Microservices Identification Through Interface Analysis

Luciano Baresi[1], Martin Garriga[1(✉)], and Alan De Renzis[2]

[1] Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy
{luciano.baresi,martin.garriga}@polimi.it
[2] Faculty of Informatics, National University of Comahue, Neuquén, Argentina
alanderenzis@fi.uncoma.edu.ar

Abstract. The microservices architectural style is gaining more and more momentum for the development of applications as suites of small, autonomous, and conversational services, which are then easy to understand, deploy and scale. One of today's problems is finding the adequate granularity and cohesiveness of microservices, both when starting a new project and when thinking of transforming, evolving and scaling existing applications. To cope with these problems, the paper proposes a solution based on the semantic similarity of foreseen/available functionality described through OpenAPI specifications. By leveraging a reference vocabulary, our approach identifies potential candidate microservices, as fine-grained groups of cohesive operations (and associated resources). We compared our approach against a state-of-the-art tool, sampled microservices-based applications and decomposed a large dataset of Web APIs. Results show that our approach is able to find suitable decompositions in some 80% of the cases, while providing early insights about the right granularity and cohesiveness of obtained microservices.

Keywords: Microservices · Microservice architecture · Monolith decomposition

1 Introduction

Microservices is a novel architectural style that tries to overcome the shortcomings of centralized, monolithic architectures [1,2], in which the application logic is encapsulated in big deployable chunks. The most widely adopted definition of a microservices architecture is "an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often a RESTful API" [3]. In contrast to monoliths, microservices foster independent deployability and scalability, and can be developed using different technology stacks [4,5].

F. De Paoli et al. (Eds.): ESOCC 2017, LNCS 10465, pp. 19–33, 2017.
DOI: 10.1007/978-3-319-67262-5_2

Although microservices can be seen as an evolution of Service-Oriented Architectures (SOA), they are inherently different regarding sharing and reuse [6]: given that service reuse has often been less than expected [7], instead of reusing existing microservices for new tasks or use cases, they should be small and independent enough to allow for rapidly developing a new one that can coexist, evolve or replace the previous one according to the business needs [1].

Several companies have recently migrated, or are considering migrating, their existing applications to microservices [8], and new microservice-native applications are being conceived. While the adoption of this architectural style should help one address the typical facets of a modern software system: for example, its distribution, coordination among parts, and operation, some aspects are still blurred [9,10]. One key issue is the definition of the right granularity level, that is, the trade-off between size and number of microservices [1].

The problem is not new: the literature has already addressed the *decomposition* problem—for identifying modules, packages, components, and "traditional" services—mainly by means of clustering techniques upon design artifacts [11] or source code [12]. However, the boundaries between software modules settled by these approaches were too flexible and allowed software to evolve into "big balls of mud" [13]. Microservices make these boundaries physical, and their unique characteristics in terms of distribution, granularity, and independent deployability, call for a new wave of techniques. Notwithstanding the existing body of knowledge, the elicitation of strong interface boundaries at the right level of granularity, along with proper tool support, remains an important challenge inherited from the early times of SOA [14]. The identification of "proper" microservices not only aims to partition the system to ease maintenance [7], but also defines how the system will be able to evolve and scale.

This paper borrows from the aforementioned experiences to introduce a novel approach to reason on microservices starting from an initial OpenAPI specification [15] (a language-agnostic, machine-readable interface for REST APIs) of the operations that the application should offer. This means that either the application, along with its interfaces, already exists and it must be re-engineered, or some design artifacts/specifications are available.

The process starts with mapping available OpenAPI specifications onto the entries of a reference vocabulary by means of a fitness function. In this paper, we use `Schema.org`[1] as reference, but any other shared vocabulary or even a domain-specific ontology would be appropriate. The fitness function is based on DISCO (DIStributionally related words using CO-occurrences, [16]), a precomputed database of collocations and distributionally similar words that allows for computing the semantic similarity of terms according to their co-occurrences in large corpora of text. The goal is to provide a usable, automated solution to devise a decomposition—that is, a set of candidate microservices defined by groups of operations and their associated resources. The idea is to pair standardized (OpenAPI) specifications with homogeneous—because of the shared reference vocabulary—semantic characterizations. The reference vocabulary also

[1] http://Schema.org/docs/full.html.

act as a context that allows us to address large domains, in which certain concepts are used with different meanings across the system. The main properties driving the decomposition are granularity (a tradeoff between size and number of microservices), loose coupling (minimising inter-service calls) and high cohesion (keeping similar functionality together), while allowing the user to explore different alternatives by tunning the procedure.

