Extraction of Microservices from Monolithic Software Architectures

Genc Mazlami, Jürgen Cito, Philipp Leitner
Software Evolution and Architecture Lab
Department of Informatics
University of Zurich
{*firstname.lastname*}@*uzh.ch*

***Abstract*—Driven by developments such as mobile computing, cloud computing infrastructure, DevOps and elastic computing, the microservice architectural style has emerged as a new alternative to the monolithic style for designing large software systems. Monolithic legacy applications in industry undergo a migration to microservice-oriented architectures. A key challenge in this context is the extraction of microservices from existing monolithic code bases. While informal migration patterns and techniques exist, there is a lack of formal models and automated support tools in that area. This paper tackles that challenge by presenting a formal microservice extraction model to allow algorithmic recommendation of microservice candidates in a refactoring and migration scenario. The formal model is implemented in a web-based prototype. A performance evaluation demonstrates that the presented approach provides adequate performance. The recommendation quality is evaluated quantitatively by custom microservice-specific metrics. The results show that the produced microservice candidates lower the average development team size down to half of the original size or lower. Furthermore, the size of recommended microservice conforms with microservice sizing reported by empirical surveys and the domain-specific redundancy among different microservices is kept at a low rate.**

***Keywords*-microservices; extraction; coupling; graph-based clustering;**

I. Introduction

In recent years, the software engineering community has seen a tendency towards cloud computing [1]. The changing infrastructural circumstances pose a demand for architectural styles that leverage the opportunities given by cloud infrastructure and tackle the challenges of building cloud-native applications. An architectural style that has drawn a substantial amount of attention in the industry in this context – as for instance in [2], [3] – is the *microservices* architecture.

Microservices come with several benefits such as the fact that services are independently developed and independently deployable, enabling more flexible horizontal scaling in IaaS environments and more efficient team structures among developers. It is therefore no surprise that big internet industry players like Google and eBay [4], Netflix [5] and many others have undertaken serious efforts for moving from initially monolithic architectures to microservice-oriented application landscapes. The common problem in these efforts is that identifying components of monolithic applications that can be turned into cohesive, standalone services is a tedious manual effort that encompasses the analysis of many dimensions of software architecture views and often heavily relies on the experience and know-how of the expert performing the extraction.

In this paper, we aim to tackle the above-mentioned extraction problem algorithmically. To that end, we present three formal coupling strategies and embed those in a graph-based clustering algorithm. The coupling strategies rely on (meta-) information from monolithic code bases to construct a graph representations of the monoliths that are in turn processed by the clustering algorithm to generate recommendations for potential microservice candidates in a refactoring scenario. Furthermore, a prototype implementation of the approach is evaluated.

The rest of this paper is structured as follows: Section II gives an overview over related work. Section III formally defines the extraction model, the coupling strategies and the clustering algorithm. In section IV, an evaluation with respect to performance and quality is performed. Finally, section VI concludes the work and discusses potential limitations and future work.

II. Related Work

Decomposing software systems has been an important branch in the software engineering research discipline – extracting microservices from monoliths is the next evolutionary form of the original challenge of decomposing systems. Parnas et al. were among the first to systematically investigate the criteria that should be used when decomposing systems into modules [6], focusing on decomposition based on the principle of *information hiding*. Komondoor et al. present an approach to extract independent online services from legacy enterprise business applications. The authors note that the resulting challenge of migrating monolithic legacy applications is a labor-intensive and manual task, and that the tool support and automation in that area is not satisfactory. They present a static analysis-based approach that is tailored towards the imperative nature of batch processing programs in legacy languages such as COBOL and uses backwards data flow slicing to extract highly cohesive services.

Microservices can be seen as the current reincarnation of the idea of service-based computing. However, work particularly around *micro*services is limited. Pahl and Jamshidi conducted a systematic secondary study and reviewed and classified the existing research body on microservices [7].

978-1-5386-0752-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICWS.2017.61

The authors detected a specific lack of tool-support for microservices in the current state of the art. A good part of the efforts in microservices research is only conceptual and the review reveals a higher number of use case studies than technology solutions and tool support studies [7]. Schermann et al. investigate industrial practices in services computing [8], by empirically studying service computing practices in 42 companies of different sizes, with special attention given to the microservices trend. Among other aspects, service size and complexity were reviewed. The results imply that services vary in size, but extreme values such as very small services with under 100 LOC or very large services with more than 10000 LOC are rarely encountered in practice. Another interesting result is the observation that services are dedicated. In other words, the respondents reported that the services they are involved in typically are dedicated to relatively narrow and concise tasks [8]. Extracting microservices from monoliths has been recently recognized as a need [9]. The authors present a systematic, yet manual, approach to identify microservices in large monolithic enterprise systems by constructing a dependency graph between client and server components, database tables and business areas. A notable attempt at providing a structured way of identifying microservices in monolithic code bases is ServiceCutter [10]. The authors present a tool that supports structured service decomposition through graph cutting. The internal representation of the system to be decomposed is based on a catalog of 16 different coupling criteria that were abstracted from literature and industry know-how. Software engineering artifacts and documents such as domain models and use cases act as an input to generate the coupling values that build the graph representation. However, *ServiceCutter* has no means of mining or constructing the necessary structure information from the monolith itself and hence must rely on the user to provide the software artifacts in a specific expected model.

