Received: 27 December 2022 | Revised: 14 August 2023 | Accepted: 4 September 2023

DOI: 10.1002/spe.3276

SPECIAL ISSUE PAPER

WILEY

RADF: Architecture decomposition for function as a service

Lulai Zhu[1] | **Damian Andrew Tamburri**[2] | **Giuliano Casale**[1]

[1]Department of Computing, Imperial College London, London, UK

[2]Jheronimus Academy of Data Science, Eindhoven University of Technology, 's-Hertogenbosch, The Netherlands

Correspondence
Lulai Zhu, Department of Computing, Imperial College London, Exhibition Road, London SW7 2AZ, UK.
Email: lulai.zhu15@imperial.ac.uk

Funding information
European Union's Horizon 2020 Research and Innovation Program, Grant/Award Number: 825040

Abstract

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 existing application to the serverless architecture is, however, an intricate task as a great number of decisions need to be made along the way. We propose in this paper RADF, a semi-automatic approach that decomposes a monolith into serverless functions by analyzing the business logic inherent in the interface of the application. The proposed approach adopts a two-stage refactoring strategy, where a coarse-grained decomposition is performed at first, followed by a fine-grained one. As such, the decomposition process is simplified into smaller steps and adaptable to generate a solution at either microservice or function level. We have implemented RADF in a holistic DevOps methodology 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, DevOps, function as a service, microservices

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 lightweight containers.[3]

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,

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/.

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, thereby being very easy to understand and maintain. The scaling of functions is conducted separately as per 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 an existing application to the serverless architecture is generally difficult, as exemplified by case studies reported in References 8-10. The major challenge lies in how to decompose a monolith into serverless functions that satisfy the specified functional and nonfunctional requirements. As a common functional requirement, the decomposition solution should preserve the behavior of the application. The nonfunctional requirements define qualities expected of the decomposition solution, such as coupling, cohesion, performance, cost, security, and privacy. Software designers have to make massive decisions during the decomposition process in order to meet both functional and nonfunctional requirements. Indeed, a plethora of decomposition approaches have been proposed so far.[11] Almost all of them aim for a decomposition solution at the service level and cannot help much with migration to the serverless architecture.

To this end, we propose a semi-automatic approach called RADF for refactoring a monolithic application into one compatible with the FaaS paradigm. The proposed approach divides the entire process into two stages: 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, taking into account the business logic of the application. Besides, we devise a refactoring procedure that combines dynamic tracing and static dependency analysis to obtain a provisional provisional code structure. The devised procedure exploits the outcome at each decomposition stage and rearrange the implementation of the application accordingly. A decomposition solution can then be finalized by manually resolving the remaining dependencies between microservices or serverless functions.

We have additionally implemented RADF as part of the RADON methodology,[12] which offers DevOps support for serverless applications based on OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA)[13] and facilitates the deployment of the decomposition solution on a specific cloud platform. 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 the literature.

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 presented 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, that is, coarse-grained decomposition, has gained extensive attention in the past few years. Baresi et al.[14] introduced the idea of identifying microservices by means of interface analysis. Specifically, they map operations documented in the API specification onto the concepts in a reference vocabulary and regard those grouped under the same concept as defining a candidate microservice. Al-Debagy and Martinek[15] extended this idea by computing semantic similarities between API operations from their names according to a pretrained word embedding model, word2vec[16] or fastText,[17] and combines the affinity propagation algorithm[18] with the silhouette index[19] to obtain a partition of API operations. Sun et al.[20] recently proposed a new approach that considers two different types of similarities, namely candidate topic and response message, and clusters API operations using the spectral clustering algorithm[21] and the Caliński–Harabasz index.[22] Although RADF identifies candidate microservices likewise

through interface analysis, it exploits not only the API documentation but also the source code of the application. The latter obviously contains additional information than that in the former.

A variety of other approaches for automating coarse-grained decomposition are available in References 23-32. For example, Gysel et al.[23] 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[33] or the epidemic label propagation algorithm.[34] A data flow-driven approach was proposed by Li et al.[26] It 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. A multimodel-based approach to microservice identification was proposed by Daoud et al.,[30] 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. Compared to these approaches, RADF provides a refactoring procedure for rearranging the implementation of the application into a provisional code structure compliant with the microservice architecture.

There are merely a couple of publications devoted to automatic decomposition of a monolith into serverless functions, that is, fine-grained decomposition. To convert Java code into Lambda* functions, Spillner and Dorodko[35] 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. Spillner also offered Lambada[36] as an extension of his joint work with Dorodko to Python code. Another tool named NodeJS2FaaS was developed by de Carvalho and de Araújo[37] for migrating Node.js code to various FaaS compute services including AWS Lambda,* Google Cloud Functions,† and Microsoft Azure Functions.‡ NodeJS2FaaS performs the migration in five steps: extraction, normalization, assembly, compression, and publication. Yussupov et al.[38] suggested serverless parachutes, of which the idea is to extract crucial components from annotated source code and prepare them as standby serverless functions for exceptional workloads. Zhao et al.[39] introduced BeeHive, a semi-FaaS execution model that relies on the runtime environment to extract code snippets from a Java application and offload them to the target FaaS platform. BeeHive intrinsically follows the development principles advocated later by Ghemawat.[40]

The aforementioned five approaches have two common weaknesses. Except in BeeHive, source code is analyzed only statically. Advanced language features, for example reflection, dynamic loading, and dependency injection, are consequently left aside, leading to discrepancies between the detected and the actual code structure. 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. Candidate microservices are first identified with respect to the business logic inherent in the interface of the application. This leverages the state-of-the-art techniques for natural language processing and dynamic tracing. Each identified microservice is responsible for a group of correlated API operations. A number of serverless functions are then created to compose the microservice, one implementing a single API operation. The final decomposition solution is obtained by refactoring the source code based on execution traces collected via dynamic tracing and dependencies found via static analysis. Therefore, the refactoring procedure can deal with both dynamic and static language features.

3 | RUNNING EXAMPLE

We consider the famous Cargo Shipping system§ as a running example to demonstrate RADF in the subsequent sections. The Cargo Shipping system is a sample Java application brought by Evans[41] to illustrate domain-driven design, a software development approach that maps software artifacts onto business domain concepts defined by human experts. Prior works such as References 14, 23, 26, and 30 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.

*https://aws.amazon.com/lambda/
†https://cloud.google.com/functions/
‡https://azure.microsoft.com/en-us/services/functions/
§https://github.com/citerus/dddsample-core

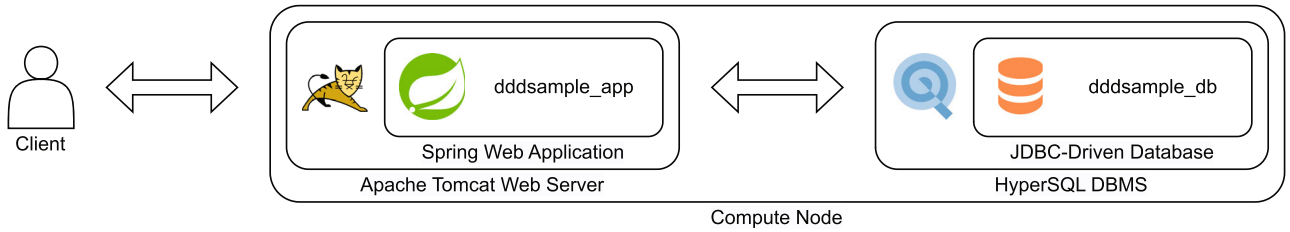

FIGURE 1 The architecture of the Cargo Shipping system.

As can be seen from Figure 1, the Cargo Shipping system has a typical monolithic architecture consisting of a Spring[¶] web application named `dddsample_app` and a JDBC-[#]driven database named `dddsample_db`. The two components are deployed, respectively, on an Apache Tomcat[‖] web server and a HyperSQL[**] 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, and 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 wishes 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 to be made along the way. 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 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.