In summary, the contribution of this work is an automated process for identifying candidate microservices by means of a lightweight, domain-agnostic semantic analysis of the concepts in the input specification with regard to a reference vocabulary.

The rest of this paper is organized as follows. Section 1.1 presents an example application to illustrate our approach. Section 2 introduces the main technologies used throughout the paper. Section 3 presents our approach for identifying microservices. Section 4 discusses the experimental validation. Section 5 surveys related work and Sect. 6 concludes the paper.

1.1 Example Application: Cargo Tracking

Figure 1 shows a simplified class diagram (domain model) of Cargo Tracking[2], a well-known example application [17] used to illustrate the approach. Each class defines a key concept and introduces a first set of attributes and operations.

The main focus of the application is to move a `Cargo` (identified by a `TrackingId`) between two `Locations` through a `RouteSpecification`. Once a `Cargo` becomes available, it is associated with one of the `Itineraries` (lists of `CarrierMovements`), selected from existing `Voyages`. `HandlingEvents` then trace the progress of the `Cargo` on the `Itinerary`. The `Delivery` of a `Cargo` informs about its state, estimated arrival time, and being on track.

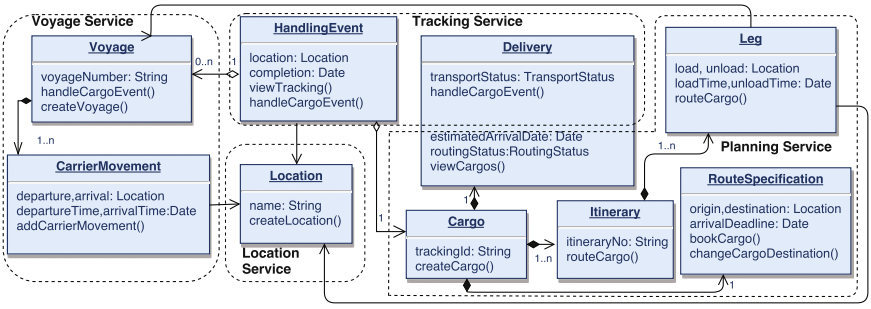

Fig. 1. Domain model and expected decomposition (dotted boxes) of the Cargo Tracking application.

[2] https://github.com/citerus/dddsample-core (Java implementation).

2 Background

DISCO [16] is a pre-computed database of collocations and distributionally similar words. The similarities are based on the statistical analysis of very large text collections (e.g., Wikipedia), through co-occurrence functions. For each word, DISCO indexes the first and second order vectors of related words.

The similarity between two words is then obtained by computing the similarity—based on co-occurrences—of the corresponding vectors. The highest the similarity value ($[0, 1]$) is, the closer the two words are. For example, if *bread* co-occurred with *bake*, *eat*, and *oven*, and *cake* also co-occurred with these three words, then *bread* and *cake* would be distributionally similar [16], and their similarity value would be 1 (if the vectors only comprised the three words).

OpenAPI, formerly known as Swagger[3], is a machine-readable, language-agnostic interface for RESTful APIs. Although OpenAPI can be seen as yet another attempt to define Web Service interfaces, it is just intended to describe RESTful APIs, and is supported by major industry partners such as Google, IBM, Microsoft, and PayPal. OpenAPI follows a JSON-based format[4] and is modular and extensible by means of the `$ref` keyword, with the goal of linking elements to concepts in a shared schema, or even a reference vocabulary. The elements/objects tagged with keyword `$ref` are then linked to a concept in a certain schema, which can be based on high level vocabularies, such as FOAF[5] or `Schema.org`. For example:

```
{"name":"Cargo",
 "description": "A cargo (product) identified by its TrackingId.",
   "schema": {
    "$ref": "schema.org.apis.apievangelist.com/api-commons/product/
                openapi-spec.json"
}}
```

says that `Cargo` is a `Product`, as defined in `Schema.org`. That is, all the attributes defined for type `Product` in the reference vocabulary are then usable in this description, and any external (automated) client can easily exploit them.

3 Our Approach

The identification process consists of matching the terms used in the OpenAPI specifications supplied as input against a reference vocabulary to suggest possible decompositions. Note that when OpenAPI specifications are not available beforehand, they can be automatically generated from existing interface specifications[6] The terms extracted from input artifacts are iteratively mapped on the

[3] http://swagger.io.

[4] Developers can thus exploit OpenAPI through the same tools and libraries used for JSON (e.g., Jackson).

[5] http://xmlns.com/foaf/spec/.

[6] E.g., the APIMatic tool (https://apimatic.io/transformer) accepts Swagger, WSDL, WADL and RAML among others.

concepts in the vocabulary by means of a fitness function based on the semantic similarity measure provided by DISCO. The best concept mappings are obtained through maximization of a co-occurrence matrix that contains all the possible pairs of terms and concepts.