The research gap that this paper attempts to close is mainly twofold. On one side, there is a large body of prior work on traditional software decomposition and modularization, including techniques such as traditional software metrics, repository mining or static analysis. Among all those solutions, there are none that are specifically designed for an application in a microservice extraction scenario. On the other hand, there exists a small and recent body of work on the topics of microservice migration, extraction of microservices from monoliths. Among these attempts at tackling the microservice extraction problem, we are the first to provide semi-automated or formal approach that covers recommendation of microservices without heavy user input. The approach presented in this paper provide the best of both worlds traditional decomposition techniques on one side and microservice extraction approaches and design principles on the other side to close the illustrated research gap.

III. A Microservice Extraction Model

All of the extraction strategies outlined in this paper are embedded into a predefined extraction model. It comprises of three extraction stages: the *monolith* stage, the *graph* stage and the *microservices* stage. There are two transformations between the stages: The *construction* step transforms the monolith into the graph representation, and the *clustering* step decomposes the graph representation of the monolith into microservice candidates. The transformations performed during the construction step differ according to the extraction strategy in use.

The starting point is always an actual code base or repository of an implemented application in some form of version control system (e.g., Git[1]). The aspects of a code base important to the extraction process are captured by the *monolith* representation. A monolith M is a triple $M = (C_M, H_M, D_M)$, where C_M is the set of class files, H_M is the change history and D_M is the set of developers that contributed code to the monolith M. Each class file $c_k \in C_M$ has a file *name*, a file *path* which is assumed to be unique in the monolith M and file *contents* in the form of text.

The change history H_M of a monolith M is defined as an ordered sequence of *change events*: $H_M = (h_1, h_2, \dots, h_n)$. Each change event h_i is defined as a triple $h_i = (E_i, t_i, d_i)$, where $E_i = \{c_1, c_2, \dots, c_n | c_k \in C_M\}$ is a set of classes from the set C_M that were added or modified by the change event h_i, t_i is the *timestamp* at which the change event occurred, and $d_i \in D_M$ is the developer that commited the corresponding change.

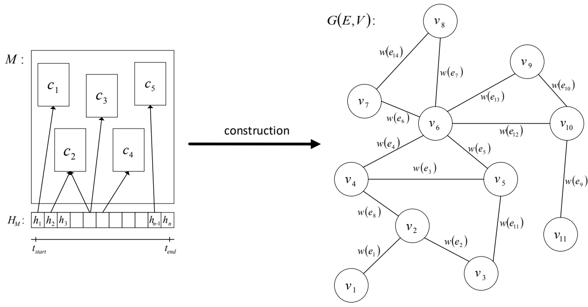

Figure 1. Construction Step

The *monolith* is transformed into the *graph representation* by the *construction* step, as illustrated in Figure 1. The construction step employs one of the *coupling strategies* to mine the information in the monolith and create an undirected, edge-weighted graph G. In the graph $G = (E, V)$, the vertices $v_i \in V$ each correspond to a class $c_i \in C$ from the monolith. Each graph edge $e_k \in E$ has a weight defined by the *weight function*. The weight function determines how strong the coupling between the neighboring edges is

[1] https://git-scm.com/

according to the coupling strategy in use. A higher weight value indicates stronger coupling. Note that not all classes $c_k \in C$ from the original monolith will have an edge with their corresponding vertex $v_i \in V$. There may be classes that do not exhibit any coupling to any other class when using the coupling strategies, which would lead to the class being discarded from further processing in the graph.

The graph representation is then cut into pieces to obtain the candidates for microservices using the graph clustering algorithm described below.

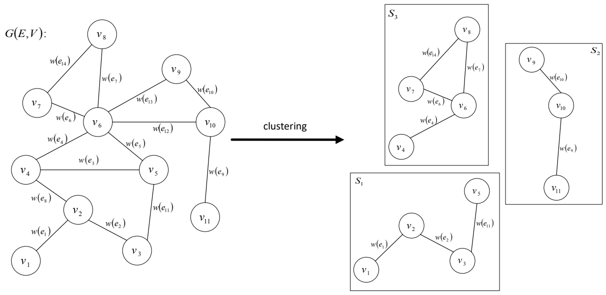

Figure 2. Clustering Step

The clustering results in the *microservice* recommendation R_M for the monolith M. Formally, a microservice recommendation is a *forest* R_M consisting of at least 1 *connected component* S_j. A connected component S_j in the forest is referred to as a microservice, while N denotes the number of microservices in the recommendation D_M.