[¶] https://spring.io/
[#] https://www.oracle.com/database/technologies/appdev/jdbc.html
[‖] https://tomcat.apache.org/
[**] http://hsqldb.org/

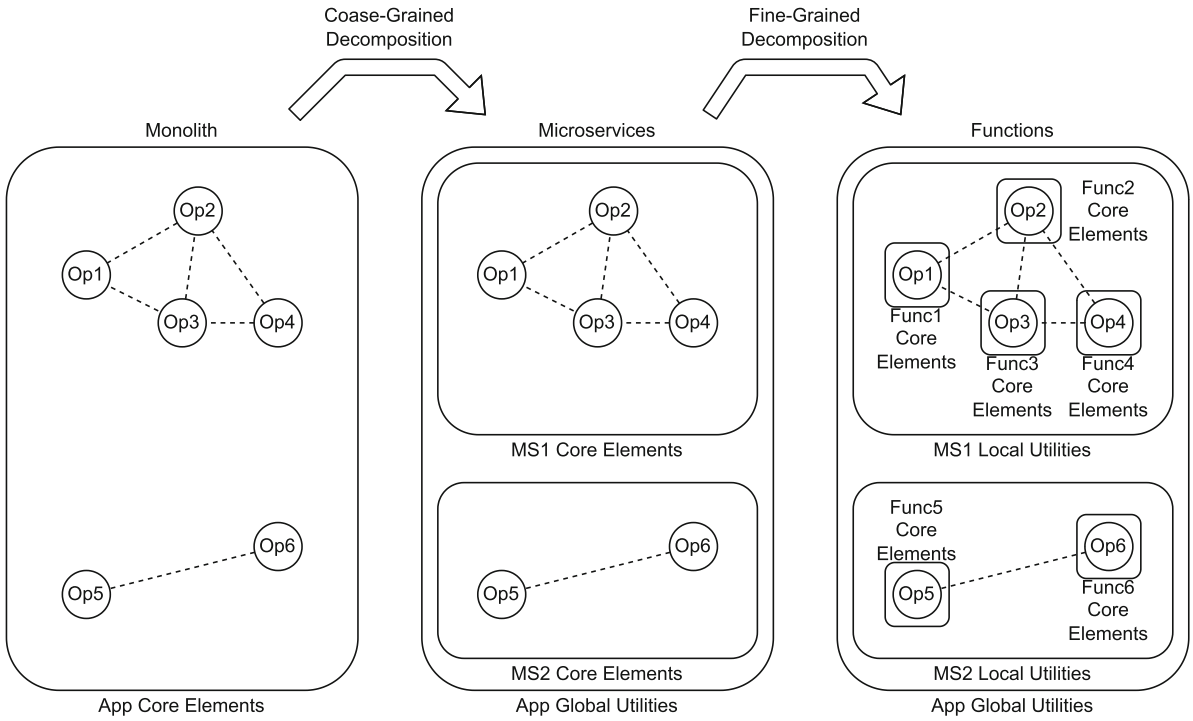

FIGURE 2 The refactoring strategy for decomposing a monolith into serverless functions.

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. We assume the monolithic application to be programmed in an object-oriented fashion and consider all the classes implementing the application to be its core elements. 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. The core elements of the application are separated into the core elements of the microservices and global utilities shared among them. In the second stage, every microservice is further decomposed into as many serverless functions as API operations in the corresponding group. The core elements of the microservice are separated into the core elements of the serverless functions and local utilities shared among them. A three-layer code structure comprising the core elements of serverless functions, the local utilities of microservices, and the global utilities of the application is obtained at the end.

Despite being ubiquitous in software design, layering poses an extra operational burden when it comes to a FaaS-based application. Since serverless functions are totally isolated from one another, the operation team has to update all the affected functions if any utility layer of the application is modified. Fortunately, the most popular FaaS compute service—AWS Lambda—automates the management of layers[††] to mitigate this issue. We expect that comparable features would be introduced by public cloud providers and the open-source community into other FaaS offerings sooner or later.

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 works 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 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,

[††] https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html

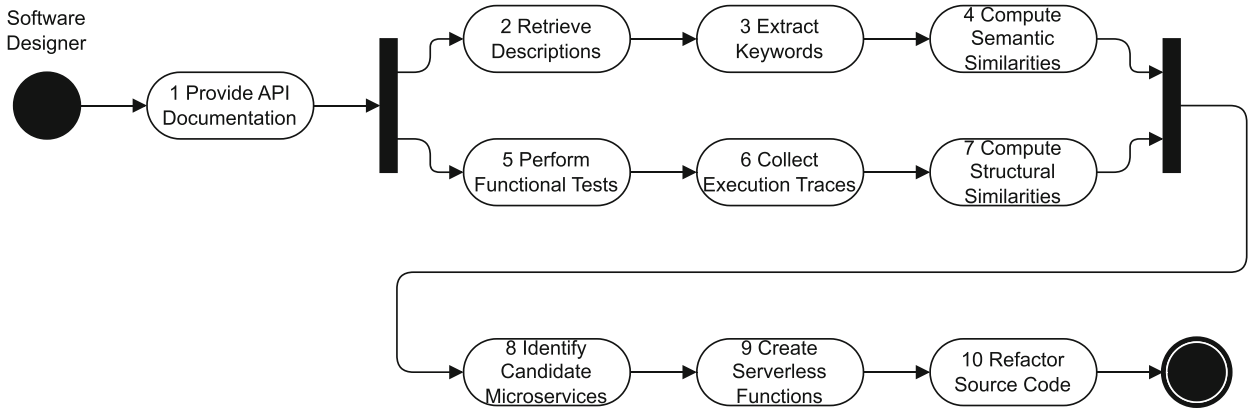

FIGURE 3 The reference workflow for decomposing a monolith into serverless functions.

the software designer must provide an API documentation, preferably compliant with a certain standard like the OpenAPI specification.[‡‡] 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.

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 reuse the name of the corresponding entry point or conclude 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.

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.[42] Each word is then assigned a part-of-speech label by the tagger and normalized to its

[‡‡] https://swagger.io/specification/

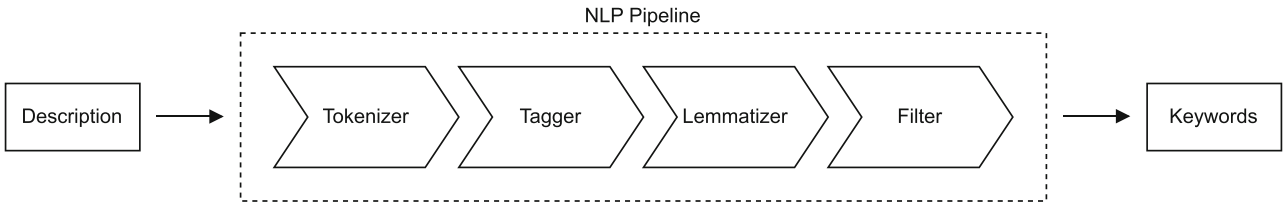

FIGURE 4 The natural language processing pipeline for extracting keywords from the description of an API operation.

citation form by the lemmatizer. After that, the filter removes any words not labeled as adjectives, nouns, or proper nouns. Those left are deemed to be the keywords of the description. Take the API operation 1 of the Cargo Shipping system as an example. Keywords extracted from the corresponding description—"Book a new cargo,"—are "new," and "cargo." There are many advanced methods available for keyword extraction.[43] Nevertheless, simply selecting nouns and noun phrases as the keywords is sufficient in our case because the description of an API operation is relatively short, often one or two sentences.

Thanks to the recent development of techniques such as word2vec,[16] GloVe,[44] and fastText,[17] the semantics of a word can be accurately encoded into a dense vector of real values. These techniques assume that words occurring in similar contexts should have similar meanings, known as the distributional hypothesis,[45] and represent each word as a point in a vector space of the specified dimension based on its surrounding words. The resultant semantic model is therefore termed a word embedding. Most of the techniques make use of neural networks to learn word embeddings in an implicit manner. Word2vec for example trains a two-layer neural network that predicts the target word for some context words (continuous bag-of-words) or the context words for a target word (continuous skip-gram), and finds the entries of semantic vectors as the final weights at the hidden layer. Without losing generality, we denote by $\boldsymbol{u}_t$ the semantic vector of an extracted keyword t in a given word embedding.

Word embeddings 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. As an 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.[46] 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, and $\mathcal{A}$ is the set of articles in the English Wikipedia. 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 derived forms[§§] widespread in practice rather than primitive forms present 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.[47] 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 an application defines a system of rules on how data can be created, stored, and altered. In object-oriented programming, business data and rules centering around a specific type of object are encapsulated as the

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

[a]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.

