A natural way to compose such a service in a FaaS-compatible manner is to handle one API operation with a dedicated serverless function.

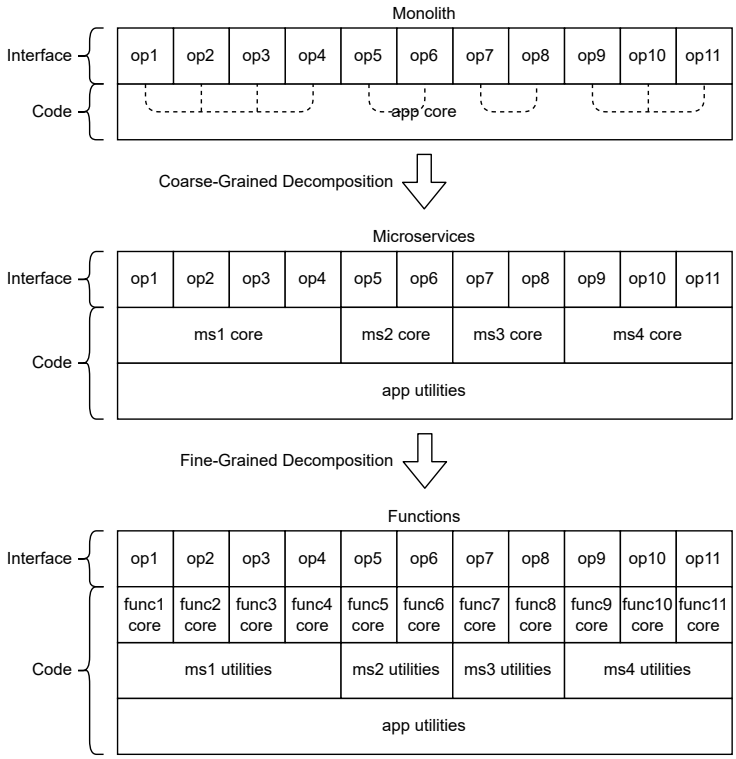

Figure 2: The refactoring strategy for decomposing a monolith into serverless functions.

Building on these notions, the refactoring strategy illustrated in Figure 2 is adopted to decompose a monolith into serverless functions. RADF divides the entire process into two stages, namely coarse- and fine-grained decompositions. All the classes are initially assumed to form a single layer: the application core. In the first stage, the monolith is decomposed into microservices by partitioning its interface into groups of correlated operations, each of which implies a candidate microservice. Classes implementing the application are separated into the core elements of the microservices and global utilities shared among them. This leads to a code structure with two

layers called microservice cores and application utilities respectively. In the second stage, every microservice is further decomposed into as many serverless functions as API operations in the corresponding group. Classes belonging to the microservice are separated into the core elements of the serverless functions and local utilities shared among them. A three-layer code structure consisting of function cores, microservice utilities and application utilities is thus obtained.

Dividing the decomposition process into the coarse- and fine-grained stages reduces the difficulty of the refactoring procedure and enables the reuse of ideas and techniques from prior work on microservice identification. Microservices and serverless functions arising from this refactoring strategy also follow the single responsibility principle, which can enhance the quality of the resultant decomposition solution. Since the interface of the application remains unchanged, its behavior is preserved from a client perspective. No additional modifications are demanded on the client side after the refactoring.

4.2. Reference Workflow

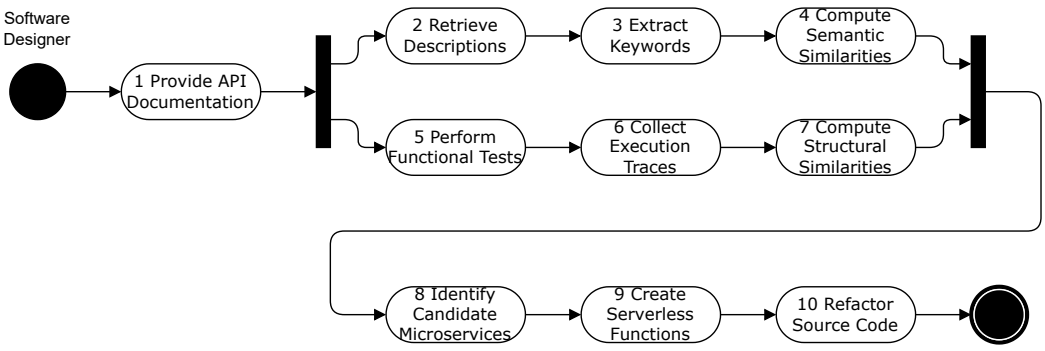

Figure 3: The reference workflow for decomposing a monolith into serverless functions.

Figure 3 shows the reference workflow adopted to decompose a monolith into serverless functions. RADF produces the decomposition solution to an application through semantic and structural analysis of its interface. As a requisite, the software designer must provide an API documentation, preferably compliant with a certain standard like the OpenAPI specification.[11] The semantic analysis starts by retrieving the descriptions of API operations. The keywords of the descriptions are then extracted and used to compute the semantic similarities. As for the structural analysis, functional tests are performed to collect the execution traces of API operations. The structural similarities are computed from the traces. Afterward, candidate microservices are identified by clustering API operations based on their semantic and structural similarities. A number of serverless functions are created to compose each identified microservice. One handles an individual API operation. Finally, the source code of the application is refactored accordingly.

[11] `https://swagger.io/specification/`

The API documentation is essential for the above reference workflow, but not all the applications have a well-documented interface. The Cargo Shipping system is such an example. In this case, the software designer ought to scan the interface of the application and gather adequate information for both semantic and structural analysis. In the semantic analysis, a concise description is required for every API operation. The software designer could simply use the name of the corresponding entry point or derive one from their understanding of the business logic. Technical details about API operations are needed in the structural analysis to perform functional tests and collect execution traces. These include the path, method, parameters, body and response as well as the entry point. Appendix A reports the gathered information about the interface of the Cargo Shipping system.

5. Similarity Analysis

This section introduces techniques used to analyze the semantics and structure of an API operation. Similarity measures that we devise for quantifying to what extent a pair of API operations are semantically and structurally correlated are also given, in company with their aggregated form.

5.1. Semantic Similarity

In an API documentation, the semantics of an operation is expressed in the form of a natural language description, from which a human, usually a software developer, can infer valuable information about the operation such as what it does, how it works and when to use it. A pair of API operations are found correlated typically when their descriptions embody similar semantics. As a result, the semantic similarity is possibly an effective measure for discovering correlated API operations.

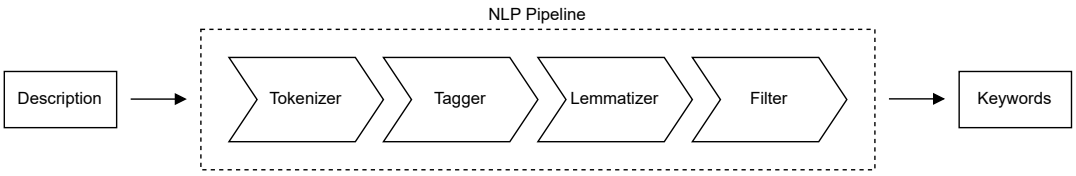

Figure 4: The NLP pipeline for extracting keywords from the description of an API operation.

To extract keywords from the description of an API operation, we build a natural language processing (NLP) pipeline as illustrated in Figure 4. This NLP pipeline consists of four components, namely tokenizer, tagger, lemmatizer and filter. The tokenizer aims to segment words, numbers and punctuation marks in the input description, which is the very first step of almost any NLP procedure [27]. Each word is then assigned a part-of-speech label by the tagger and normalized to its citation form by the lemmatizer. After that, the filter removes any words not labeled as

adjectives, nouns or proper nouns. Those left are regarded as the keywords of the description. Take the API operation 1 of the Cargo Shipping system for example. Keywords extracted from the corresponding description, i.e. "Book a new cargo", are "new" and "cargo". There are many advanced methods available for keyword extraction [28]. Nevertheless, simply selecting nouns and noun phrases as the keywords is sufficient in our case, given that the description of an API operation is relatively short, one or two sentences in general.

Thanks to the recent development of techniques such as word2vec [29], GloVe [30] and fastText [31], the semantics of a word can be accurately represented as a dense vector. Vector space models learned using these techniques are termed word embeddings. They naturally satisfy additive compositionality, allowing the representation of a text by summing up the semantic vectors of all the words. Plain summation however does not capture the fact that different words contribute unequally to the overall semantics of the text. For example, "cargo" is a more important keyword than "new" in describing the API operation 1 of the Cargo Shipping system. The same problem is confronted in the task of information retrieval, where term frequency (TF) and inverse document frequency (IDF) weights come to aid [32]. Inspired by this solution, we apply word embedding along with TF-IDF weighting and compute the semantic vector of API operation i as

$$\boldsymbol{v}_i = \sum_{t \in q_i} \mathrm{tf}(t, d_i)\mathrm{idf}(t, \mathcal{D})\mathrm{idf}(t, \mathcal{A})\boldsymbol{u}_t, \tag{1}$$

where d_i is the description of the API operation, q_i is the set of keywords extracted from d_i, $\mathcal{D}$ is the set of descriptions in the API documentation, $\mathcal{A}$ is the set of articles in the English Wikipedia, and $\boldsymbol{u}_t$ is the semantic vector of keyword t. Table 1 reports the TF-IDF weighting scheme adopted to compute the semantic vector of API operation i. The definitions of $\mathrm{tf}(t, d_i)$ and $\mathrm{idf}(t, \mathcal{D})$ follow practical forms[12] seen in industry rather than theoretical forms shown in textbooks. It is notable that the combination of word embedding and TF-IDF weighting has led to success in various NLP tasks such as text classification [33]. Given the semantic vectors of two operations i and j, we use the half-wave rectified cosine of the included angle $\theta_{i,j}$ to measure their similarity:

$$s_{i,j}^{\mathrm{sem}} = \max\{0, \cos\theta_{i,j}\} = \max\left\{0, \frac{\boldsymbol{v}_i \cdot \boldsymbol{v}_j}{\|\boldsymbol{v}_i\|_2 \|\boldsymbol{v}_j\|_2}\right\}, \tag{2}$$

which differs from the cosine similarity in the sense that opposite semantics are interpreted as being completely unrelated.