Algorithm 1. Decomposition Algorithm

```
   Data: OpenAPI specs, ref. vocabulary
   Result: OpenAPI microservices' specifications
1  mappings ← ∅ ;
2  foreach input specification do
3  |  map ← SemanticAssessment(specification,vocabulary);
4  |  mappings ← mappings + map;
5  end
6  candidateMS ← GroupSimilar(mappings,vocabulary,level);
7  microserviceSpecs ← GenerateOpenApiSpecs(candidateMS, vocabulary);
8  return microserviceSpecs
```

Algorithm 1 summarizes the main steps of the decomposition algorithm. It receives a set of OpenAPI specifications and the reference vocabulary as input, and computes the best mappings between them through the DISCO-based semantic assessment algorithm (Line 3), further detailed later. This step generates a mapping between each operation in the input and a *reference concept* in the vocabulary, that is, the concept that most accurately describes the operation. The idea is that operations that share the same reference concept are highly cohesive, and should be grouped together (Line 6). Parameter `level`[7] determines the granularity of these groupings, that is, the level of interest in the hierarchy of concepts. For example, `level=0` would only generate one candidate microservice, since everything would be grouped up to the root node of the vocabulary—`Thing` in `Schema.org`. The empirical assessment (Sect. 4), allowed us to set `level` to 2 to achieve a good compromise between the number of microservices and their granularity. Needless to say, the user can play with different values for `level`, identify different groupings, and analyze them.

Then, the suggested decomposition (Line 6) comprises one candidate microservice per identified reference concept. Each microservice is defined through its operations and their parameters, (public) complex types, and return values.

For example, if we started from the operations in Fig. 1 for the Cargo Tracking application, the process of Algorithm 1 would map `Delivery` and `Handling` onto `DeliveryEvent` (in `Schema.org`), and they would share the latter as reference concept. `Delivery` and `Handling` should then be part of the same candidate microservice, which could be named, for instance, `EventTracker`.

The OpenAPI specification of microservice `EventTracker` would then contain the operations defined within `Delivery` and `Handling`, and also a reference to the corresponding "shared" concept. The complete results for the case study are discussed in Sect. 4.

[7] Its values can range from 0 to the maximum depth of the vocabulary tree, which is 5 in Schema.org.

Algorithm 2 details the DISCO-based semantic assessment, called at Line 3 of the decomposition algorithm (Algorithm 1). It analyzes each operation of a specification artifact, along with the resources it defines (parameters, return value, complex types), with respect to the concepts in the shared vocabulary. The algorithm uses a robust term separator[8] [18] to identify and split words in the input terms (T) even when identifiers do not strictly follow any predefined naming convention (Line 3). The term separator also filters *stop words*[9], that is, meaningless words such as articles, pronouns, prepositions, digits, single alphabet characters, and possibly further domain- or context-specific words.

Then, the algorithm iteratively maps the set of input terms T onto all possible concepts C in the vocabulary by using DISCO (Line 5 to 8). For example, let us consider term `CargoTracking` and concept `DeliveryEvent`, with the following similarity scores:

	Cargo	Tracking
Delivery	0.3	0.1
Event	0.2	0.1

At a first glance, the best mappings are $(cargo, delivery)$ and $(cargo, event)$ with overall $score = (0.3 + 0.2)/2 = 0.25$. However, this mapping is not valid since it would consider word `Cargo` twice, but it would not use `Tracking`, and thus it would not be an acceptable mapping for the whole term. We must then find a suitable set of mappings that cover all the words in t and maximize the overall mapping score. When both t and c contain multiple words, finding the best mapping is not trivial, since it should consider all the words in t. This is done by applying the fitness function (Formula 1), followed by the Hungarian algorithm [19], a classical algorithm that solves the assignment problem in $O(n^3)$. As said, both t and c can be composed of multiple words (as `CargoTracking` and `DeliveryEvent`). $col(t_i, c_j)$ is the set of collocation scores for pairs of words $(t_i, c_j) \in (t, c)$, and N is the number of collocations between the different words in t and c that conform to the mapping (e.g., if t and c contain two words, then $N = 2$ since there can only be two possible valid mappings with two pairs each). Values range from 0 to 1, given the range of DISCO similarity function and the normalization factor N. The highest *col* is, the closest the two terms are. Note that although *col* ranges between 0 and 1, values are in general closer to 0, since $col = 1$ would mean that all the words appear together for all their occurrences in the DISCO corpus, which is highly unlikely in practice [16]. Scores are stored in a correlation matrix, where each column is a word in t and each row corresponds to a word in c linked to at least an element in t. Finally, the algorithm uses the matrix (Line 9) to identify the most adequate mappings.

$$score(t, c) = \sum (col(t_i, c_j))/N \tag{1}$$

[8] https://github.com/aderenzis/IdentifiersTermSeparator.
[9] http://www.webconfs.com/stop-words.php.