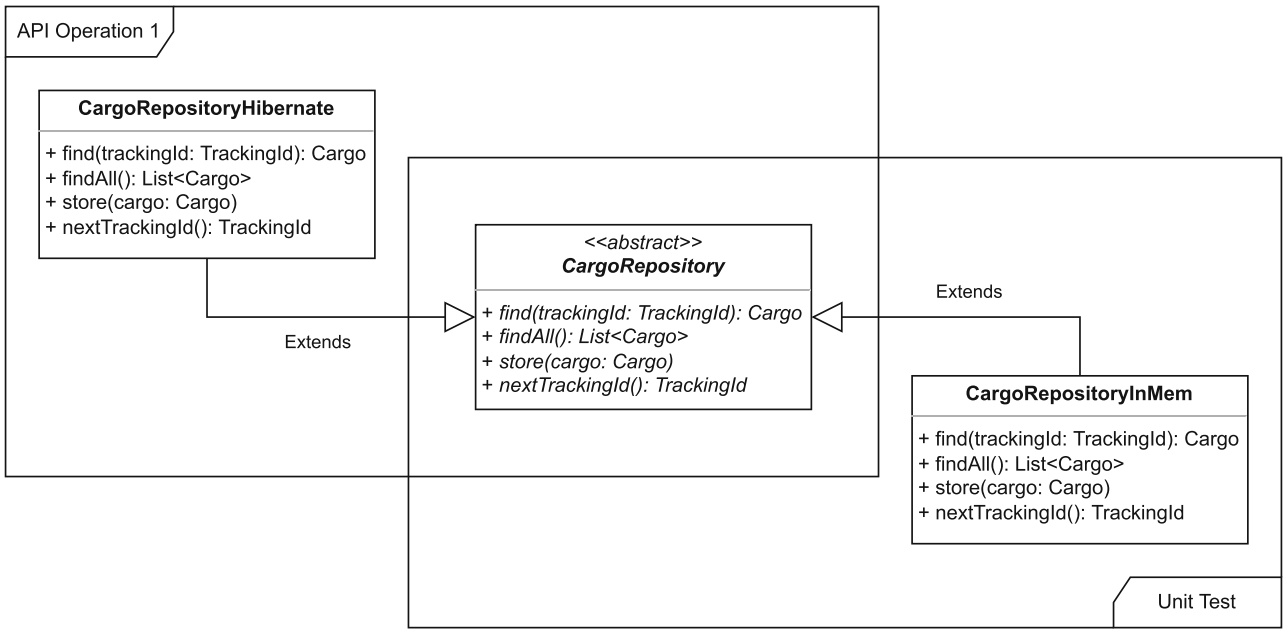

FIGURE 5 A scenario that cannot be properly dealt with by either static or dynamic strategy.

fields 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, static analysis cannot tell whether the API operation depends on `CargoRepositoryHibernate` or `CargoRepositoryInMem`. On the other hand, dynamic analysis would neglect the dependency of the API operation on `CargoRepository` as tracing the class hierarchies of objects at runtime is too costly. We choose the dynamic strategy to explore the actual structures of API operations and compute their structural similarities more accurately. The refactoring procedure described in Section 7 fills missing classes in the decomposition solution resorting to the static strategy.

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

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

which ranges from 0 if $\mathcal{T}_i \cap \mathcal{T}_j = \emptyset$ 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 Equation (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 as an 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.

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 Equation (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[16] 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.[48] 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.

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 $x_1, x_2, \ldots, x_M$ into K disjoint groups $C_1, C_2, \ldots, C_K$, termed 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.[49] We consider four general-purpose clustering algorithms,

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			
API operation 10	**API operation 11**	**Jaccard index**	**Proposed measure**
CargoTrackingController TrackCommand	CargoTrackingController TrackCommand … (15 other classes)	0.12	1.00

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

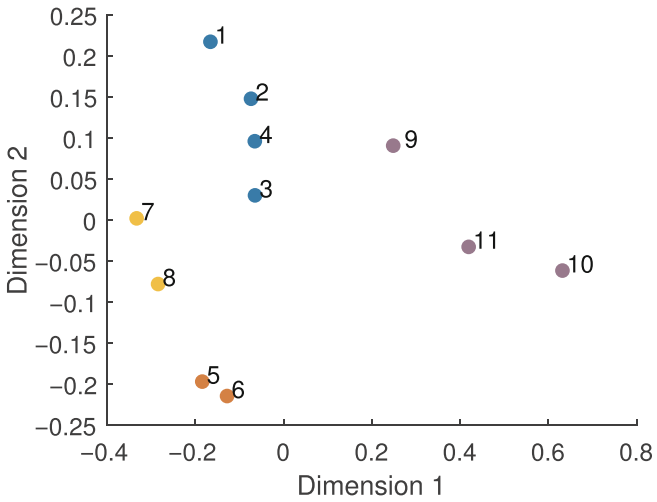

FIGURE 6 API operations of the Cargo Shipping system in a two-dimensional representation.

hierarchical,[50] *K*-medoids,[51] density-based spatial clustering of applications with noise (DBSCAN),[52] and spectral,[21] that do not have restrictions on the similarity or distance measure. An investigation of these algorithms is presented 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 dataset. Many indices have been devised for this purpose, including Caliński–Harabasz,[22] Dunn,[53] gamma,[54] Davies–Bouldin,[55] silhouette,[19] Krzanowski–Lai,[56] and CDbw.[57] 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 datasets;
- the Krzanowski–Lai and CDbw indices may only be applied to data points explicitly defined on a 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 $M - 1$, that is, from 2 to 10. These indices except Davies–Bouldin are optimal at the maximum value. As can be seen from Figure 7, the Caliński–Harabasz,