5.2. Structural Similarity

The business logic of a modern application is programmed in an object-oriented fashion, where business data and rules centering around a specific type of object are encapsulated as the fields

[12] `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html`

Table 1: The TF-IDF weighting scheme for computing the semantic vector of API operation i.

Definition*	**Description**	**Purpose**
$\mathrm{tf}(t, d_i) = \ln(\lvert\{w \in d_i \mid w = t\}\rvert) + 1$	Sublinear TF weight of keyword t in the description d_i of the API operation	Scale up the effect of recurring keywords
$\mathrm{idf}(t, \mathcal{D}) = \ln\left(\frac{\lvert\mathcal{D}\rvert}{\lvert\{d \in \mathcal{D} \mid t \in d\}\rvert}\right) + 1$	Empirical IDF weight of keyword t on the set $\mathcal{D}$ of descriptions in the API documentation	Scale down the effect of domain keywords
$\mathrm{idf}(t, \mathcal{A}) = \ln\left(\frac{\lvert\mathcal{A}\rvert}{\lvert\{a \in \mathcal{A} \mid t \in a\}\rvert}\right)$	Standard IDF weight of keyword t on the set $\mathcal{A}$ of articles in the English Wikipedia	Scale down the effect of general keywords

* The curly brackets in the definition of $\mathrm{tf}(t, d_i)$ symbolize bags; those in the definitions of $\mathrm{idf}(t, \mathcal{D})$ and $\mathrm{idf}(t, \mathcal{A})$ symbolize sets.

and methods of the corresponding class respectively. It is often the case that a pair of correlated API operations depend on largely overlapping subsets of classes and exhibit similar structures. Therefore, the structural similarity may also be an effective measure for discovering correlated API operations.

There are basically two strategies, static and dynamic, to explore the structure of an API operation. The static strategy parses the source code into an abstract syntax tree and probes the structure of the API operation by iterating over all the nodes. The dynamic strategy carries out a functional test on the API operation and draws the structure from execution traces collected in different test cases. Both strategies have their own issue. Consider the problematic scenario shown in Figure 5. This is a real scenario encountered during the refactoring of the Cargo Shipping system. `CargoRepository` is an abstract class extended by `CargoRepositoryHibernate` and `CargoRepositoryInMem`. Objects of the two concrete classes are constructed at runtime through dependency injection and accessed by the API operation 1 and a unit test respectively via references of the abstract class. On one hand, it is impossible to verify whether the API operation depends on `CargoRepositoryHibernate` or `CargoRepositoryInMem` statically. On the other hand, the dependency of the API operation on `CargoRepositoryHibernate` is determinable dynamically, but `CargoRepository` would be neglected as the class hierarchy of an object is normally not traced. We choose the dynamic strategy to explore the structures of API operations more accurately. The refactoring procedure described in Section 7 fills missing classes in the decomposition solution resorting to static dependency analysis.

We are particularly interested in the subset of classes observed in the execution traces of each API operation, and call it the class trace for conciseness. Let $\mathcal{T}_i$ denote the class trace of API

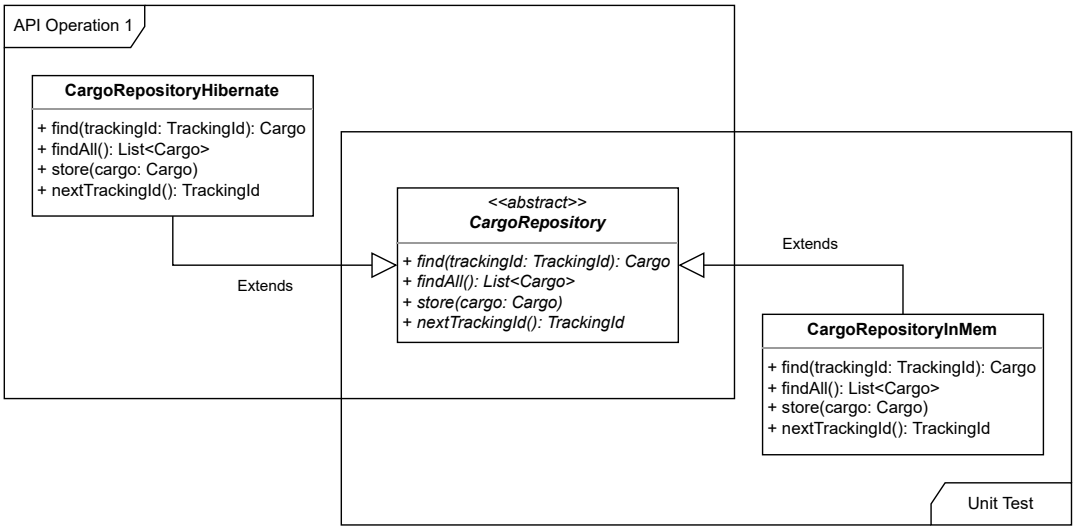

Figure 5: A scenario that cannot be properly dealt with by either static or dynamic strategy.

operation i. We define the structural similarity between two API operations i and j by

$$s_{i,j}^{\mathrm{str}} = \frac{|\mathcal{T}_i \cap \mathcal{T}_j|}{\min\{|\mathcal{T}_i|, |\mathcal{T}_j|\}}, \tag{3}$$

which ranges from 0 if $\mathcal{T}_i \cap \mathcal{T}_j = \varnothing$ to 1 if $\mathcal{T}_i \subset \mathcal{T}_j$ or vice versa, measuring to what extent the smaller of $\mathcal{T}_i$ and $\mathcal{T}_j$ is contained in the larger. The similarity measure proposed in (3) is more sensitive than the Jaccard index in detecting a common API design pattern where an interaction between a client and the application is realized by using a pair of API operations to exchange forms. Take the API operations 10 and 11 of the Cargo Shipping system for example. These two API operations jointly enable a customer to inspect the handling history of a cargo with a given tracking ID. The former gets an empty form for the customer to enter the tracking ID of a cargo, while the latter shows the handling history of that cargo upon form submission. Table 2 compares the Jaccard index and the proposed measure on the API operations 10 and 11 of the Cargo Shipping system. Although the two API operations are obviously correlated, the Jaccard index of their class traces is merely 0.12. By contrast, the proposed measure results in a value of 1.00.

Table 2: Comparison of the Jaccard index and the proposed measure on the API operations 10 and 11 of the Cargo Shipping system.

Class Trace		**Jaccard Index**	**Proposed Measure**
API Operation 10	**API Operation 11**		
CargoTrackingController TrackCommand	CargoTrackingController TrackCommand ... (15 other classes)	0.12	1.00

5.3. Aggregation

We identify candidate microservices as groups of correlated API operations by means of clustering. Typical clustering algorithms assume the availability of a similarity or distance for each pair of data points. Thus, we aggregate the semantic and structural similarities between two API operations i and j as

$$s_{i,j} = \alpha s_{i,j}^{\text{sem}} + (1 - \alpha) s_{i,j}^{\text{str}}, \tag{4}$$

where α is a trade-off factor adjustable from 0 to 1. One may set α to be greater than 0.5 if the API documentation provides a precise description for every operation. A value of less than 0.5 is instead preferable in the case of an application whose source code is well structured. Since the similarity measure given by (4) is bounded between 0 and 1, we further define a distance measure as its complement to 1:

$$d_{i,j} = 1 - s_{i,j}. \tag{5}$$

Table 3 reports pairwise distances obtained by applying similarity analysis to the API operations of the Cargo Shipping system with a word2vec model [29] and a trade-off factor α of 0.5. To show the implication in Table 3, we represent the API operations as data points in a two-dimensional space through non-metric multidimensional scaling [34]. It is easy to find from Figure 6 four groups of correlated API operations: $\{1, 2, 3, 4\}$, $\{5, 6\}$, $\{7, 8\}$ and $\{9, 10, 11\}$, each corresponding to a candidate microservice for the Cargo Shipping system. This partition exactly matches that suggested by experienced software designers in the evaluation experiments.

Table 3: Pairwise distances between the API operations of the Cargo Shipping system.

ID	1	2	3	4	5	6	7	8	9	10	11
1	0.0000	0.1615	0.3012	0.2240	0.5401	0.5401	0.3239	0.5153	0.6105	0.8565	0.6690
2	0.1615	0.0000	0.2403	0.1250	0.5030	0.4736	0.5020	0.4191	0.4848	0.8611	0.5670
3	0.3012	0.2403	0.0000	0.0933	0.3802	0.4236	0.4276	0.4496	0.4632	0.8782	0.5449
4	0.2240	0.1250	0.0933	0.0000	0.4147	0.5085	0.3844	0.4816	0.4922	0.8611	0.5486
5	0.5401	0.5030	0.3802	0.4147	0.0000	0.0577	0.3498	0.3199	0.6524	0.9476	0.6420
6	0.5401	0.4736	0.4236	0.5085	0.0577	0.0000	0.4287	0.2886	0.5774	0.9476	0.6420
7	0.3239	0.5020	0.4276	0.3844	0.3498	0.4287	0.0000	0.0922	0.6143	0.9531	0.6753
8	0.5153	0.4191	0.4496	0.4816	0.3199	0.2886	0.0922	0.0000	0.5660	0.9673	0.6548
9	0.6105	0.4848	0.4632	0.4922	0.6524	0.5774	0.6143	0.5660	0.0000	0.5965	0.2076
10	0.8565	0.8611	0.8782	0.8611	0.9476	0.9476	0.9531	0.9673	0.5965	0.0000	0.0000
11	0.6690	0.5670	0.5449	0.5486	0.6420	0.6420	0.6753	0.6548	0.2076	0.0000	0.0000

6. Cluster Discovery

After the similarity analysis, candidate microservices can be automatically identified by clustering API operations. Clustering is an unsupervised machine learning technique that aims to divide a collection of M data points x_1, x_2, …, x_M into K disjoint groups C_1, C_2, …, C_K, termed

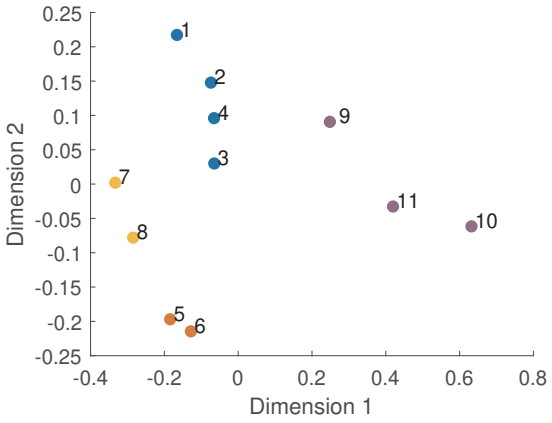

Figure 6: API operations of the Cargo Shipping system in a two-dimensional representation.

clusters, according to their similarities or distances. As such, data points in the same cluster are more similar or closer in some sense to each other than to those in a different cluster. There are plenty of clustering algorithms available in the literature [35]. We consider four general-purpose clustering algorithms, hierarchical [36], K-medoids [37], DBSCAN [38] and spectral [39], that do not have restrictions on the similarity or distance measure. An investigation of these algorithms is present in Appendix B.