In the end, the concept in the reference vocabulary with the highest mapping score for a given input operation is elected as *reference concept*. The algorithm then returns a list with the best mapping for each operation in the input specification.

Back to the running example, for operation `CreateCargo` defined in `Cargo`, the concept in the vocabulary that shares the highest similarity according to DISCO is `Vehicle`, where: $(col(Create, Vehicle) = 0.07 + col(Cargo, Vehicle) = 0.61)/2 = 0.34$. Then, given the desired grouping granularity `Vehicle` can also become a `Product` in the vocabulary hierarchy. Since `Cargo` in Fig. 1 only shows one operation, it is grouped under `Product` as reference concept.

Algorithm 2. Semantic Assessment Algorithm

```
   Data: OpenAPI specification, ref. vocabulary
   Result: best mappings
 1 bestMappings ← ∅ ;
 2 foreach operation in specification do
 3 |   termsInput ← TermSeparation(operation);
 4 |   correlationMatrix ← [][];
 5 |   foreach concept in vocabulary do
 6 |   |   termsContext ← TermSeparation(concept);
 7 |   |   correlationMatrix ← DiscoCoOcurrrences(termsIput,termsContext);
 8 |   end
 9 |   bestMappings ← bestMappings + hungarianMax(correlationMatrix);
10 end
11 return bestMappings
```

4 Evaluation

This section presents the experiments we conducted to assess and validate the approach[10].

4.1 Decomposition of the Cargo Tracking Application

We performed the decomposition of the cargo tracking application (presented in Sect. 2), and compared our approach against Service Cutter [20], a state-of-the-art tool for microservice decomposition. The dotted boxes in Fig. 1 (Sect. 2) show the expected decomposition for the cargo tracking application (as defined in [20]). The input to our tool is an OpenAPI specification of the application that describes its different interfaces, operations, and resources. `Schema.org` is given as reference vocabulary. Figure 3 presents the candidate decomposition we obtained. As examples, we can take a closer look at some mappings. For interface `Voyage`, its operation `CreateVoyage` was mapped to the reference concept `Trip`, which is in turn an `Intangible` in `Schema.org`. Analogously, operation `RouteCargo` of interface `Leg` is also mapped to the reference concept `Trip`. Thus, these two operations will be grouped together in the candidate microservice `PlanningService`, along with all the other operations mapped

[10] Both the experimental prototype of the decomposition tool and the datasets used are available here: https://github.com/mgarriga/decomposer.

to `Trip` or other `Intangibles`. In turn, the remaining operation in `Voyage` is `HandleCargoEvent`, which is mapped to reference concept `Event`. This operation will be grouped under another candidate microservice named `EventTracker`, with the other operations also mapped to `Event` (or other concepts under `Event` in `Schema.org`), such as `ViewCargos` (from `Delivery`) and `ViewTrackings` (from `HandlingEvent`).

The input to Service Cutter is a set of specification artifacts, and a set of weighted coupling criteria, and the output is a graph where nodes represent candidate microservices, and weighted arcs indicate how cohesive and/or coupled two candidates are. Finally, a clustering algorithm provides the most suitable service cuts. Figure 2 depicts the best decomposition provided by Service Cutter, after manually prioritizing and fine-tuning the weights of coupling criteria to reflect the requirements of the application.

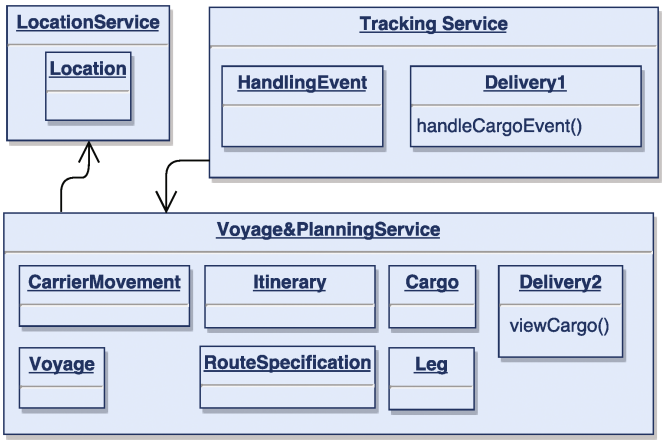

Fig. 2. Obtained decomposition with Service Cutter [20]