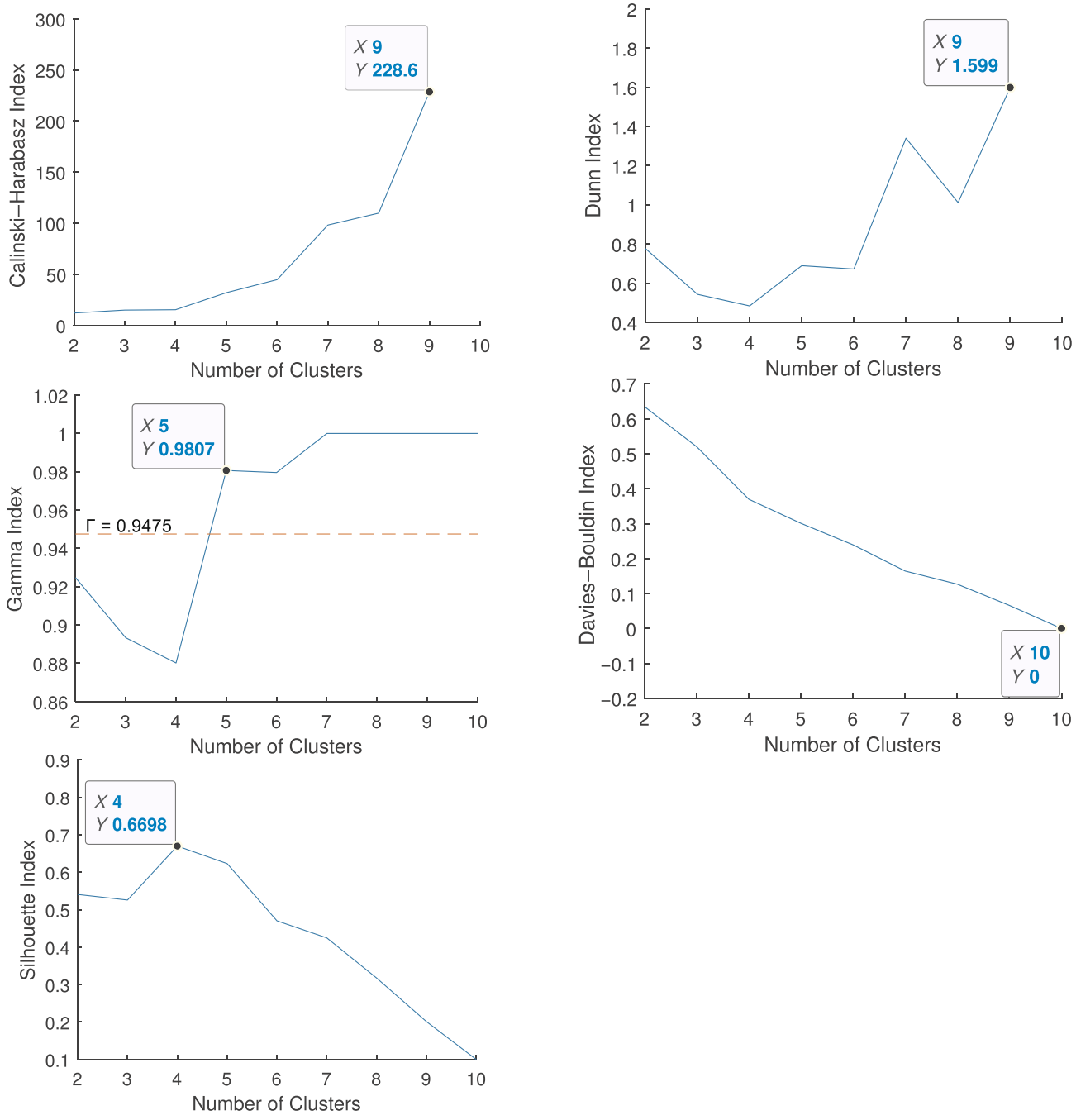

FIGURE 7 Caliński–Harabasz, Dunn, gamma, Davies–Bouldin, and silhouette indices of partitions resulting from hierarchical clustering on the API operations of the Cargo Shipping system (singular values are dropped).

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.

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.

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

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

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

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

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. As an 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.

Algorithm 3. Incorporating classes that are not captured at runtime

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: $\mathcal{U}_\mathrm{G}, \mathcal{E}_k, \mathcal{U}_k, \mathcal{E}_i$

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

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 depended on by API operations across at least a specified proportion η of microservices. Any class not satisfying the aforementioned condition is regarded as a core element of the microservice with the most dependent API operations. When multiple alternatives are available, the one with the highest ratio of such API operations is chosen.

Step 3. For every core element of a microservice, decide whether it is a local utility or a core element of a serverless function. We deem a core element of a microservice to be a local utility as long as more than one API operation depending on that core element is assigned to the microservice. If not, it is put into the core elements of the serverless function for handling the solely dependent API operation.

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 (noninheritance) dependencies between classes, and prioritize the former over the latter in making decomposition decisions. Any class not captured at runtime 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 consider the class as a core element of the sole microservice with dependent classes, and then 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 the last step. The uncaptured classes are examined consecutively, and a decision is made on a class only when all the classes depending on that class have been included. This is repeated until nothing changes. At the end, classes within dependency cycles or subgraphs are possibly left side.

Step 5. Address the reverse and inheritance dependency issue. Steps 2, 3, and 4 could introduce three types of dependencies that are difficult to fulfill in practice:

Algorithm 4. Addressing the reverse and inheritance 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: $\mathcal{U}_{\mathrm{G}}, \mathcal{E}_k, \mathcal{U}_k, \mathcal{E}_i$

1: **for do**$k \in [1..K]$
2: $\mathcal{X} := \{(q,p) \in \mathcal{D}_{\mathrm{I}} \bigcup \mathcal{D}_{\mathrm{A}} \mid q \in \mathcal{U}_k \wedge (\exists i \in \mathcal{O}_k, p \in \mathcal{E}_i)\}$ ▷ Set of dependencies from the local utilities of microservice k to the core elements of serverless functions composing it
3: **while** $\mathcal{X} \neq \emptyset$ **do**
4: **for** $(q,p) \in \mathcal{X}$ **do**
5: $i := \{i \in \mathcal{O}_k \mid p \in \mathcal{E}_i\}(\cdot)$
6: $\mathcal{E}_i := \mathcal{E}_i \setminus \{p\}$
7: $\mathcal{U}_k := \mathcal{U}_k \bigcup \{p\}$
$\mathcal{X} := \{(q,p) \in \mathcal{D}_{\mathrm{I}} \bigcup \mathcal{D}_{\mathrm{A}} \mid q \in \mathcal{U}_k \wedge (\exists i \in \mathcal{O}_k, p \in \mathcal{E}_i)\}$
8: $\mathcal{Y} := \{(q,p) \in \mathcal{D}_{\mathrm{I}} \mid \exists (l,k) \in [1..K]^2, q \in \mathcal{E}_l \wedge p \in \mathcal{E}_k \wedge l \neq k\}$ ▷ Set of inheritance dependencies between the core elements of different microservices
9: **while** $\mathcal{Y} \neq \emptyset$ **do**
10: **for** $(q,p) \in \mathcal{Y}$ **do**
11: $k := \{k \in [1..K] \mid p \in \mathcal{E}_k\}(\cdot)$
12: $\mathcal{E}_k := \mathcal{E}_k \setminus \{p\}$
13: **if** $p \in \mathcal{U}_k$ **then**
14: $\mathcal{U}_k := \mathcal{U}_k \setminus \{p\}$
15: **else**
16: $i := \{i \in \mathcal{O}_k \mid p \in \mathcal{E}_i\}(\cdot)$
17: $\mathcal{E}_i := \mathcal{E}_i \setminus \{p\}$
18: $\mathcal{U}_{\mathrm{G}} := \mathcal{U}_{\mathrm{G}} \bigcup \{p\}$
19: $\mathcal{Y} := \{(q,p) \in \mathcal{D}_{\mathrm{I}} \mid \exists (l,k) \in [1..K]^2, q \in \mathcal{E}_l \wedge p \in \mathcal{E}_k \wedge l \neq k\}$
20: $\mathcal{Z} := \{(q,p) \in \mathcal{D}_{\mathrm{I}} \bigcup \mathcal{D}_{\mathrm{A}} \mid q \in \mathcal{U}_{\mathrm{G}} \wedge (\exists k \in [1..K], p \in \mathcal{E}_k)\}$ ▷ Set of dependencies from the global utilities to the core elements of microservices
21: **while** $\mathcal{Z} \neq \emptyset$ **do**
22: **for** $(q,p) \in \mathcal{Z}$ **do**
23: **repeat** 4–11
24: $\mathcal{Z} := \{(q,p) \in \mathcal{D}_{\mathrm{I}} \bigcup \mathcal{D}_{\mathrm{A}} \mid q \in \mathcal{U}_{\mathrm{G}} \wedge (\exists k \in [1..K], p \in \mathcal{E}_k)\}$

- dependencies from the local utilities of a microservice to the core elements of serverless functions composing it;
- inheritance dependencies between the core elements of different microservices;
- dependencies from the global utilities to the core elements of microservices.

The first type of dependencies are resolved by exhaustively moving the target classes from the core elements of the serverless functions to the local utilities of the microservice. As for the second and third types of dependencies, we delete those required from the core elements of microservices, consistently from the local utilities of microservices or the core elements of serverless functions as well, and add them to the global utilities.

8 | IMPLEMENTATION

RADF has been implemented as the architecture decomposition feature of the RADON methodology[12] 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

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 (noninheritance) 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 $\mathcal{S}$

output. Both input and output models conform to RADON Particles,[¶¶] a unified modeling profile that broadens the scope of TOSCA Simple YAML Profile v1.3[##] to serverless computing.

TOSCA[13] is an OASIS standard language for modeling cloud applications and their orchestration workflows. 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. 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.

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[|||] and the text analytics toolbox,[***] 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.