Clustering algorithms normally requires the specification of a few input parameters. Some of the parameters, for example the neighborhood ϵ of a data point in the DBSCAN algorithm, are difficult to determine from a priori knowledge. A common solution is to collect partitions under multiple parameter settings and keep the one that optimizes a validity index computed based on the data set. Many indices have been devised for this purpose, including Caliński-Harabasz [40], Dunn [41], gamma [42], Davies-Bouldin [43], silhouette [44], Krzanowski-Lai [45] and CDbw [46]. Unfortunately, most of them are not suitable in our case:

- the Caliński-Harabasz, Dunn and Davies-Bouldin indices are observed to be insensitive on the given data sets;
- the Krzanowski-Lai and CDbw indices may only be applied to data points explicitly defined on an vector space.

To evidence the first argument, we conduct the single-linkage hierarchical clustering on the API operations of the Cargo Shipping system, and compute the Caliński-Harabasz, Dunn, gamma, Davies-Bouldin and silhouette indices of the resultant partition as the number K of clusters to discover increments from 2 to 9. These indices except Davies-Bouldin are optimal at the maximum value. It can be seen from Figure 7 that the Caliński-Harabasz, Dunn and Davies-Bouldin indices seem to favor as many clusters as possible while the gamma and silhouette indices exhibit good effectiveness in seeking the right number of clusters. We therefore opt for the latter two indices. Appendix C reviews technical details about them.

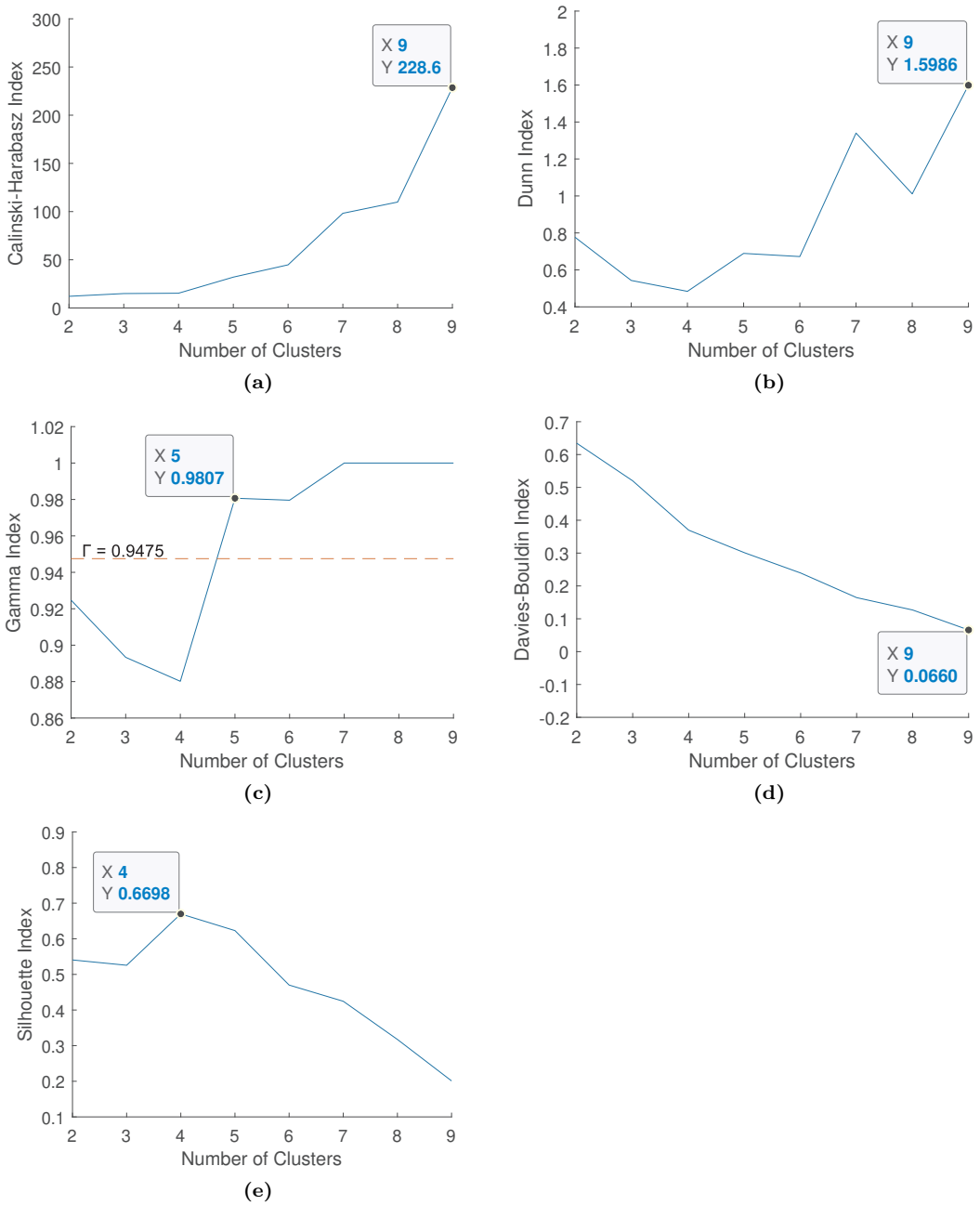

Figure 7: Caliński-Harabasz, Dunn, gamma, Davies-Bouldin and silhouette indices of partitions resulting from hierarchical clustering on the API operations of the Cargo Shipping system.

7. Code Refactoring

In what follows, we outline a practical procedure for refactoring a monolithic application into one compatible with the FaaS paradigm. This procedure applies the two-stage refactoring strategy discussed in Section 4.1 and attains the target code structure at each decomposition stage based on the identified microservices and the class traces of API operations. Classes that can hardly

be captured at runtime, such as pure abstract and exception classes, are also taken into account via static dependency analysis. The procedure consists of five sequential steps. For the sake of clarification, the last four steps are formulated in Algorithms 1, 2, 3 and 4 respectively. Table 4 summarizes notation used therein.

Table 4: Notation used in Algorithms 1, 2, 3 and 4.

Symbol	Definition
i, j	API operation indices
k, l	Microservice indices
p, q	Class indices
M	Number of API operations
K	Number of Microservices
N	Number of Classes
η	Proportion of microservices by which a global utility is depended on
$\mathcal{T}_i$	Set of classes observed in the execution traces API operation i
$\mathcal{O}_k$	Set of correlated API operations assigned to microservice k
$\mathcal{D}_{\mathrm{I}}$	Set of inheritance dependencies between classes
$\mathcal{D}_{\mathrm{A}}$	Set of access (non-inheritance) dependencies between classes
$\mathcal{U}_{\mathrm{G}}$	Set of global utilities shared among all the microservices
$\mathcal{E}_k$	Set of core elements implementing microservice k
$\mathcal{U}_k$	Set of local utilities shared among serverless functions composing microservice k
$\mathcal{E}_i$	Set of core elements implementing serverless function i
$\mathcal{L}$	Mutable set acting like a linked list
$\mathcal{X}, \mathcal{Y}, \mathcal{Z}$	Temporary sets computed along the way
$\mathcal{S}(\cdot)$	An arbitrary member in a given set $\mathcal{S}$

Step 1. Split the entry point classes of the application so that fields and methods related to each API operation are encapsulated in a separate class. For example, the `CargoTrackingController` class of the Cargo Shipping system provides entry points for both API operations 10 and 11, and thus needs to be split into two classes possibly named `CargoTrackingGetController` and `CargoTrackingOnSubmitController`. This may demand significant effort for certain applications, subject to how tightly fields and methods within the same entry point class are coupled.

Step 2. For every class observed in the execution traces of API operations, decide whether it is a global utility or a core element of a microservice. We define a global utility as a class on which the dependence of microservices reaches a specified proportion η. Any class not satisfying this condition is regarded as a core element of the microservice with the most number of API operations dependent on it. When multiple alternatives are available, the one with the highest ratio of such API operations is chosen.

Algorithm 1: Deciding whether an observed class is a global utility or a core element of a microservice

Input: M, K, η, $\mathcal{T}_i$, $\mathcal{O}_k$
Output: $\mathcal{U}_\mathrm{G}$, $\mathcal{E}_k$

$\mathcal{U}_\mathrm{G} := \varnothing$
for $k \in [1\,..\,K]$
    $\mathcal{E}_k := \varnothing$
    ▷ Set of classes observed in the execution traces of API operations
    $\mathcal{X} := \bigcup_{i=1}^{M} \mathcal{T}_i$
    for $p \in \mathcal{X}$
        ▷ Set of microservices with API operations dependent on class p
        $\mathcal{Y} := \{k \mid \exists i, i \in \mathcal{O}_k, p \in \mathcal{T}_i\}$
        if $|\mathcal{Y}| \geq \eta K$
            $\mathcal{U}_\mathrm{G} := \mathcal{U}_\mathrm{G} \bigcup \{p\}$
        else
            ▷ Set of microservices with the most number of API operations dependent on class p
            $\mathcal{Z} := \mathrm{argmax}_{k \in [1..K]} |\{i \mid i \in \mathcal{O}_k, p \in \mathcal{T}_i\}|$
            if $|\mathcal{Z}| > 1$
                ▷ Set of microservices with the highest ratio of API operations dependent on class p
                $\mathcal{Z} := \mathrm{argmax}_{k \in \mathcal{Z}} |\{i \mid i \in \mathcal{O}_k, p \in \mathcal{T}_i\}|/|\mathcal{O}_k|$
            $k := \mathcal{Z}(\cdot)$
            $\mathcal{E}_k := \mathcal{E}_k \bigcup \{p\}$

Step 3. For every class belonging to a microservice, decide whether it is a local utility of the microservice or a core element of a serverless function. We deem a class to be a core element of a serverless function if no API operations except the one handled by that serverless function depends on it. Otherwise, the class is put into local utilities shared among serverless functions comprising the microservice.