Our microservice decomposition process generated different candidate microservices than those obtained with Service Cutter. No approach returned the "expected" service decomposition, although it was defined manually in [20]. Thus, one can argue whether the expected decomposition is optimal, since it may be subjective, and biased by certain design decisions. From a comparative perspective, the main difference is service `Voyage&Planning` (Fig. 2) which in Service Cutter's decomposition encapsulates seven input artifacts, nine operations and two different business aspects. In contrast, our solution decomposes it in three different microservices (Fig. 3): `Trip`, `Planning` and `EventTracking`, all with a similar and finer granularity (three, four and five operations respectively). The only candidate microservice that could be too fine-grained is `Cargo`, which only encapsulates one operation.

From a comparative perspective, our approach requires as input the reference vocabulary and the OpenAPI descriptions of the interfaces (which can be

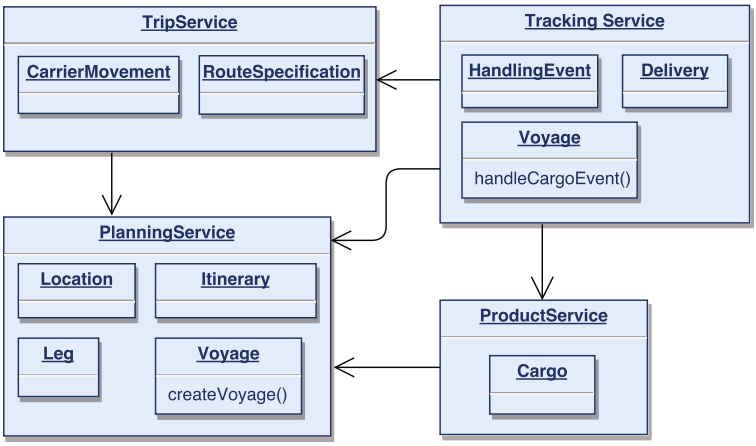

Fig. 3. Obtained decomposition with our approach

automatically generated from other descriptions). In turn, Service Cutter requires a detailed and exhaustive specification of the system, together with ad-hoc specification artifacts associated with coupling criteria [20]. The availability of such a broad range of documentation is, at least, arguable.

This section provided insights about the rationale of our approach and a comparison with a state-of-the-art-tool through a simple example. The experiments described in the next section use real-life microservice applications and a broader dataset of real-world Web APIs to help us better devise the feasibility of our approach.

4.2 Decomposition of Microservice Applications

The goal of the second experiment is to automatically devise adequate decompositions of two microservice-based applications[11]: Money Transfer, composed of four microservices (Customers, Accounts, Transfer, and Login) and Kanban Board, composed of three microservices (Boards, Tasks, and Authentication).

The original microservice architecture of each application acts as a gold standard to validate the results obtained with our approach. Again, we used the OpenAPI specifications as input—a single JSON per application, that acts as its "monolithic-like" description—and `Schema.org` as vocabulary.

Table 1 shows the decompositions for both applications. Each group of operations constitutes a different candidate microservice. Then, the rightmost column indicates if the mapping is adequate in the context of each decomposition, that is, whether the grouped operations corresponded to the same microservice in the original architecture.

[11] http://eventuate.io/exampleapps.html – from the curator of microservices.io [8].

Particularly, for MoneyTransfer, 8 operations out of 10 (80%) were correctly decomposed, that is, as prescribed in the original architecture. For example, operation `getAccountForCustomer` was correctly placed in microservice `Account` despite containing also terms of `Customer`. This is based on the co-occurrences criteria and the use of a reference vocabulary to provide contextual information to the concept analysis. This can be illustrated also by considering an operation with completely different terms, e.g., `getStatement`, which would be grouped into microservice `Account` since `Account` and `Statement` are highly correlated according to DISCO (0.48 as similarity value). For the two remaining operations, `getCustomersByEmail` was placed in another candidate microservice, while `transactionsHistory` was not mapped to any concept of `Schema.org`, since the relationships found are too weak (according to the defined threshold) to devise a similarity.

In turn, for KanbanBoard, 10 operations out of 13 (77%) were correctly decomposed. As for the three remaining operations, they were grouped together in another candidate microservice. Obtained results suggest that our approach is able to detect correct candidate microservices for around 80% of an application's functionality, given that the expected decomposition (gold standard) was known beforehand.

Table 1. Obtained decomposition for MoneyTransfer and KanbanBoard.