[¶¶] https://github.com/radon-h2020/radon-particles
[##] https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html
[|||] https://www.mathworks.com/help/stats/index.html
[***] https://www.mathworks.com/help/textanalytics/index.html

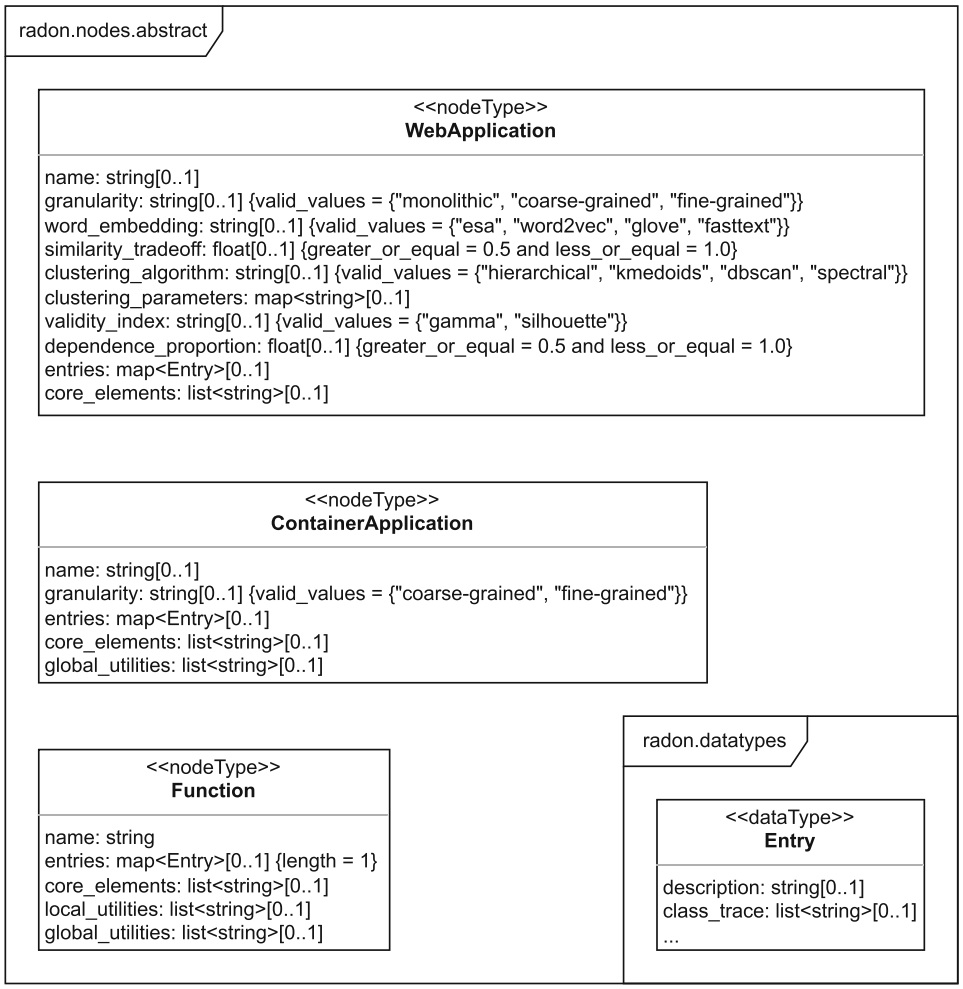

FIGURE 8 Definitions of the `WebApplication`, `ContainerApplication`, and `Function` node types in RADON particles.

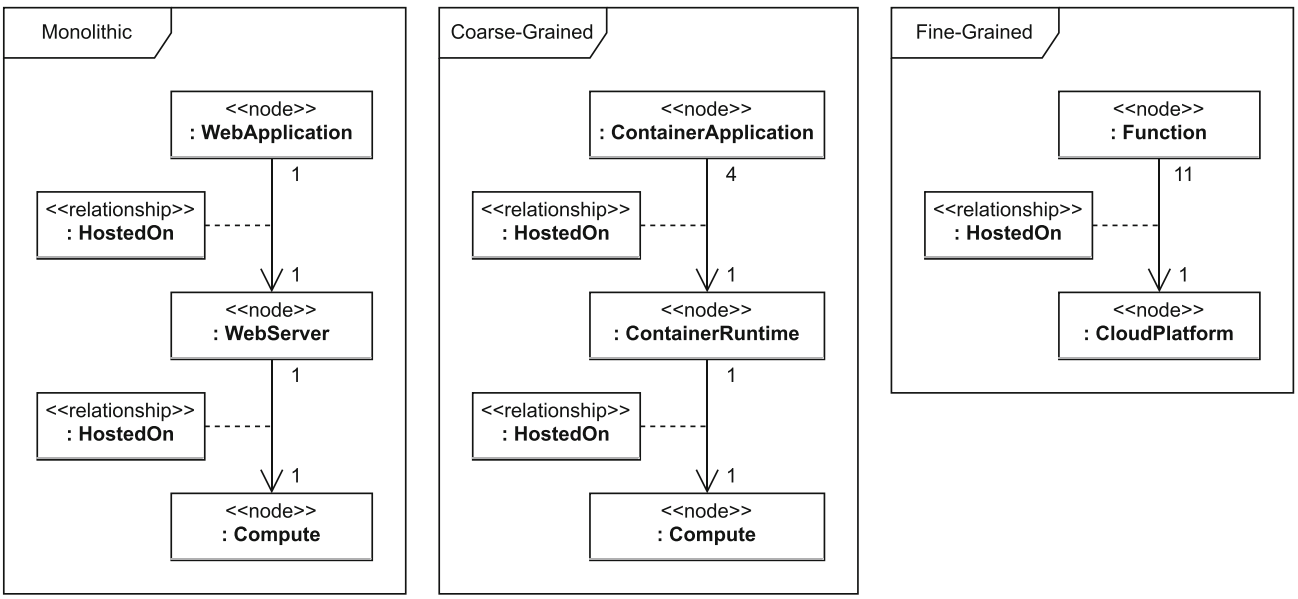

FIGURE 9 Topology and Orchestration Specification for Cloud Applications (TOSCA) models of the Cargo Shipping system at different granularity levels.

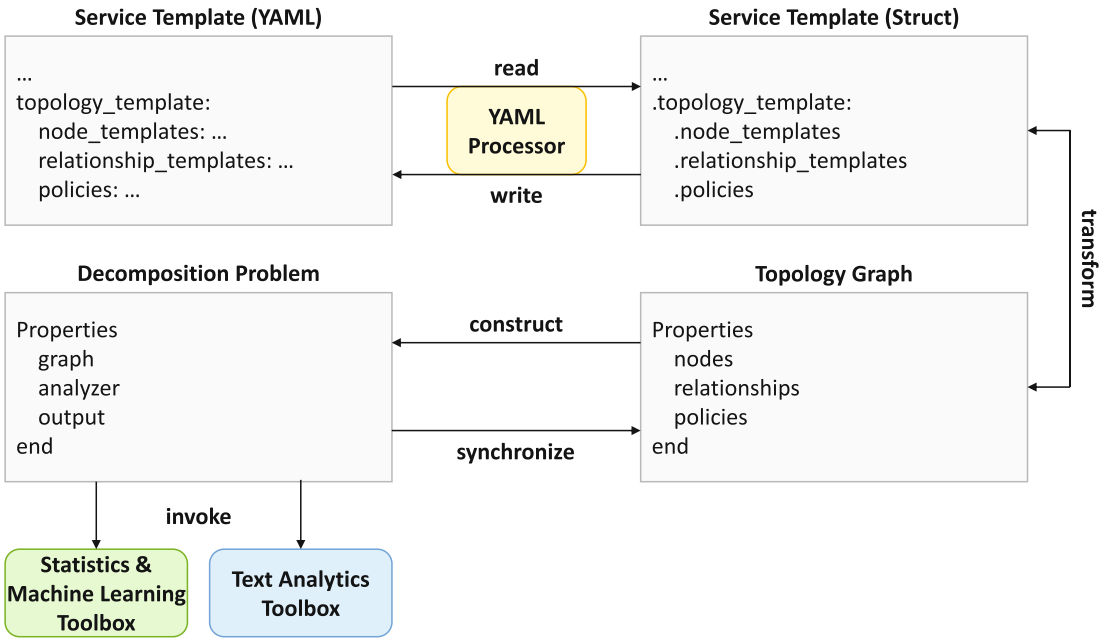

FIGURE 10 The architecture decomposition feature of the RADON methodology.

A few points are worthy of notice. Although we implement RADF in MATLAB, no ownership of a MATLAB license is required as the scripts are compiled into an executable running on MATLAB Runtime[†††] during the RADON project. One may alternatively come up with a Python implementation based on rich NLP and machine learning modules out there. To preserve language- and platform-agnosticities, 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, and represents the decomposition solution to a given application as a TOSCA model at the abstract level. The remainder of the activities have to be done manually with the help of language-specific tools for dynamic tracing and static dependency analysis. After code refactoring, the abstract decomposition solution can be further instantiated and deployed on a specific cloud platform using the graphical modeling and orchestration features of the RADON methodology.