Algorithm 2: Deciding whether an observed class is a local utility of a microservice or a core element of a serverless function

Input: M, K, $\mathcal{T}_i$, $\mathcal{E}_k$
Output: $\mathcal{U}_k$, $\mathcal{E}_i$

for $k \in [1\,..\,K]$
    $\mathcal{U}_k := \varnothing$
for $i \in [1\,..\,M]$
    $\mathcal{E}_i := \varnothing$
for $k \in [1\,..\,K]$
    for $p \in \mathcal{E}_k$
        ▷ Set of API operations assigned to microservice k and depending on class p
        $\mathcal{X} := \{i \mid i \in \mathcal{O}_k, p \in \mathcal{T}_i\}$
        if $|\mathcal{X}| > 1$
            $\mathcal{U}_k := \mathcal{U}_k \bigcup \{p\}$
        else
            $i := \mathcal{X}(\cdot)$
            $\mathcal{E}_i = \mathcal{E}_i \bigcup \{p\}$

Step 4. Incorporate classes that are not captured at runtime. These are mainly pure abstract and exception classes, which normally do not appear in the execution traces. We resort to static analysis to detect both inheritance and access (non-inheritance) dependencies between classes, and

prioritize the former over the latter in making decomposition decisions. Any class not covered so far is treated as a global utility if it is depended on by another global utility or the core elements of more than one microservice. Otherwise, we assign the class to the single microservice with classes dependent on it, and decide whether it is a local utility of the microservice or a core element of a serverless function following rules similar to those used in Step 3.

Algorithm 3: Incorporating classes that are not captured at runtime[13]

Input: M, K, N, $\mathcal{T}_i$, $\mathcal{D}_\mathrm{I}$, $\mathcal{D}_\mathrm{A}$, $\mathcal{U}_\mathrm{G}$, $\mathcal{E}_k$, $\mathcal{U}_k$, $\mathcal{E}_i$
Output: $\mathcal{U}_\mathrm{G}$, $\mathcal{E}_k$, $\mathcal{U}_k$, $\mathcal{E}_i$

▷ Set of classes not observed in the execution traces of API operations
$\mathcal{L} := [1\,..\,N] \setminus \bigcup_{i=1}^{M} \mathcal{T}_i$
$size := 1$
while $|\mathcal{L}| < size$
  $size := |\mathcal{L}|$
  for $p \in \mathcal{L}$
    ▷ Set of classes inherited from class p
    $\mathcal{X} := \{q \mid (q, p) \in \mathcal{D}_\mathrm{I}\}$
    if $\mathcal{X} = \varnothing$
      ▷ Set of classes accessing class p
      $\mathcal{X} := \{q \mid (q, p) \in \mathcal{D}_\mathrm{A}\}$
    if $\mathcal{X} = \varnothing$ **or** $\mathcal{L} \bigcap \mathcal{X} \neq \varnothing$
      continue
    $\mathcal{L} := \mathcal{L} \setminus \{p\}$
    ▷ Set of microservices with classes dependent on class p
    $\mathcal{Y} := \{k \mid \mathcal{E}_k \bigcap \mathcal{X} \neq \varnothing\}$
    if $\mathcal{U}_\mathrm{G} \bigcap \mathcal{X} \neq \varnothing$ **or** $|\mathcal{Y}| > 1$
      $\mathcal{U}_\mathrm{G} := \mathcal{U}_\mathrm{G} \bigcup \{p\}$
    else
      $k := \mathcal{Y}(\cdot)$
      $\mathcal{E}_k := \mathcal{E}_k \bigcup \{p\}$
      if $\mathcal{U}_k \bigcap \mathcal{X} \neq \varnothing$
        $\mathcal{U}_k := \mathcal{U}_k \bigcup \{p\}$
      else
        ▷ Set of serverless functions composing microservice k and containing classes
        ▷ dependent on class p
        $\mathcal{Z} := \{i \mid i \in \mathcal{O}_k, \mathcal{E}_i \bigcap \mathcal{X} \neq \varnothing\}$
        if $|\mathcal{Z}| > 1$
          $\mathcal{U}_k := \mathcal{U}_k \bigcup \{p\}$
        else
          $i := \mathcal{Z}(\cdot)$
          $\mathcal{E}_i := \mathcal{E}_i \bigcup \{p\}$

Step 5. Address the inheritance and reverse dependency issue. It is unavoidable to introduce the following unrealistic dependencies in the previous steps:

- inheritance dependencies between the core elements of different microservices;
- reverse dependencies from the global utilities to the core elements of microservices.

Such dependencies can be resolved by iteratively moving the required classes into the global utilities.

[13] This algorithm tends to leave aside classes within an inheritance or access cycle

Algorithm 4: Addressing the inheritance and reverse dependency issue

Input: M, K, $\mathcal{D}_\mathrm{I}$, $\mathcal{D}_\mathrm{A}$, $\mathcal{U}_\mathrm{G}$, $\mathcal{E}_k$, $\mathcal{U}_k$, $\mathcal{E}_i$
Output: $\mathcal{U}_\mathrm{G}$, $\mathcal{E}_k$, $\mathcal{U}_k$, $\mathcal{E}_i$

▷ Set of inheritance dependencies between the core elements of different microservices
1: $\mathcal{X} := \{(q,p) \mid (q,p) \in \mathcal{D}_\mathrm{I} \wedge (\exists (l,k), q \in \mathcal{E}_l, p \in \mathcal{E}_k, l \neq k)\}$
2: **while** $\mathcal{X} \neq \varnothing$
3: **for** $(q,p) \in \mathcal{X}$
4: $k := \{k \mid p \in \mathcal{E}_k\}(\cdot)$
5: $\mathcal{E}_k := \mathcal{E}_k \setminus \{p\}$
6: **if** $p \in \mathcal{U}_k$
7: $\mathcal{U}_k := \mathcal{U}_k \setminus \{p\}$
8: **else**
9: $i := \{i \mid p \in \mathcal{E}_i\}(\cdot)$
10: $\mathcal{E}_i := \mathcal{E}_i \setminus \{p\}$
11: $\mathcal{U}_\mathrm{G} := \mathcal{U}_\mathrm{G} \bigcup \{p\}$
12: $\mathcal{X} := \{(q,p) \mid (q,p) \in \mathcal{D}_\mathrm{I} \wedge (\exists (l,k), q \in \mathcal{E}_l, p \in \mathcal{E}_k, l \neq k)\}$
▷ Set of reverse dependencies from the global utilities to the core elements of microservices
13: $\mathcal{Y} := \{(q,p) \mid (q,p) \in \mathcal{D}_\mathrm{I} \bigcup \mathcal{D}_\mathrm{A} \wedge q \in \mathcal{U}_\mathrm{G} \wedge (\exists k, p \in \mathcal{E}_k)\}$
14: **while** $\mathcal{Y} \neq \varnothing$
15: **for** $(q,p) \in \mathcal{Y}$
16: **repeat** 4–11
17: $\mathcal{Y} := \{(q,p) \mid (q,p) \in \mathcal{D}_\mathrm{I} \bigcup \mathcal{D}_\mathrm{A} \wedge q \in \mathcal{U}_\mathrm{G} \wedge (\exists k, p \in \mathcal{E}_k)\}$

8. Implementation

RADF has been implemented as the architecture decomposition feature of the RADON methodology[14] to facilitate the refactoring of a monolithic application into a FaaS-based one. This feature takes a TOSCA model as input, solves the decomposition problem in that model, updates it according to the solution found and returns the resultant model as output. Both input and output models conform to RADON Particles,[15] a unified modeling profile that broadens the scope of TOSCA Simple YAML Profile v1.3[16] to serverless computing.

Topology and Orchestration Specification for Cloud Applications (TOSCA) is an OASIS standard language for modeling cloud applications and their orchestration workflows [47]. In essence, a TOSCA model is a service template, which depicts the topology of an application as a colored directed graph containing nodes, relationships and the attached policies. It can be packaged in company with associated artifacts, such as binaries, scripts and configuration files, into a cloud service archive and then executed by a TOSCA-compliant orchestrator to automate the deployment and management of the application.

As illustrated in Figure 8, we have extended the definitions of three node types, namely `WebApplication`, `ContainerApplication` and `Function`, in RADON Particles to support RADF.

[14] `https://github.com/radon-h2020/radon-methodology`
[15] `https://github.com/radon-h2020/radon-particles`
[16] `https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html`

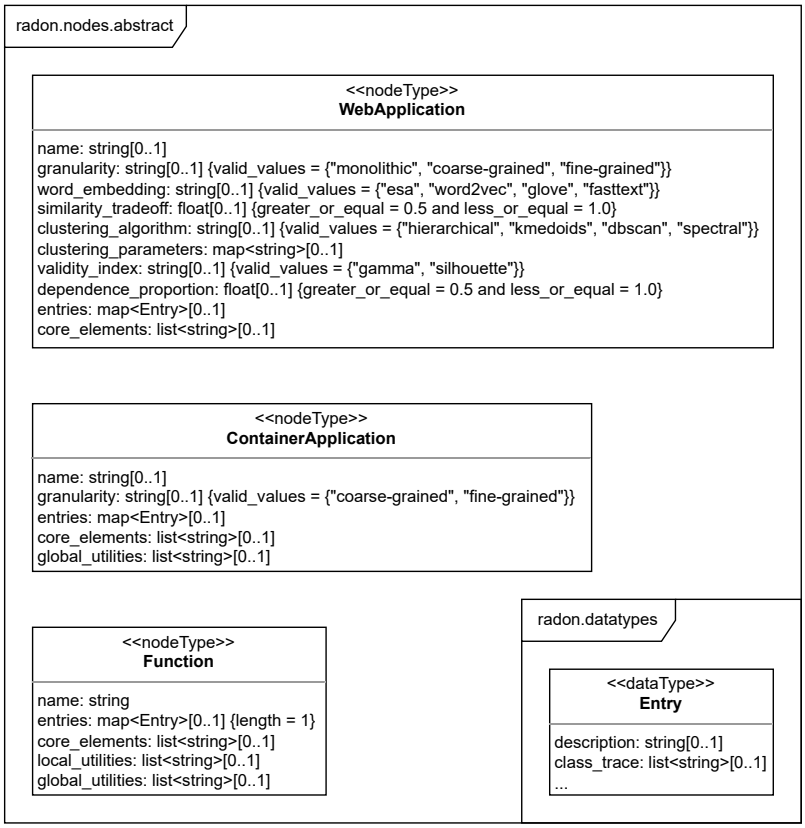

Figure 8: Definitions of the `WebApplication`, `ContainerApplication` and `Function` node types in RADON Particles.