Application	Cand. microservice	Operation	Suitable?
Money Transfer	Customer	`createCustomer,getCustomer, getCurrentUser`	Yes
	Account	`getAccountsForCustomer addToAccount,createAccount`	Yes
	Login	`doAuthorization`	Yes
	MoneyTransfer	`moneyTransfer`	Yes
	Other	`getCustomersByEmail`	No
	N/A	`transactionsHistory`	No
		Total	8/10
Kanban Board	Task	`listAllTasks,saveTask,update-Task,deleteTask,backlogTask, completeTask,getTaskHistory`	Yes
	Auth	`doAuthentication`	Yes
	Board	`listAllBoards,getBoard`	Yes
	Other	`readAction,scheduleAction,resumeAction`	No
		Total	10/13

4.3 Decomposition of a Large Dataset of Real-World APIs

The goal of this experiment is to decompose a dataset of real-world APIs and analyze the potential applicability/utility of our approach. Moreover, this is

helpful to profile the decomposition process and find its optimal configuration, according to expected decompositions defined by software engineers. We used a dataset of OpenAPI specifications from APIs.Guru[12], currently the largest repository of publicly available, real-world OpenAPI specifications. From all the APIs available in the repository (550 in total), we focused on specifications with at least two operations, which is the minimal condition to be potentially decomposable, and less than fifty operations, which avoids the noise introduced by too large APIs. We ended up with a dataset of 452 OpenAPI specifications defining a total of 6634 endpoints, which are equivalent to the notion of operations in this paper.

From this dataset, we randomly selected 5 samples of 14 services, that were delivered to five different software engineers (both PhD. students and researchers in software engineering with industry experience). Then the engineers manually defined the decompositions for these services. Note that the engineers were unaware of the rationale behind our approach, to avoid biasing their answers. We configured different similarity *thresholds over the fitness function* (Formula 1) *and different values for the grouping level* (Algorithm 1) and executed the decomposition over the sample services, comparing our candidate microservices with those suggested by the developers. The results were measured in terms of *precision* and *recall*, according to the expected and achieved decompositions. Figure 4 shows the precision/recall curve that considers an average of the different samples and different configurations for the aforementioned values *threshold* and *level*. The tiny x on the curve represents the optimal compromise between precision/recall among all the tested configurations, where $precision = 0.8$ and $recall = 0.8$.

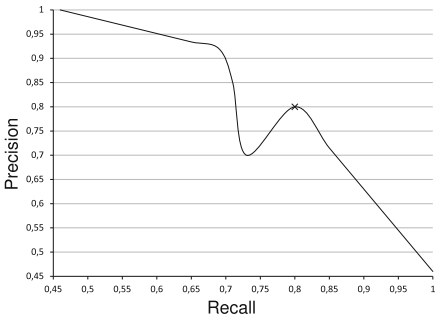

Fig. 4. Precision/Recall curve for the APIs.Guru dataset.

Table 2. APIs.Guru dataset and number of concepts mapped in Schema.org.

Operations	Services	Avg. concepts
2...5	115	1.47
6...10	106	2.56
11...20	120	4.18
21...30	54	6.25
31...40	34	7.79
41...50	23	8.26
	Tot.: 452	Avg.: 3.8

After this profiling and configuration step, we executed the decomposition algorithm with the whole dataset of 452 OpenAPI specifications as input. Table 2 shows the number of operations per service and the average concepts mapped

[12] https://apis.guru/openapi-directory/.

in `Schema.org`. Input APIs were decomposed in 3.8 candidate microservices on average. Although it is not possible to analyze each suggested decomposition individually, this value can be considered close enough to the expected range for this dataset, since the previous step of manual decomposition generated 3.2 microservices per API on average. It could be also interesting to analyze whether the obtained decompositions minimize the number of inter-service calls for sample use cases, but this is outside the scope of this experiment.

This experiment shows that the OpenAPI specifications in the repository are good candidates for decomposition. The original dataset of 452 APIs potentially contains 1735 microservices, which would be cohesive and fine-grained, according to our decomposition approach. This also suggests the applicability/utility of our approach to decompose real-world service APIs, particularly in scenarios where these APIs define a high number of operations, which can then be cumbersome to understand and analyze.

4.4 Possible Limitations

These experiments, and some others not reported here, helped us identify some possible limitations of our solution. In certain cases, we noticed that the input artifacts may be mapped to too few concepts of the shared vocabulary, and thus the decomposition would generate coarse-grained microservices. If it is the case, one should think of: (a) using a domain-specific vocabulary to reduce the ambiguity of terms, (b) fine-tuning parameter `level` to analyze different decompositions, and (c) augmenting obtained results with manual improvements to get a more appropriate decomposition.