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,[‡‡‡] Pet Clinic,[§§§] Polls App,[¶¶¶] Spring React Blog,[###] Spring Boot Blog,[‖‖‖] and Shopping Cart.[****] These are all Java applications designed in a monolithic architecture. As a

[†††] https://www.mathworks.com/products/compiler/matlab-runtime.html
[‡‡‡] https://github.com/mybatis/jpetstore-6
[§§§] https://github.com/spring-petclinic/spring-framework-petclinic
[¶¶¶] https://github.com/callicoder/spring-security-react-ant-design-polls-app
[###] https://github.com/keumtae-kim/spring-boot-react-blog
[‖‖‖] https://github.com/reljicd/spring-boot-blog
[****] https://github.com/reljicd/spring-boot-shopping-cart

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

Clustering algorithm	Clustering parameter	Optional values
Hierarchical	Definition of inter-cluster distance: $D(C_k, C_l)$	Equations (B1)[a], (B2)[a], (B3)[a]
	Number of clusters to discover: K	$[2..M-1]$[b]
K-Medoids	Number of clusters to discover: K	$[2..M-1]$[b]
	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	$[2..M-1]$[b]
	Scaling factor of the Gaussian kernel: σ	0.6006
	Number of times to repeat the clustering: R	5, 10, 20, 40, 80

[a]These corresponds to the single, complete, and average linkages, respectively.
[b]M denotes the total number of data points.

preparation, the interfaces of the applications are scanned and documented in a format similar to Tables A1 and A2. 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.[††††] 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,[16] pretrained on part of the Google News dataset. 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 the K-medoids algorithm multiple times and keep the partition that minimizes the objective function in Equation (B4). 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.

Coarse-grained decomposition solutions generated under different combinations of clustering algorithms and validity indices are assessed against the gold standard in terms of the F1 score originating from Reference 58. 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

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

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

[††††]https://github.com/btraceio/btrace

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

$P(C_k, C'_{k'})$ 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. Equation (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 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.

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.

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

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

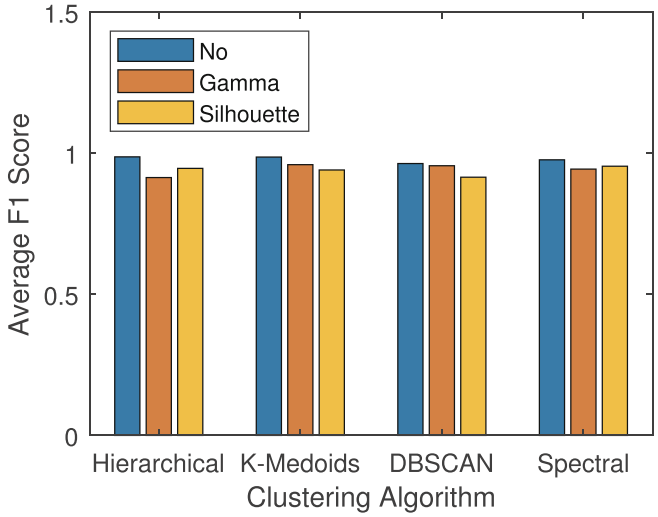

FIGURE 11 Average F1 scores of coarse-grained decomposition solutions arising from different combinations of clustering algorithms and validity indices.

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

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 methodology 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. Classes implementing the Cargo Shipping system are then reorganized by practicing the procedure described in Section 7 with a static code analysis tool called Sonargraph Architect.[‡‡‡‡] 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),[59] efferent coupling (EC),[59] instability (I),[59] and relational cohesion (RC),[60] as detailed in Table 8. The same application and quality metrics have been used in prior works to evaluate other decomposition approaches including Service Cutter,[23] concept mapping,[14] data flow-driven,[26] and multimodel-based.[30] We select these as the baseline approaches and collate their evaluation results from the literature to compare RADF with them.

Tables 9 reports the quality metrics of decomposition solutions obtained by applying different approaches to the Cargo Shipping system. 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 the others: 13.00 through concept mapping. Although the solution produced by RADF looks the most unstable, it is by no means inferior to the baseline approaches given that the high instability is largely due to a reduction in the afferent coupling instead of a growth in the efferent coupling. Unlike the others, RADF does not cause certain microservices to be loosely cohesive. Consider the worst cases in this respect. The

[‡‡‡‡] 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)[59]	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)[59]	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)[59]	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)[60]	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.

TABLE 9 Quality metrics of decomposition solutions obtained by applying different approaches to the Cargo Shipping system.

		Quality metric			
Approach	**Microservice**	**AC**	**EC**	**I**	**RC**
RADF	Cargo	2	7	0.78	11.7
	Planning	4	2	0.33	12.1
	Location	3	2	0.40	10.9
	Tracking	0	1	1.00	11.4
	Average	2.25	3.00	0.63	11.53
Service cutter[23]	Location	14	1	0.07	21.5
	Tracking	11	3	0.21	4.3
	Voyage and 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[26]	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
MultiModel-Based[30]	Preparation	23	0	0.00	14.0
	Handling	18	3	0.14	11.0
	Planning and 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

Abbreviations: AC, afferent coupling; EC, efferent coupling; I, instability; RC, relational cohesion.

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.

RADF decomposes a monolith into serverless functions. Quality metrics listed in Table 8 can also be used to quantify the coupling and cohesion of serverless functions within the scope of each identified microservice. Notably, the afferent coupling, efferent coupling and instability of any serverless function created by RADF are always zero. This is because Step 2 of the refactoring procedure introduced in Section 7 guarantees that no dependencies exist between the core elements of serverless functions composing the same microservice. The function-level relational cohesions for the Cargo, Planning, Location, and Tracking microservices are on average 4.40, 4.50, 1.00, and 2.57, respectively.

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 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 bidirectional encoder representations from transformers[61] or generative pre-trained transformer,[62] 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[56] and CDbw,[57] is a possible solution to mitigate this disagreement. Last but not the least, the evaluation experiments are performed on applications small in scale. Further research is needed to demonstrate the applicability of the proposed approach with a real case study.

AUTHOR CONTRIBUTIONS

Lulai Zhu: Conceptualization, investigation, methodology, implementation, evaluation, writing and editing. **Damian Andrew Tamburri:** Conceptualization, investigation, writing, review and work package leading. **Giuliano Casale:** Conceptualization, investigation, writing, review and project coordination.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for carefully reading the manuscript. Their comments have been of great help in improving its quality and clarity.

FUNDING INFORMATION

This work was supported by the European Union's Horizon 2020 research and innovation program under grant agreement No. 825040 (RADON).

CONFLICT OF INTEREST STATEMENT

The authors declare no potential conflict of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Zenodo at https://doi.org/10.5281/zenodo.7014842.

ORCID

Lulai Zhu https://orcid.org/0000-0003-2086-8467
Damian Andrew Tamburri https://orcid.org/0000-0003-1230-8961
Giuliano Casale https://orcid.org/0000-0003-4548-7951