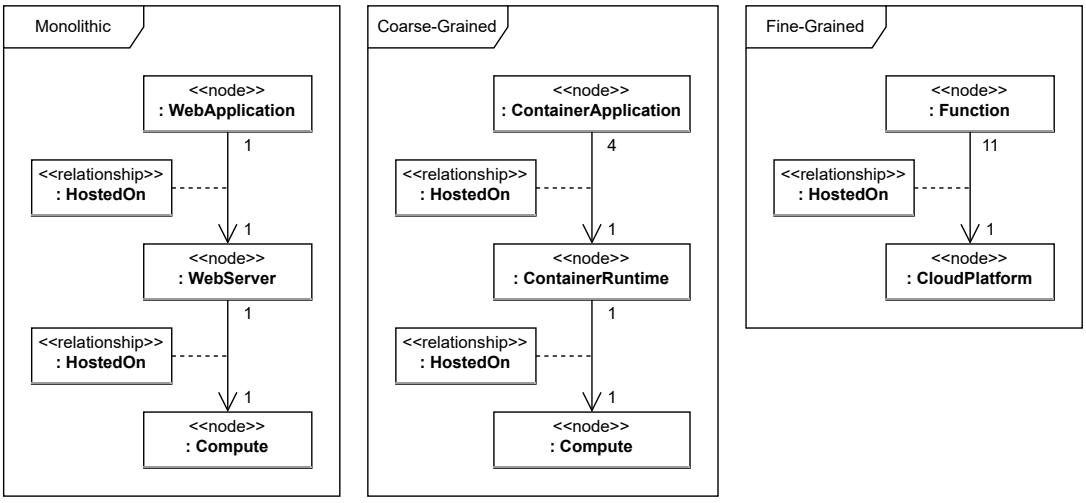

Figure 9: TOSCA models of the Cargo Shipping system at different granularity levels.

These node types are invented for representing a monolith, a microservice and a serverless function, respectively. They can be used along with the `WebServer`, `ContainerRuntime`, `Compute` and `CloudPlatform` node types as well as the `HostedOn` relationship type to draw a monolithic, a

coarse-grained, a fine-grained or even a hybrid architecture. Figure 9 shows the TOSCA models of the Cargo Shipping system at different granularity levels.

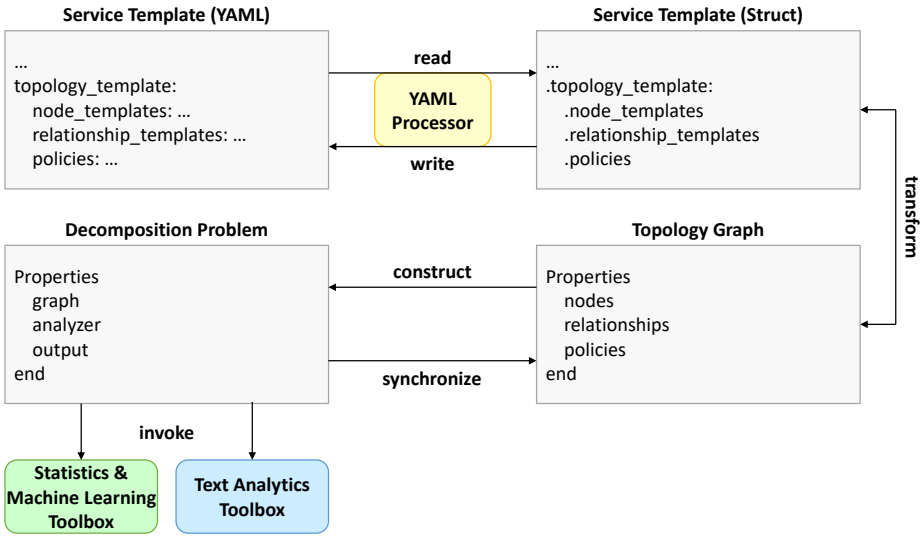

Figure 10: The architecture decomposition feature of the RADON methodology.

The architecture decomposition feature of the RADON methodology is implemented as illustrated in Figure 10. Given the TOSCA model of a monolithic application, the feature relies on an integrated YAML processor to read the service template into a MATLAB structure and generates a so-called topology graph through model-to-model transformation. This graph encompasses nodes, relationships and policies defined in the service template. A decomposition problem is then constructed from the topology graph and solved by performing similarity analysis, cluster discovery and code refactoring, as detailed in Sections 5, 6 and 7. Apart from the common facilities, two specialized MATLAB toolboxes are invoked during the solution process: the statistics & machine learning[17] and the text analytics toolbox,[18] which provide the implementations of clustering algorithms and the support for building the NLP pipeline and loading word embeddings respectively. The service template will be updated once the decomposition solution is found. It is worth pointing out that the architecture decomposition feature of the RADON methodology only automates the activities 3, 4, 7, 8, 9 and partly 10 of the reference workflow described earlier in Section 4.2. The remainder have to be done manually with the help of language-specific tools for dynamic tracing and static dependency analysis.

[17] `https://www.mathworks.com/help/stats/index.html`
[18] `https://www.mathworks.com/help/textanalytics/index.html`

9. Evaluation

This section reports an evaluation of RADF apropos its capability of identifying candidate microservices. Furthermore, we have evaluated the feasibility of RADF for code refactoring on the Cargo Shipping system and compared the final solution with baselines obtained by applying other decomposition approaches.

9.1. Capability for Microservice Identification

Besides the Cargo Shipping system, six additional applications are selected from the GitHub codebase to evaluate the capability of RADF for microservice identification, including Pet Store,[19] Pet Clinic,[20] Polls App,[21] Spring React Blog,[22] Spring Boot Blog[23] and Shopping Cart.[24] These are all Java applications designed in a monolithic architecture. As a preparation, the interfaces of the applications are scanned and documented in a format similar to Tables A.1 and A.2. Two experienced software designers are invited to identify candidate microservices for each application according to the definitions of API operations and their understanding of source code. After that, a discussion takes place to reach a consensus on the identified microservices, which form the gold standard for evaluation.

Following the reference workflow of RADF, we conduct functional tests on API operations and collect execution traces with the BTrace tool.[25] The class trace of an API operation is acquired by putting together classes appearing in its execution traces. A TOSCA model annotated with the descriptions and class traces of API operations is created for every application and decomposed by the RADON methodology at the microservice level using different clustering algorithms and validity indices. We also consider the special scenario where the parameters of clustering algorithms are manually set by human experts based on a priori knowledge. This helps to reveal the best performance of a clustering algorithm in the presence of human experts and the effectiveness of a validity index in automatically parameterizing a clustering algorithm.

RADF advocates analyzing the semantics of API operations via word embedding. In the experiments, we leverage a word2vec model, GoogleNews-vectors-negative300 [29], pretrained on part of the Google News data set. The trade-off factor α is fixed at 0.5 all the time to balance between semantic and structural similarities. Table 5 lists the parameter settings of clustering algorithms for coarse-grained decomposition. We regard the definition of the inter-cluster distance $D(C_k, C_l)$ as a parameter of hierarchical clustering due to its great impact on the behavior of the algorithm. The initialization of K-medoids is randomized. To guarantee reproducibility, we run

[19] `https://github.com/mybatis/jpetstore-6`
[20] `https://github.com/spring-petclinic/spring-framework-petclinic`
[21] `https://github.com/callicoder/spring-security-react-ant-design-polls-app`
[22] `https://github.com/keumtae-kim/spring-boot-react-blog`
[23] `https://github.com/reljicd/spring-boot-blog`
[24] `https://github.com/reljicd/spring-boot-shopping-cart`
[25] `https://github.com/btraceio/btrace`

the K-medoids algorithm multiple times and keep the partition that minimizes the objective function in (B.4). The number R of times to repeat the clustering is increased exponentially from 5 to 80 until the value of the objective function no longer improves. A similar setup is enacted on the K-means algorithm for spectral clustering as well. The minimum neighbors m of a core point is specified as 1 in DBSCAN, which is to prevent outliers from being classified as noise points and consequently excluded out of any clusters. For the spectral clustering algorithm, the scaling factor σ of the Gaussian kernel is set to 0.6006. This choice will result in a distance $d(x_i, x_j)$ greater than 0.5 being mapped onto an edge weight $w(x_i, x_j)$ less than $1 - d(x_i, x_j)$ and vice versa.

Table 5: Parameter settings of clustering algorithms for coarse-grained decomposition.

Clustering Algorithm	**Clustering Parameter**	**Optional Values**
Hierarchical	Definition of inter-cluster distance: $D(C_k, C_l)$	(B.1)*, (B.2)*, (B.3)*
	Number of clusters to discover: K	$[2\,..\,M-1]^{\dagger}$
K-Medoids	Number of clusters to discover: K	$[2\,..\,M-1]^{\dagger}$
	Number of times to repeat the clustering: R	5, 10, 20, 40, 80
DBSCAN	Neighborhood of a data point: ϵ	$[0.5, 0]$
	Minimum neighbors of a core point: m	1
Spectral	Number of clusters to discover: K	$[2\,..\,M-1]^{\dagger}$
	Scaling factor of the Gaussian kernel: σ	0.6006
	Number of times to repeat the clustering: R	5, 10, 20, 40, 80

* These corresponds to the single, complete and average linkages, respectively.
† M denotes the total number of data points.