Our approach relies on well-defined and described interfaces that provide meaningful names, and follow programming naming conventions such as camel casing and hyphenation. Unfortunately, this is not always the case and some situations are difficult to cope with (e.g., identifiers like `op1`, `param` or `response`). This can be mitigated by the heuristics in the term separation algorithm, and by applying state-of-the-art techniques to improve readability and understandability of interfaces [18].

To conclude, a limitation that is not specific to our approach is the lack of a comprehensive, well-known dataset of microservices to run experiments and replicate/compare the results. Although an industry case study in a large organization is important for validation of a single approach [21], an open-source large dataset of microservices can act as a gold-standard for current and future research in the field. Due to this limitation, we performed our validation upon case studies, example applications, and a large dataset of traditional Web APIs.

5 Related Work

The approach presented in this paper can be seen from a clustering perspective, since candidate microservices are devised by grouping operations according to their shared reference concepts. Clustering techniques have been broadly applied

in the SOA field, for Web Service discovery [22,23] and composition [24]. Traditional flat clustering techniques, such as k-means, are straightforward to apply but their results in the context of traditional Web Services [23] and microservices [20] report a below-average performance. More complex techniques, such as Hierarchical Agglomerative Clustering (HAC, [25]), have proven to be more effective than traditional flat clustering at the cost of lower efficiency but, to the best of our knowledge, these techniques have not been applied to the field of microservices, thus further research in this direction is required to determine their suitability.

Moving to other decomposition approaches for microservices, the Service Cutter tool and framework [20] and the comparison with our approach are already discussed in Sect. 4.1. In the same direction, the work in [21] describes a technique to identify microservices based on dependency graphs among the different tiers of the application (client, server, database). This is a white-box approach, in which interfaces between components in different tiers are analyzed to generate the dependency graph, and then code inspection is performed to devise in detail the boundaries of candidate microservices. The authors claim that the approach is successful since in the case study (a large banking application), candidate microservices were identified and suggested for all subsystems. The authors assume the availability of white-box information (i.e., source code), which is not always the case. Additionally, for complex domains such as banking, it is suggested to start the decomposition gradually and at the edges (where the system is more dynamic and its external interfaces are explicit) [2].

The Enterprise Services Architecture Model Integration (ESAMI) [26] supports the systematic manual integration of microservices by exploiting an ad-hoc architectural reference model [27], and correlation matrices to identify similarities. In contrast, we generalize the idea of reference model, which can be any high-level shared vocabulary or even a domain-specific ontology. We also provide automated support for the identification of microservices.

From the deployment point of view, [28] addresses decomposition in microservices as a suitable means for cloud migration, being the first cloud-native novel architectural style. An industry case study shows applicability scenarios and migration patterns. In this case, the target microservices in the architecture are defined a priori and in a manual way, since the focus is on the deployment of the solution while our approach focuses on its design. Also [29] presents a microservices-based architecture from a deployment point of view. They do not fully migrate the application to microservices at application-level, but preserved the monolithic structure of the application and replicated certain components. This work considers microservices as a way to scale the development process itself rather than the application's functionality, as our solution does.

6 Conclusions and Future Work

This paper proposes a novel approach to support the identification of microservices and the specification of the resulting artifacts both during the initial phases

of the design of a new system and while re-architecting existing applications. The specification artifacts of available operations are mapped onto the entries of a reference vocabulary to highlight their similarities and thus their willingness of being part of different microservices. Then, identified microservices are rendered using OpenAPI, which allows for standardization and fine-grained reuse. Conducted experiments show that our approach found suitable decompositions in some 80% of the cases, while providing early insights about the right granularity and cohesiveness of obtained microservices.

Our future work comprises the addition of non-functional aspects that can affect the decomposition (response time, resource allocation or cost) and the support to "smart" deployment and execution through our deployment framework EcoWare [30].