REFERENCES

1. Loukis E, Arvanitis S, Kyriakou N. An empirical investigation of the effects of firm characteristics on the propensity to adopt cloud computing. *Inform Syst e-Bus Manage*. 2017;15(4):963-988.
2. Buyya R, Srirama SN, Casale G, et al. A manifesto for future generation cloud computing: research directions for the next decade. *ACM Comput Surv*. 2018;51(5):1-38.
3. Kratzke N. A brief history of cloud application architectures. *Appl Sci*. 2018;8(8):1368:1-1368:26.
4. Newman S. *Building Microservices*. O'Reilly Media; 2015.
5. Richardson C. *Microservices Patterns: With Examples in Java*. Simon & Schuster; 2018.
6. Roberts M, Chapin J. *What Is Serverless?* O'Reilly Media; 2017.
7. Jonas E, Schleier-Smith J, Sreekanti V, et al. Cloud programming simplified: A Berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* 2019.
8. Goli A, Hajihassani O, Khazaei H, Ardakanian O, Rashidi M, Dauphinee T. Migrating from monolithic to serverless: a FinTech case study. In: Amaral JN, Koziolek A, Trubiani C, Iosup A, eds. *Companion of the 11th ACM/SPEC International Conference on Performance Engineering*. ACM; 2020:20-25.
9. Lichtenthäler R, Prechtl M, Schwille C, Schwartz T, Cezanne P, Wirtz G. Requirements for a model-driven cloud-native migration of monolithic web-based applications. *Softw Intens Cyber Phys Syst*. 2020;35(1):89-100.
10. Stafford A, Toosi FG, Mjeda A. Cost-aware migration to functions-as-a-service architecture. In: Heinrich R, Mirandola R, Weyns D, eds. *Companion of the 15th European Conference on Software Architecture*. CEUR-WS; 2021.
11. 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.
12. Casale G, Artač M, Van De Heuvel WJ, et al. RADON: Rational decomposition and orchestration for serverless computing. *Softw Intens Cyber Phys Syst*. 2020;35(1):77-87.
13. Binz T, Breitenbücher U, Kopp O, Leymann F. TOSCA: portable automated deployment and management of cloud applications. In: Bouguettaya A, Sheng QZ, Daniel F, eds. *Advanced Web Services*. Springer; 2014:527-549.
14. Baresi L, Garriga M, De Renzis A. Microservices identification through interface analysis. In: De Paoli F, Schulte S, Johnsen EB, eds. *Proceedings of the 6th European Conference on Service-Oriented and Cloud Computing*. Springer; 2017:19-33.
15. Al-Debagy O, Martinek P. A new decomposition method for designing microservices. *Period Polytech Electric Eng Comput Sci*. 2019;63(4):274-281.
16. Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* 2013.
17. Bojanowski P, Grave E, Joulin A, Mikolov T. Enriching word vectors with subword information. *Trans ACL*. 2017;5:135-146.
18. Frey BJ, Dueck D. Clustering by passing messages between data points. *Science*. 2007;315(5814):972-976.
19. Rousseeuw PJ. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *J Comput Appl Math*. 1987;20:53-65.
20. Sun X, Boranbaev S, Han S, Wang H, Yu D. Expert system for automatic microservices identification using API similarity graph. *Expert Syst*. 2022;e13158.
21. Shi J, Malik J. Normalized cuts and image segmentation. *IEEE Trans Pattern Anal Mach Intell*. 2000;22(8):888-905.
22. Caliński T, Harabasz J. A dendrite method for cluster analysis. *Commun StatTheory Methods*. 1974;3(1):1-27.
23. Gysel M, Kölbener L, Giersche W, Zimmermann O. Service cutter: a systematic approach to service decomposition. In: Aiello M, Johnsen EB, Dustdar S, Georgievski I, eds. *Proceedings of the 5th European Conference on Service-Oriented and Cloud Computing*. Springer; 2016:185-200.
24. Mazlami G, Cito J, Leitner P. Extraction of microservices from monolithic software architectures. In: Altintas I, Chen S, eds. *Proceedings of the 24th IEEE International Conference on Web Services*. IEEE; 2017:524-531.
25. De Alwis AAC, Barros A, Polyvyanyy A, Fidge C. Function-splitting heuristics for discovery of microservices in enterprise systems. *Proceedings of the 16th International Conference on Service-Oriented Computing*. Springer; 2018:37-53.
26. Li S, Zhang H, Jia Z, et al. A dataflow-driven approach to identifying microservices from monolithic applications. *J Syst Softw*. 2019;157:110380.
27. 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.
28. Desai U, Bandyopadhyay S, Tamilselvam S. Graph neural network to dilute outliers for refactoring monolith application. In: Desai U, Bandyopadhyay S, Tamilselvam S, eds. *Proceedings of the 35th AAAI Conference on Artificial Intelligence*. AAAI; 2021:72-80.
29. Kalia AK, Xiao J, Krishna R, Sinha S, Vukovic M, Banerjee D. Mono2Micro: a practical and effective tool for decomposing monolithic Java applications to microservices. In: Spinellis D, Gousios G, Chechik M, Di Penta M, eds. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM; 2021:1214-1224.
30. Daoud M, El Mezouari A, Faci N, Benslimane D, Maamar Z, El Fazziki A. A multi-model based microservices Identiftcation approach. *J Syst Arch*. 2021;118:102200.

31. Freire AFAA, Sampaio AF, Carvalho LHL, Medeiros O, Mendonça NC. Migrating production monolithic systems to microservices using aspect oriented programming. *Softw Pract Exp*. 2021;51(6):1280-1307.
32. Raj V, Ravichandra S. A service graph based extraction of microservices from monolith services of service-oriented architecture. *Softw Pract Exp*. 2022;52(7):1661-1678.
33. Newman MEJ, Girvan M. Finding and evaluating community structure in networks. *Phys Rev E*. 2004;69(2):026113.
34. Raghavan UN, Albert R, Kumara S. Near linear time algorithm to detect community structures in large-scale networks. *Phys Rev E*. 2007;76(3):036106.
35. Spillner J, Dorodko S. Java code analysis and transformation into AWS lambda functions. *arXiv preprint arXiv:1702.05510* 2017.
36. Spillner J. Transformation of Python applications into function-as-a-service deployments. *arXiv preprint arXiv:1705.08169* 2017.
37. Carvalho dLR, Araújo dAPF. Framework Node2FaaS: automatic NodeJS application converter for function as a service. In: Muñoz VM, Ferguson D, Helfert M, Pahl C, eds. *Proceedings of the 9th International Conference on Cloud Computing and Services Science*. SciTePress; 2019:271-278.
38. Yussupov V, Breitenbücher U, Hahn M, Leymann F. Serverless parachutes: preparing chosen functionalities for exceptional workloads. In: Ceballos C, Torres H, eds. *Proceedings of the 23rd IEEE International Enterprise Distributed Object Computing Conference*. IEEE; 2019:226-235.
39. Zhao Z, Wu M, Tang J, Zang B, Wang Z, Chen H. BeeHive: sub-second elasticity for web services with semi-FaaS execution. In: Aamodt TM, Enright Jerger ND, Swift MM, eds. *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM; 2023:74-87.
40. Ghemawat S, Grandl R, Petrovic S, et al. Towards modern development of cloud applications. In: Schwarzkopf M, Baumann A, Crooks N, eds. *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. ACM; 2023:110-117.
41. Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley; 2004.
42. Jurafsky D, Martin JH. *Speech and Language Processing*. Prentice Hall; 2009.
43. Firoozeh N, Nazarenko A, Alizon F, Daille B. Keyword extraction: issues and methods. *Nat Lang Eng*. 2020;26(3):259-291.
44. Pennington J, Socher R, Manning CD. GloVe: global vectors for word representation. In: Moschitti A, Pang B, Daelemans W, eds. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. ACL; 2014:1532-1543.
45. Harris ZS. Distributional structure. *Word*. 1954;10(2):146-162.
46. Manning CD, Raghavan P, Schütze H. *An Introduction to Information Retrieval*. Cambridge University Press; 2008.
47. Lilleberg J, Zhu Y, Zhang Y. Support vector machines and Word2vec for text classification with semantic features. In: Ge N, Lu J, Wang Y, Howard N, Chen P, Tao X, Zhang B, Zadeh LA, eds. *Proceedings of the 14th IEEE International Conference on Cognitive Informatics and Cognitive Computing*. IEEE; 2015:136-140.
48. Kruskal JB. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*. 1964;29(1):1-27.
49. Xu D, Tian Y. A comprehensive survey of clustering algorithms. *Ann Data Sci*. 2015;2(2):165-193.
50. Johnson SC. Hierarchical clustering schemes. *Psychometrika*. 1967;32(3):241-254.
51. Kaufman L, Rousseeuw PJ. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons; 1990.
52. Ester M, Kriegel HP, Sander J, Xu X. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Simoudis E, Han J, Fayyad UM, eds. *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*. AAAI; 1996:226-231.
53. Dunn JC. Well-separated clusters and optimal fuzzy partitions. *J Cybern*. 1974;4(1):95-104.
54. Baker FB, Hubert LJ. Measuring the power of hierarchical cluster analysis. *J ASA*. 1975;70(349):31-38.
55. Davies DL, Bouldin DW. A cluster separation measure. *IEEE Trans Pattern Anal Mach Intell*. 1979;PAMI-1(2):224-227.
56. Krzanowski WJ, Lai YT. A criterion for determining the number of groups in a data set using sum-of-squares clustering. *Biometrics*. 1988;44(1):23-34.
57. Halkidi M, Vazirgiannis M. A density-based cluster validity approach using multi-representatives. *Pattern Recogn Lett*. 2008;29(6):773-786.
58. Larsen B, Aone C. Fast and effective text mining using linear-time document clustering. In: Fayyad UM, Chaudhuri S, Madigan D, eds. *Proceedings of the 5th ACM International Conference on Knowledge Discovery and Data Mining*. ACM; 1999:16-22.
59. Martin RC. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall; 2003.
60. Larman C. *Applying UML and Patterns: an Introduction to Object Oriented Analysis and Design and Interactive Development*. Pearson; 2012.
61. Devlin J, Chang MW, Lee K, Toutanova K. BERT: pre-training of deep bidirectional transformers for language understanding. In: Burstein J, Doran C, Solorio T, eds. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*. ACL; 2019:4171-4186.
62. Radford A, Narasimhan K, Salimans T, Sutskever I. Improving language understanding by generative pre-training. *OpenAI* 2018.
63. Lance GN, Williams WT. A general theory of classificatory sorting strategies: 1. *Hierarch Syst Comput J*. 1967;9(4):373-380.
64. MacQueen J. Some methods for classification and analysis of multivariate observations. In: Le Cam LM, Neyman J, eds. *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*. University of California Press; 1967:281-297.
65. Von Luxburg U. A tutorial on spectral clustering. *Stat Comput*. 2007;17(4):395-416.