Coarse-grained decomposition solutions generated under different combinations of clustering algorithms and validity indices were assessed against the gold standard in terms of the F1 score originating from [48]. Let C_k be the group of API operations assigned to microservice k in the gold standard, and let $C'_{k'}$ be that associated with microservice k' in a generated solution. We compute the precision, recall and F1 score of $C'_{k'}$ with respect to C_k as

$$P(C_k, C'_{k'}) = \frac{|C_k \cap C'_{k'}|}{|C'_{k'}|}, \tag{6}$$

$$R(C_k, C'_{k'}) = \frac{|C_k \cap C'_{k'}|}{|C_k|}. \tag{7}$$

$$F_1(C_k, C'_{k'}) = \frac{2P(C_k, C'_{k'})R(C_k, C'_{k'})}{P(C_k, C'_{k'}) + R(C_k, C'_{k'})}. \tag{8}$$

$P(C_k, C'_{k'})$ is the ratio of the shared API operations to those in $C'_{k'}$, while $R(C_k, C'_{k'})$ is the ratio of the shared API operations to those in C_k. $F_1(C_k, C'_{k'})$ is the harmonic mean of the two measures. The overall F1 score of the generated solution is defined by

$$F_1 = \sum_{k=1}^{K} \frac{|C_k|}{M} \max_{k' \in [1..K']} F_1(C_k, C'_{k'}), \tag{9}$$

where K and K' are the numbers of groups in the gold standard and the generated solution

respectively. (9) essentially matches every group in the gold standard to one in the generated solution with the highest F1 score against it, and takes the weighted average of such F1 scores across all the groups in the gold standard.

Table 6: F1 scores of coarse-grained decomposition solutions arising from different combinations of clustering algorithms and validity indices.

Clustering Algorithm	**Validity Index**	**Application***							**Average**
		CS	PS	PC	PA	SRB	SBB	SC	
Hierarchical	No	1.0000	1.0000	1.0000	0.9117	1.0000	1.0000	1.0000	0.9874
	Gamma	0.9455	1.0000	1.0000	0.8788	0.7569	0.9121	0.9048	0.9140
	Silhouette	1.0000	1.0000	1.0000	0.9067	1.0000	0.9333	0.7857	0.9465
K-Medoids	No	1.0000	1.0000	1.0000	0.9067	1.0000	1.0000	1.0000	0.9867
	Gamma	0.8935	1.0000	1.0000	0.8788	1.0000	0.9455	1.0000	0.9597
	Silhouette	1.0000	1.0000	1.0000	0.9067	1.0000	0.9333	0.7460	0.9409
DBSCAN	No	1.0000	0.9216	1.0000	0.8788	1.0000	0.9455	1.0000	0.9637
	Gamma	0.9455	0.9216	1.0000	0.8788	1.0000	0.9455	1.0000	0.9559
	Silhouette	1.0000	0.9216	1.0000	0.7303	1.0000	0.9455	0.8095	0.9153
Spectral	No	1.0000	1.0000	1.0000	1.0000	1.0000	0.9333	0.9048	0.9769
	Gamma	0.9455	1.0000	1.0000	0.8788	1.0000	0.8788	0.9048	0.9440
	Silhouette	1.0000	1.0000	1.0000	1.0000	1.0000	0.9333	0.7460	0.9542

* CS: Cargo Shipping, PS: Pet Store, PC: Pet Clinic, PA: Polls App, SRB: Spring React Blog, SBB: Spring Boot Blog, SC: Shopping Cart.

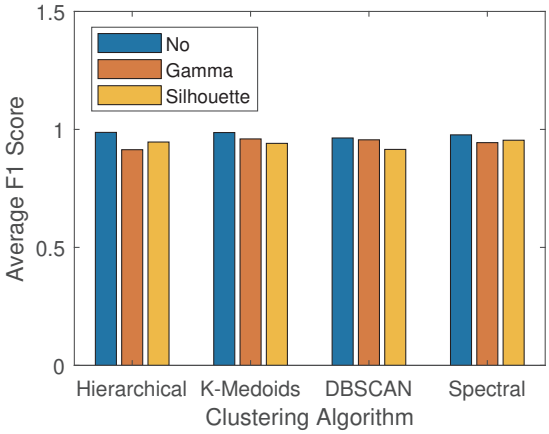

Figure 11: Average F1 scores of coarse-grained decomposition solutions arising from different combinations of clustering algorithms and validity indices.

Table 6 reports the F1 scores of coarse-grained decomposition solutions arising from different combinations of clustering algorithms and validity indices. Average values present in Table 6 are also visualized in Figure 11 for comparison. In the manual scenario, both hierarchical clustering and K-medoids yield the same candidate microservices as in the gold standard for six out of the seven applications. At the third place is the spectral clustering algorithm, which only fails in the last two cases. DBSCAN achieves an average F1 score of 0.9637 despite being the worst under manual parameterization. In the automatic scenario, the silhouette index typically outperforms the gamma index for the first six applications but works much more poorly for the last one. This defeat is mainly because the Shopping Cart application has some standalone API operations that

should be assigned to dedicated microservices. By convention, the silhouette value of a data point degenerates to 0 if it belongs to a singleton cluster. Maximizing the silhouette index tends to produce microservices with more than one API operation and thus eventuates in a bad decomposition solution to the application. Building on the above findings, Table 7 summarizes the preferable combinations of clustering algorithms and validity indices for coarse-grained decomposition.

Table 7: Preferable combinations of clustering algorithms and validity indices for coarse-grained decomposition.

Parameterization	**Standalone API Operations**	**Clustering algorithm**	**Validity Index**
Manual	No	Hierarchical/K-medoids/Spectral	No
	At least one	Hierarchical/K-medoids/DBSCAN	
Automatic	No	Hierarchical/Spectral	Silhouette
	At least one	K-medoids/DBSCAN	Gamma

9.2. Feasibility for Code Refactoring

In the aforementioned experiments, we identify four candidate microservices, namely Cargo, Planning, Location and Tracking, for the Cargo Shipping system using the hierarchical clustering algorithm and the silhouette index. The identified microservices are consistent with those suggested by the software designers and respectively responsible for four groups of API operations: $\{1, 2, 3, 4\}$, $\{5, 6\}$, $\{7, 8\}$ and $\{9, 10, 11\}$. To evaluate the feasibility of RADF for code refactoring, we employ again the RADON decomposition tool to further decompose the resultant TOSCA model at the function level, which leads to a fine-grained decomposition solution where each API operation is handled by an individual serverless function. The classes of the Cargo Shipping system are then reorganized by practicing the procedure described in Section 7 with a static code analysis tool called Sonargraph Architect.[26] In particular, we set the dependence proportion η to 1 so that a class is viewed as a global utility if and only if all the microservices expect its availability.

Coupling and cohesion are two types of qualities that software designers are most concerned about in architecture decomposition. Coupling is the degree to which two modules are mutually dependent, whereas cohesion is the degree to which one module forms a logically atomic unit. In an ideal architectural design, all the modules should be not only loosely coupled from one another but also tightly cohesive on their own. We assess the decomposition solution to the Cargo Shipping system according to four quality metrics, afferent coupling (AC) [49], efferent coupling (EC) [49], instability (I) [49] and relational cohesion (RC) [50], as detailed in Table 8. The same application and quality metrics have been used in prior work to evaluate other decomposition approaches including Service Cutter [11], concept mapping [14], data flow-driven [17] and multi-model-based [22]. We select these as the baseline approaches and collate their evaluation results from the literature to compare RADF with them.

[26] `https://www.hello2morrow.com/products/sonargraph/architect9`

Table 8: Quality metrics for assessing the decomposition solution to the Cargo Shipping system.

Quality	Metric	Description
Coupling	Afferent Coupling (AC) [49]	AC is defined as the number of classes in other modules, that depend on those in this module, and thus quantifies the module's responsibilities to other modules.
	Efferent Coupling (EC) [49]	EC is defined as the number of classes in other modules, that classes in this module depend on, and thus quantifies the module's requirements on other modules.
	Instability (I) [49]	I is calculated as the ratio of EC to the sum of AC and EC, and measures the resilience of a module to change. It varies from 0 to 1, with values of 0 and 1 signifying completely stable and unstable modules respectively.
Cohesion	Relational Cohesion (RC) [50]	RC refers to the ratio between the numbers of dependencies and classes within a module. Instance creation, class inheritance, method invocation and field access among others are all counted as dependencies.

Tables 9 and 10 report the quality metrics of decomposition solutions obtained by applying RADF and the baseline approaches to the Cargo Shipping system, respectively. As can be seen, the average afferent coupling among microservices arising from RADF is 2.25, which is 82.7% lower than the best value attained by previous approaches, i.e. 13.00 through concept mapping. Although the solution produced by RADF looks the most unstable, it is by no means inferior to the baselines given that the high instability is largely due to a reduction in the afferent coupling instead of a growth in the efferent coupling. Unlike other approaches, RADF does not cause certain microservices to be loosely cohesive. Consider the worst cases in this respect. The Tracking microservice yielded by Service Cutter for example has a rational cohesion of 4.3, whereas that of the Location microservice resulting from RADF is 10.9. Since the goal of RADF is to decompose a monolith into serverless functions, Table 9 provides two additional quality metrics, afferent coupling per function (ACF) and efferent coupling per function (ECF), as the indicators of how many incoming and outgoing dependencies still need to be resolved for each function.

Table 9: Quality metrics of the decomposition solution obtained by applying RADF to the Cargo Shipping system.

Approach	Microservice	Quality Metric					
		AC	EC	I	RC	ACF	ECF
RADF	Cargo	2	7	0.78	11.7	0.50	1.75
	Planning	4	2	0.33	12.1	2.00	1.00
	Location	3	2	0.40	10.9	1.50	1.00
	Tracking	0	1	1.00	11.4	0.00	0.33
	Average	2.25	3.00	0.63	11.53	1.00	1.02

10. Conclusion

A semi-automatic approach to architecture decomposition, named RADF, is proposed in this paper. The proposed approach adopts a two-stage strategy to refactor a monolithic application

Table 10: Quality metrics of decomposition solutions obtained by applying the baseline approaches to the Cargo Shipping system.