References

1. Hassan, S., Bahsoon, R.: Microservices and their design trade-offs: a self-adaptive roadmap. In: IEEE International Conference on Services Computing (SCC), pp. 813–818. IEEE (2016)
2. Fowler, M.: Monolith first (2015). http://martinfowler.com/bli-ki/MonolithFirst.html
3. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term (2014). http://martinfowler.com/articles/microservices.html
4. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: migration to a cloud-native architecture. IEEE Softw. **33**(3), 42–52 (2016)
5. Garriga, M.: Towards a microservices taxonomy. In: Microservices: Science and Engineering Workshop, Co-located with Software Engineering and Formal Methods (SEFM), Trento, Italy (2017, accepted for publication)
6. Richards, M.: Microservices vs. service-oriented architecture. (2015)
7. Wilde, N., Gonen, B., El-Sheikh, E., Zimmermann, A.: Approaches to the evolution of SOA systems. In: El-Sheikh, E., Zimmermann, A., Jain, L.C. (eds.) Emerging Trends in the Evolution of Service-Oriented and Enterprise Architectures. ISRL, vol. 111, pp. 5–21. Springer, Cham (2016). doi:10.1007/978-3-319-40564-3_2
8. Richardson, C.: Microservices architecture (2014). http://micro-services.io/
9. George, F.: Challenges in implementing microservices (2015). http://gotocon.com/dl/goto-amsterdam-2015/slides/FredGeorgeChallengesInImplementingMicroServices.pdf
10. Zimmermann, O.: Do microservices pass the same old architecture test? Or: Soa is not dead-long live (micro-) services. In: Microservices Workshop at SATURN Conference, SEI (2015)
11. Browning, T.R.: Applying the design structure matrix to system decomposition and integration problems: a review and new directions. IEEE Trans. Eng. Manag. **48**(3), 292–306 (2001)
12. Kuhn, A., Ducasse, S., Gorba, T.: Semantic clustering: identifying topics in source code. Inf. Softw. Technol. **49**(3), 230–243 (2007). 12th Working Conference on Reverse Engineering
13. Chen, L.: Continuous delivery: overcoming adoption challenges. J. Syst. Softw. **128**, 72–86 (2017)

14. Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. "Big" web services: making the right architectural decision. In: 17th International Conference on World Wide Web, pp. 805–814. ACM Press (2008)
15. OpenAPI Consortium: The OpenAPI Initiative (OAI) (2016). https://www.openapis.org/
16. Kolb, P.: Experiments on the difference between semantic similarity and relatedness. In: Proceedings of the 17th Nordic Conference on Computational Linguistics - NODALIDA 2009. Link University Electronic Press, May 2009
17. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, Reading (2004)
18. Renzis, A.D., Garriga, M., Flores, A., Cechich, A., Mateos, C., Zunino, A.: A domain independent readability metric for web service descriptions. Comput. Stand. Interfaces **50**, 124–141 (2017)
19. Kuhn, H.W.: The Hungarian method for the assignment problem. Naval Res. Logistic Q. **2**, 83–97 (1955)
20. Gysel, M., Kölbener, L., Giersche, W., Zimmermann, O.: Service Cutter: a systematic approach to service decomposition. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I. (eds.) ESOCC 2016. LNCS, vol. 9846, pp. 185–200. Springer, Cham (2016). doi:10.1007/978-3-319-44482-6_12
21. Levcovitz, A., Terra, R., Valente, M.T.: Towards a technique for extracting microservices from monolithic enterprise systems. In: 3rd Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM), pp. 97–104 (2015)
22. Nayak, R., Lee, B.: Web service discovery with additional semantics and clustering. In: IEEE/WIC/ACM International Conference on Web Intelligence, pp. 555–558. IEEE, Silicon Valley (2007)
23. Cong, Z., Fernandez, A., Billhardt, H., Lujak, M.: Service discovery acceleration with hierarchical clustering. Inf. Syst. Front. **17**(4), 799–808 (2015)
24. Alrifai, M., Skoutas, D., Risse, T.: Selecting skyline services for QoS-based web service composition. In: Proceedings of the 19th International Conference on World Wide Web, pp. 11–20. ACM (2010)
25. Murtagh, F., Legendre, P.: Ward's hierarchical agglomerative clustering method: which algorithms implement ward's criterion? J. Classif. **31**(3), 274–295 (2014)
26. Bogner, J., Zimmermann, A.: Towards integrating microservices with adaptable enterprise architecture. In: 2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW), pp. 1–6, September 2016
27. Zimmermann, A., Sandkuhl, K., Pretz, M., Falkenthal, M., Jugel, D., Wissotzki, M.: Towards an integrated service-oriented reference enterprise architecture. In: Proceedings of the 2013 International Workshop on Ecosystem Architectures, pp. 26–30. ACM (2013)
28. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: an experience report. In: Celesti, A., Leitner, P. (eds.) ESOCC Workshops 2015. CCIS, vol. 567, pp. 201–215. Springer, Cham (2016). doi:10.1007/978-3-319-33313-7_15
29. Toffetti, G., Brunner, S., Blöchlinger, M., Spillner, J., Bohnert, T.M.: Self-managing cloud-native applications: design, implementation, and experience. Future Gener. Comput. Syst. **72**, 165–179 (2017)
30. Baresi, L., Guinea, S., Leva, A., Quattrocchi, G.: A discrete-time feedback controller for containerized cloud applications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pp. 217–228. ACM, New York (2016)