How to cite this article: Zhu L, Tamburri DA, Casale G. RADF: Architecture decomposition for function as a service. *Softw Pract Exper*. 2023;1-29. doi: 10.1002/spe.3276

APPENDIX A. INTERFACE OF RUNNING EXAMPLE

TABLE A1 Definitions of API operations exposed by the Cargo Shipping system (Part 1).

ID	Path	Method	Parameter/body/response[a]
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

[a] 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 A2 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),[50] where

$$D(C_k, C_l) = \min_{x_i \in C_k, x_j \in C_l} d(x_i, x_j); \tag{B1}$$

- the complete linkage (i.e., the longest distance),[50] where

$$D(C_k, C_l) = \max_{x_i \in C_k, x_j \in C_l} d(x_i, x_j); \tag{B2}$$

- the average linkage (i.e., the average distance),[63] where

$$D(C_k, C_l) = \frac{1}{|C_k||C_l|} \sum_{x_i \in C_k} \sum_{x_j \in C_l} d(x_i, x_j). \tag{B3}$$

B.2 *K*-Medoids

K-medoids is a variant of the celebrated K-means algorithm.[64] Its basic idea is to search for K medoids $o_1, o_2, \ldots, o_K$ such that the total distance from each data point x_i to the closest medoid is minimized:

$$\begin{aligned} \min_{o_1, o_2, \ldots, o_K} \quad & \sum_{i=1}^{M} \min_{k \in [1..K]} d(x_i, o_k), \\ \text{s.t.} \quad & \forall k \in [1..K], o_k \in \{x_1, x_2, \ldots, x_M\}, \end{aligned} \tag{B4}$$

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 (B5) is however NP-hard to solve exactly. We obtain approximate solutions to it resorting to a widely applied heuristic named partitioning around medoids (PAM)[51] 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 (B5) decreases most, until the medoids no longer change.

B.3 DBSCAN

DBSCAN[52] is a nonparametric 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 $x_i, \ldots, x_j$ 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, which are outliers not directly density-reachable from any core points.

B.4 Spectral clustering

Spectral clustering models a dataset 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.[65] We adapt the normalized spectral clustering described in Reference 21 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

$$w(x_i, x_j) = e^{-\left(\frac{d(x_i,x_j)}{\sigma}\right)^2}, \tag{B6}$$

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:

$$\boldsymbol{L}^{\mathrm{RW}} = \boldsymbol{I} - \boldsymbol{D}^{-1}\boldsymbol{A}, \tag{B7}$$

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 dataset in the new representation.

APPENDIX C. VALIDITY INDICES

C.1 Gamma index

The gamma index[54] 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:

$$N_+ = \sum_{(x_i,x_j)\in\mathcal{P}_{\mathrm{W}}} \sum_{(x_{i'},x_{j'})\in\mathcal{P}_{\mathrm{B}}} \mathbb{1}_{\{d(x_i,x_j)<d(x_{i'},x_{j'})\}}, \tag{C1}$$

where $\mathcal{P}_{\mathrm{W}}$ and $\mathcal{P}_{\mathrm{B}}$ are the sets of pairs of data points within the same cluster and between different clusters, respectively, that is, $\mathcal{P}_{\mathrm{W}} = \{(x_i, x_j) \in \mathcal{P} | \exists k \in [1..K], x_i \in C_k \wedge x_j \in C_k\}$ and $\mathcal{P}_{\mathrm{B}} = \{(x_i, x_j) \in \mathcal{P} | \nexists k \in [1..K], x_i \in C_k \wedge x_j \in C_k\}$ with $\mathcal{P} = \{(x_i, x_j) \in (\bigcup_{k=1}^{K} C_k)^2 | i < j\}$. Let N_- be the number of times that the opposite situation occurs:

$$N_- = \sum_{(x_i,x_j)\in\mathcal{P}_{\mathrm{W}}} \sum_{(x_{i'},x_{j'})\in\mathcal{P}_{\mathrm{B}}} \mathbb{1}_{\{d(x_i,x_j)>d(x_{i'},x_{j'})\}}. \tag{C2}$$

The gamma index can then be written as

$$\Gamma = \frac{N_+ - N_-}{N_+ + N_-}, \tag{C3}$$

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[19] 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

$$a(x_i) = \frac{1}{|C_k| - 1} \sum_{\substack{x_j \in C_k \\ j \neq i}} d(x_i, x_j), \tag{C4}$$

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

$$b(x_i) = \min_{\substack{l \in [1..K] \\ l \neq k}} \frac{1}{|C_l|} \sum_{x_j \in C_l} d(x_i, x_j). \tag{C5}$$

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

$$s(x_i) = \begin{cases} 1 - \frac{a(x_i)}{b(x_i)} & \text{if} a(x_i) < b(x_i), \\ \frac{b(x_i)}{a(x_i)} - 1 & \text{if} a(x_i) > b(x_i), \\ 0 & \text{otherwise}, \end{cases} \tag{C6}$$

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:

$$\bar{s} = \frac{1}{K} \sum_{i=1}^{K} \frac{1}{|C_k|} \sum_{x_i \in C_k} s(x_i). \tag{C7}$$

AUTHOR BIOGRAPHIES

Lulai Zhu received the B.Eng. and M.Eng. degrees in measurement technology and instruments from Sichuan University, China, in 2008 and 2011, and the MSc and PhD degrees in computing science from Imperial College London, UK, in 2016 and 2022, respectively. He is currently a postdoctoral researcher with the Department of Mechanical Engineering at Politecnico di Milano, Italy. His research focuses on model-driven performance analysis and architecture design of cloud-native applications, and automated generation and tuning of digital twins for complex manufacturing systems.

Damian Andrew Tamburri is an Associate Professor with the Jheronimus Academy of Data Science at Eindhoven University of Technology, The Netherlands, double affiliated with Politecnico di Milano, Italy. His research interests rotate around DevOps/DataOps, social software engineering, and AI software engineering. He has published over 200 papers in the top journals and conferences of these fields. He has also been an active contributor and led research in many EU FP6, FP7, and Horizon 2020/Europe projects, such as S-Cube, MODAClouds, SeaClouds, DICE, ANITA, DossierCLOUD, ProTECT, RADON, SODALITE, and DESTINI. In addition, he is an associate editor of ACM TOSEM and the secretary of OASIS TOSCA TC as well as that of IFIP TC2, TC6, and TC8 on service-oriented computing.

Giuliano Casale joined the Department of Computing at Imperial College London, UK, in 2010, where he is currently a Reader. He does research in quality of service and cloud computing, topics on which he has published more than 150 refereed papers. He has been the technical program committee member for over 100 venues and the program co-chair for several conferences in the area of performance and reliability engineering, such as ACM SIGMETRICS/IFIP Performance and IEEE/IFIP DSN. His research work has received multiple recognitions, including the best paper awards at ACM SIGMETRICS and IEEE INFOCOM. During the years 2019–2023, he was the chair of ACM SIGMETRICS. He serves on the editorial boards of IEEE TNSM and ACM TOMPECS, and as the editor-in-chief of Elsevier Performance Evaluation.