Approach	**Microservice**	**Quality Metric**			
		AC	EC	I	RC
Service Cutter [11]	Location	14	1	0.07	21.5
	Tracking	11	3	0.21	4.3
	Voyage & Planning	15	4	0.21	16.7
	Average	13.33	2.67	0.16	14.17
Concept Mapping [14]	Planning	16	2	0.11	20.3
	Product	13	4	0.24	1.8
	Tracking	15	5	0.25	14.9
	Trip	8	2	0.20	0.0
	Average	13.00	3.25	0.20	9.25
Data Flow-Driven [17]	Cargo	13	4	0.24	1.8
	Planning	10	3	0.23	11.5
	Location	15	1	0.06	21.5
	Tracking	16	5	0.24	14.1
	Average	13.50	3.25	0.19	12.21
Multi-Model-Based [22]	Preparation	23	0	0.00	14.0
	Handling	18	3	0.14	11.0
	Planning & Tracking	16	2	0.11	23.0
	Delivery	12	8	0.40	5.0
	Return	7	5	0.42	0.0
	Average	15.20	3.60	0.21	10.60

into one that consists of serverless functions. In the first stage, we identify candidate microservices by analyzing the semantic and structural similarities between API operations and clustering them into separate groups accordingly. As for the second stage, a number of serverless functions are created to compose each identified microservice, one handling a single API operation. We have implemented RADF as part of a DevOps methodology following OASIS TOSCA and compared it with previous decomposition approaches in refactoring the well-known Cargo Shipping system. The proposed approach brings in a decomposition solution with lower coupling and relatively balanced cohesion against the baselines.

The semantic vector of an API operation is computed by combining static word embedding with TF-IDF weighting. However, a more precise representation could be acquired using a dynamic word embedding technique such as BERT [51] or GPT [52], which incorporates the current context of each word into the resultant vector. RADF quantifies the correlation between two API operations based on their semantic and structural similarities. It may be helpful to also take into account the so-called data similarity, a measure of to what extent a pair of operations access the same portion of data. As observed in the experiments, the partition that maximizes the silhouette or gamma index is not always the best one that we can find manually. Approximating data points in an orthogonal vector space and optimizing a more sophisticated validity index, for example Krzanowski-Lai [45] and CDbw [46], is a possible solution to mitigate this disagreement.

Acknowledgments

This work is supported by the European Union's Horizon 2020 research and innovation program under grant agreement No. 825040 (RADON). The referenced data has been published at `https://zenodo.org/` with DOI 10.5281/zenodo.7014843 under the CC BY 4.0 license.

Appendix A. Interface of Running Example

Table A.1: Definitions of API operations exposed by the Cargo Shipping system (Part 1).

ID	Path	Method	Parameter/Body/Response*
1	/admin/registration	GET	
			unlocodes: string array locations: object array
2	/admin/register	POST	
			originUnlocode: string destinationUnlocode: string arrivalDeadline: string
			trackingId: string
3	/admin/list	GET	
			cargoList: object array
4	/admin/show	GET	trackingId: string
			cargo: object
5	/admin/selectItinerary	GET	trackingId: string
			routeCandidates: object array cargo: object
6	/admin/assignItinerary	POST	
			trackingId: string legs: object array
7	/admin/pickNewDestination	GET	trackingId: string
			locations: object array cargo: object
8	/admin/changeDestination	POST	trackingId: string unlocode: string
9	/ws/RegisterEvent	POST	
			arg0: object
10	/track	GET	
			trackCommand: object
11	/track	POST	
			trackingId: string
			cargo: object

* The MIME types of the body and response for each API operation except 9 are `application/x-www-form-urlencoded` and `text/html`, respectively. In particular, the API operation 9 follows the SOAP protocol to exchange messages, thereby using `text/xml` as the MIME types of both body and response.

Table A.2: Definitions of API operations exposed by the Cargo Shipping system (Part 2).

ID	Description	Entry Point	
1	Book a new cargo	se.citerus.dddsample. interfaces.booking.web. CargoAdminController	String registration(HttpServletRequest, HttpServletResponse, Map<String, Object>)
2	Register a cargo		void register(HttpServletRequest, HttpServletResponse, RegistrationCommand)
3	List all the cargoes		String list(HttpServletRequest, HttpServletResponse, Map<String, Object>)
4	Show a cargo		String show(HttpServletRequest, HttpServletResponse, Map<String, Object>)
5	Select an itinerary		String selectItinerary(HttpServletRequest, HttpServletResponse, Map<String, Object>)
6	Assign an itinerary		void assignItinerary(HttpServletRequest, HttpServletResponse, RouteAssignmentCommand)
7	Pick a new destination		String pickNewDestination(HttpServletRequest, HttpServletResponse, Map<String, Object>)
8	Change the destination		void changeDestination(HttpServletRequest, HttpServletResponse)
9	Submit a handling report	com.aggregator. HandlingReportService	void submitReport(HandlingReport)
10	Get the handling history	se.citerus.dddsample. interfaces.tracking. CargoTrackingController	String get(Map<String, Object>)
11	Show the handling history		String onSubmit(HttpServletRequest, TrackCommand, Map<String, Object>, BindingResult)

Appendix B. Clustering Algorithms

B.1. Hierarchical Clustering

Hierarchical clustering starts with each data point assigned to a separate cluster and iteratively merges the closest two clusters until the stop criterion is satisfied. We stop the algorithm when K clusters are left. The result of hierarchical clustering is significantly subject to the definition of the inter-cluster distance, known as the linkage. Below are three definitions that we consider:

- the single linkage (i.e. the shortest distance) [36], where

$$D(C_k, C_l) = \min_{x_i \in C_k, x_j \in C_l} d(x_i, x_j); \tag{B.1}$$

- the complete linkage (i.e. the longest distance) [36], where

$$D(C_k, C_l) = \max_{x_i \in C_k, x_j \in C_l} d(x_i, x_j); \tag{B.2}$$

- the average linkage (i.e. the average distance) [53], where

$$D(C_k, C_l) = \frac{1}{|C_k||C_l|} \sum_{x_i \in C_k} \sum_{x_j \in C_l} d(x_i, x_j). \tag{B.3}$$

B.2. K-Medoids

K-medoids is a variant of the celebrated K-means algorithm [54]. Its basic idea is to search for K medoids o_1, o_2, ..., o_K such that the total distance from each data point x_i to the closest medoid is minimized:

$$\begin{aligned} \min_{o_1, o_2, \ldots, o_K} \quad & \sum_{i=1}^{M} \min_{k \in [1..K]} d(x_i, o_k), \\ \text{s.t.} \quad & \forall k \in [1..K], o_k \in \{x_1, x_2, \ldots, x_M\}, \end{aligned} \tag{B.4}$$

and construct K clusters accordingly. Compared to K-means, K-medoids is not only more robust in the case of outliers but also able to handle arbitrary distance measures. The K-medoids problem (B.4) is however NP-hard to solve exactly. We obtain approximate solutions to it resorting to a widely applied heuristic named partitioning around medoids (PAM) [37] with randomized initialization. After selecting the initial medoids, PAM iteratively performs the best swap of a medoid and a non-medoid, whereby the value of the objective function in (B.4) decreases most, until the medoids no longer change.

B.3. DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN) [38] is a non-parametric clustering algorithm. In the setup of DBSCAN, two data points within a distance ϵ are deemed to be neighbors, and one with at least m neighbors is viewed as a core point. These notions yield three types of connectivities between a pair of data points x_i and x_j:

- if x_i is a core point and x_j is a neighbor of x_i, then x_j is said to be directly density-reachable from x_i;
- if there exists a sequence x_i, ..., x_j where each data point is directly density-reachable from the previous one, then x_j is said to be density-reachable from x_i;

- if x_i and x_j are density-reachable from the same core point, then x_i and x_j is said to be density-connected.

DBSCAN essentially finds clusters as groups of density-connected data points. In addition, it can recognize noise points, i.e. outliers not directly density-reachable from any core points.

B.4. Spectral Clustering

Spectral clustering models a data set as a similarity graph and seeks the best cuts of that graph through spectral decomposition of its Laplacian matrix. Various spectral clustering algorithms have been proposed [55]. We adapt the normalized spectral clustering described in [39] for use. At first, a fully connected similarity graph is constructed, and the weight of the edge between two data points x_i and x_j is computed as

$$w(x_i, x_j) = e^{-\left(\frac{d(x_i, x_j)}{\sigma}\right)^2}, \tag{B.5}$$

where σ is the scaling factor of the Gaussian kernel. Data points are then represented by the first K eigenvectors of the normalized random-walk Laplacian matrix:

$$\boldsymbol{L}^{\mathrm{RW}} = \boldsymbol{I} - \boldsymbol{D}^{-1}\boldsymbol{A}, \tag{B.6}$$

where $\boldsymbol{I}$ is the identity matrix, and $\boldsymbol{D}$ and $\boldsymbol{A}$ are the outdegree and adjacency matrices of the similarity graph respectively. Finally, the K-means algorithm is performed to divide the data set in the new representation.

Appendix C. Validity Indices

C.1. Gamma Index

The gamma index [42] is an adaption of the Goodman and Kruskal's gamma statistic for clustering validity analysis. Let N_+ be the number of times that the distance $d(x_i, x_j)$ between a pair of data points x_i and x_j belonging to the same cluster is smaller than the distance $d(x_{i'}, x_{j'})$ between another pair of data points $x_{i'}$ and $x_{j'}$ belonging to different clusters:

$$N_+ = \sum_{(x_i, x_j) \in P_{\mathrm{W}}} \sum_{(x_{i'}, x_{j'}) \in P_{\mathrm{B}}} \mathbb{1}_{\{d(x_i, x_j) < d(x_{i'}, x_{j'})\}}, \tag{C.1}$$

where $P_{\mathrm{W}} = \{(x_i, x_j) \mid i < j \wedge (\exists k, x_i \in C_k, x_j \in C_k)\}$ and $P_{\mathrm{B}} = \{(x_i, x_j) \mid i < j\} \setminus P_{\mathrm{W}}$ are the sets of pairs of data points within the same cluster and between different clusters respectively. Let

N_- be the number of times that the opposite situation occurs:

$$N_- = \sum_{(x_i,x_j)\in P_{\mathrm{W}}} \sum_{(x_{i'},x_{j'})\in P_{\mathrm{B}}} \mathbb{1}_{\{d(x_i,x_j)>d(x_{i'},x_{j'})\}}. \tag{C.2}$$

The gamma index can then be written as

$$\Gamma = \frac{N_+ - N_-}{N_+ + N_-}, \tag{C.3}$$

which varies from -1 to 1 and quantifies the rank correlation between the pairwise distance and indicator vectors. Noticing that the expected partitions appear often when the gamma index reaches nearly 1, we take the first parameter setting that satisfies a threshold of 0.9475 as the number of clusters increases.

C.2. Silhouette Index

The silhouette index [44] combines within-cluster compactness and between-cluster separateness. Consider a data point x_i assigned to a cluster C_k. The average distance between x_i and other data points in C_k is

$$a(x_i) = \frac{1}{|C_k| - 1} \sum_{\substack{x_j \in C_k \\ j\neq i}} d(x_i, x_j), \tag{C.4}$$

whilst the average distance between x_i and those in the closest cluster other than C_k is

$$b(x_i) = \min_{\substack{l\in[1..K] \\ l\neq k}} \frac{1}{|C_l|} \sum_{x_j \in C_l} d(x_i, x_j). \tag{C.5}$$

$a(x_i)$ and $b(x_i)$ respectively measure how compact x_i is to the current cluster, i.e. C_k, and how separate x_i is from the best alternative, i.e. the closest cluster other than C_k. The silhouette value of x_i is given by

$$s(x_i) = \begin{cases} 1 - \dfrac{a(x_i)}{b(x_i)} & \text{if } a(x_i) < b(x_i), \\ \dfrac{b(x_i)}{a(x_i)} - 1 & \text{if } a(x_i) > b(x_i), \\ 0 & \text{otherwise,} \end{cases} \tag{C.6}$$

which ranges from -1 to 1 and quantifies to what extent x_i matches the current cluster with respect to the best alternative. We compute the silhouette index by averaging first the silhouette values of data points in each cluster and then the resultant values across all the clusters:

$$\bar{s} = \frac{1}{K} \sum_{i=1}^{K} \frac{1}{|C_k|} \sum_{x_i \in C_k} s(x_i). \tag{C.7}$$

References

[1] E. Loukis, S. Arvanitis, N. Kyriakou, An Empirical Investigation of the Effects of Firm Characteristics on the Propensity to Adopt Cloud Computing, Information Systems and e-Business Management 15 (4) (2017) 963–988.

[2] R. Buyya, S. N. Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. M. Vaquero, M. A. S. Netto, A. N. Toosi, M. A. Rodriguez, I. M. Llorente, S. D. C. D. Vimercati, P. Samarati, D. Milojicic, C. Varela, R. Bahsoon, M. D. D. Assuncao, O. Rana, W. Zhou, H. Jin, W. Gentzsch, A. Y. Zomaya, H. Shen, A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade, ACM Computing Surveys 51 (5) (2018) 1–38.

[3] N. Kratzke, A Brief History of Cloud Application Architectures, Applied Sciences 8 (8) (2018) 1368:1–1368:26.

[4] S. Newman, Building Microservices, O'Reilly Media, 2015.

[5] C. Richardson, Microservices Patterns: With Examples in Java, Simon & Schuster, 2018.

[6] M. Roberts, J. Chapin, What is Serverless?, O'Reilly Media, 2017.

[7] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, D. A. Patterson, Cloud Programming Simplified: A Berkeley View on Serverless Computing, arXiv preprint arXiv:1902.03383 (2019).

[8] A. Goli, O. Hajihassani, H. Khazaei, O. Ardakanian, M. Rashidi, T. Dauphinee, Migrating from Monolithic to Serverless: A FinTech Case Study, in: Companion of the 11th ACM/SPEC International Conference on Performance Engineering, ACM, 2020, pp. 20–25.

[9] R. Lichtenthäler, M. Prechtl, C. Schwille, T. Schwartz, P. Cezanne, G. Wirtz, Requirements for a Model-Driven Cloud-Native Migration of Monolithic Web-Based Applications, Software-Intensive Cyber-Physical Systems 35 (1) (2020) 89–100.

[10] A. Stafford, F. G. Toosi, A. Mjeda, Cost-Aware Migration to Functions-as-a-Service Architecture, in: Companion of the 15th European Conference on Software Architecture, CEUR-WS, 2021.

[11] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann, Service Cutter: A Systematic Approach to Service Decomposition, in: Proceedings of the 5th European Conference on Service-Oriented and Cloud Computing, Springer, 2016, pp. 185–200.

[12] M. E. J. Newman, M. Girvan, Finding and Evaluating Community Structure in Networks, Physical Review E 69 (2) (2004) 026113.

[13] U. N. Raghavan, R. Albert, S. Kumara, Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks, Physical Review E 76 (3) (2007) 036106.

[14] L. Baresi, M. Garriga, A. De Renzis, Microservices Identification through Interface Analysis, in: Proceedings of the 6th European Conference on Service-Oriented and Cloud Computing, Springer, 2017, pp. 19–33.

[15] G. Mazlami, J. Cito, P. Leitner, Extraction of Microservices from Monolithic Software Architectures, in: Proceedings of the 24th IEEE International Conference on Web Services, IEEE, 2017, pp. 524–531.

[16] A. A. C. De Alwis, A. Barros, A. Polyvyanyy, C. Fidge, Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems, in: Proceedings of the 16th International Conference on Service-Oriented Computing, Springer, 2018, pp. 37–53.

[17] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, Z. Shan, A Dataflow-Driven Approach to Identifying Microservices from Monolithic Applications, Journal of Systems and Software 157 (2019) 110380.

[18] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, Q. Zheng, Service Candidate Identification from Monolithic Systems Based on Execution Traces, IEEE Transactions on Software Engineering 47 (5) (2019) 987–1007.

[19] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation 6 (2) (2002) 182–197.

[20] U. Desai, S. Bandyopadhyay, S. Tamilselvam, Graph Neural Network to Dilute Outliers for Refactoring Monolith Application, in: Proceedings of the 35th AAAI Conference on Artificial Intelligence, AAAI, 2021, pp. 72–80.

[21] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, D. Banerjee, Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 2021, pp. 1214–1224.

[22] M. Daoud, A. El Mezouari, N. Faci, D. Benslimane, Z. Maamar, A. El Fazziki, A Multi-Model Based Microservices Identiftcation Approach, Journal of Systems Architecture 118 (2021) 102200.

[23] J. Spillner, S. Dorodko, Java Code Analysis and Transformation into AWS Lambda Functions, arXiv preprint arXiv:1702.05510 (2017).

[24] J. Spillner, Transformation of Python Applications into Function-as-a-Service Deployments, arXiv preprint arXiv:1705.08169 (2017).

[25] L. R. de Carvalho, A. P. F. de Araújo, Framework Node2FaaS: Automatic NodeJS Application Converter for Function as a Service, in: Proceedings of the 9th International Conference on Cloud Computing and Services Science, SciTePress, 2019, pp. 271–278.

[26] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2004.

[27] D. Jurafsky, J. H. Martin, Speech and Language Processing, Prentice Hall, 2009.

[28] N. Firoozeh, A. Nazarenko, F. Alizon, B. Daille, Keyword Extraction: Issues and Methods, Natural Language Engineering 26 (3) (2020) 259–291.

[29] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient Estimation of Word Representations in Vector Space, arXiv preprint arXiv:1301.3781 (2013).

[30] J. Pennington, R. Socher, C. D. Manning, GloVe: Global Vectors for Word Representation, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, ACL, 2014, pp. 1532–1543.

[31] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, Enriching Word Vectors with Subword Information, Transactions of the ACL 5 (2017) 135–146.

[32] C. D. Manning, P. Raghavan, H. Schütze, An Introduction to Information Retrieval, Cambridge University Press, 2008.

[33] J. Lilleberg, Y. Zhu, Y. Zhang, Support Vector Machines and Word2vec for Text Classification with Semantic Features, in: Proceedings of the 14th IEEE International Conference on Cognitive Informatics and Cognitive Computing, IEEE, 2015, pp. 136–140.

[34] J. B. Kruskal, Multidimensional Scaling by Optimizing Goodness of Fit to a Nonmetric Hypothesis, Psychometrika 29 (1) (1964) 1–27.

[35] D. Xu, Y. Tian, A Comprehensive Survey of Clustering Algorithms, Annals of Data Science 2 (2) (2015) 165–193.

[36] S. C. Johnson, Hierarchical Clustering Schemes, Psychometrika 32 (3) (1967) 241–254.

[37] L. Kaufman, P. J. Rousseeuw, Finding Groups in Data: An Introduction to Cluster Analysis, John Wiley & Sons, 1990.

[38] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, in: Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, AAAI, 1996, pp. 226–231.

[39] J. Shi, J. Malik, Normalized Cuts and Image Segmentation, IEEE Transactions on Pattern Analysis and Machine Intelligence 22 (8) (2000) 888–905.

[40] T. Caliński, J. Harabasz, A Dendrite Method for Cluster Analysis, Communications in Statistics: Theory and Methods 3 (1) (1974) 1–27.

[41] J. C. Dunn, Well-Separated Clusters and Optimal Fuzzy Partitions, Journal of Cybernetics 4 (1) (1974) 95–104.

[42] F. B. Baker, L. J. Hubert, Measuring the Power of Hierarchical Cluster Analysis, Journal of the ASA 70 (349) (1975) 31–38.

[43] D. L. Davies, D. W. Bouldin, A Cluster Separation Measure, IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-1 (2) (1979) 224–227.

[44] P. J. Rousseeuw, Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis, Journal of Computational and Applied Mathematics 20 (1987) 53–65.

[45] W. J. Krzanowski, Y. T. Lai, A Criterion for Determining the Number of Groups in a Data Set Using Sum-of-Squares Clustering, Biometrics 44 (1) (1988) 23–34.

[46] M. Halkidi, M. Vazirgiannis, A Density-Based Cluster Validity Approach Using Multi-Representatives, Pattern Recognition Letters 29 (6) (2008) 773–786.

[47] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, TOSCA: Portable Automated Deployment and Management of Cloud Applications, in: Advanced Web Services, Springer, 2014, pp. 527–549.

[48] B. Larsen, C. Aone, Fast and Effective Text Mining Using Linear-Time Document Clustering, in: Proceedings of the 5th ACM International Conference on Knowledge Discovery and Data Mining, ACM, 1999, pp. 16–22.

[49] R. C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, 2003.

[50] C. Larman, Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development, Pearson, 2012.

[51] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding, in: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics, ACL, 2019, pp. 4171–4186.

[52] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, Improving Language Understanding by Generative Pre-Training, Tech. rep., OpenAI (2018).

[53] G. N. Lance, W. T. Williams, A General Theory of Classificatory Sorting Strategies: 1. Hierarchical Systems, The Computer Journal 9 (4) (1967) 373–380.

[54] J. MacQueen, Some Methods for Classification and Analysis of Multivariate Observations, in: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability, University of California Press, 1967, pp. 281–297.

[55] U. Von Luxburg, A Tutorial on Spectral Clustering, Statistics and Computing 17 (4) (2007) 395–416.