Each microservice S_j is a connected, directed graph $S_j = (C_{S_j}, R_{S_j})$, where the vertices of the graph consists of the set C_{S_j} of classes in the microservice S_j. The set C_{S_j} is a proper subset of the set of classes in the monolith.

A. Extraction Strategies

1) Logical Coupling Strategy: The *Single Responsibility Principle* [11] states that a software component should have only one reason to change. By consequence, software design that follows the principle should gather together the software elements that change for the same reason. Furthermore, one of the main benefits and design goals behind the concepts of microservices is to enforce strong module boundaries [2]. Strong module boundaries and adherence to the Single Responsibility Principle provide benefits in case of a change: If developers have to make changes to a system, they only need to locate the module to be changed and only need to understand that confined module. What both of the above-mentioned arguments have in common is the fact that they are founded on the *change* of software elements such as class files. Hence, to obtain microservices that provide the mentioned benefits, it is essential to analyze the changing behavior of the original monolith. Class files that change together should consequently also belong to the same microservice. This constitutes the rationale behind the *logical coupling strategy*.

Gall et al. coined the term *logical coupling* as a retrospective measure of implicit coupling based on the revision history of an application source code [12]. For each change event $h_i = (E_i, t_i, d_i)$ in the change history H_M of a monolith M, the set of changed classes E_i contains all class files that were changed *during* the predefined session starting at t_i.

Let the classes $c_1, c_2 \in E_i$ be two distinct classes that were changed in that respective change event. And let δ be function, that indicates whether these classes c_1, c_2 have changed together in a certain commit $h_i \in H_M$:

$$\delta_{h_i}(c_1, c_2) = \begin{cases} 1 & \text{if } c_1, c_2 \text{ changed in } h_i \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Now let Δ be the *aggregated logical coupling*:

$$\Delta(c_1, c_2) = \sum_{h \in H_M} \delta_h(c_1, c_2) \tag{2}$$

In the *construction* step, the strategy employs the aggregated logical coupling Δ to compute the weights on the edges of the graph G. Let $e_l = (c_i, c_k)$ be an edge on the graph G representing the original monolith. Then the weight $w(e_l)$ is computed by $w(e_l) = \Delta(c_i, c_k)$. The graph G is constructed by computing edge weights as shown for all pairwise distinct class files $c_k \in C$ of the monolith $M := (C_M, H_M, D_M)$.

2) Semantic Coupling Strategy: In microservices literature, the notion of bounded context originating in domain-driven design is presented as a promising design rationale for microservices and their boundaries [13], [2]. According to that rationale, each microservice should correspond to one single defined bounded context from the problem domain. This results in scalable and maintainable microservices that focus on one responsibility. Hence, it is desirable to infer an extraction strategy from said concept. One possibility from prior work that enables to formally identify entities from the problem domain is by examining the contents and semantics of source code files through information retrieval techniques [14], [15].

Basically, the strategy couples together classes that contain code about the same ”things”, i.e., domain model entities. As prior work [14] has shown, identifiers and expressions in code (variable names, method names etc.) can be used to identify high level topics or domain concepts in source code. The semantic coupling uses these expressions as input with the term-frequency inverse-document-frequency method (tf-idf) [16]. It computes a scalar vector for a document given a predefined set of words. By computing these vectors for class files in the monolith and then computing the cosine similarity between vectors of pairwise distinct classes, the semantic coupling strategy can compute a score that indicates how related two files are in terms of domain concepts or ”things” expressed in code and identifiers.

The semantic coupling strategy looks at all pairwise distinct classes $c_i, c_j \in C_M$, where $i \neq j$, in a monolith M. For each pair of classes c_i, c_j the procedure described below is performed.

First each of the class files is tokenized, which results in a set of words $W_j = \{w_1, w_2, \ldots, w_n\}$ for each class c_j. During the tokenization process, all special characters and symbols specific to the programming languages used in the classes will be filtered out. The set W will now contain all identifiers from the code of the respective class. There are words or identifiers in class files that are not related to any actual domain concept or entity of the application, but are identifiers that belong to the programming language or framework in use. These stop words are filtered out of the set W_j, W_i for both of the class files c_j, c_i. During the term extraction, the *term list* $T = \{t_1, t_2, \ldots, t_k\}$ is constructed by combining all the words $w_l \in W_j$ from the first class with all the words $w_l \in W_i$ in the second class. This term list serves as the basis for the computation of the tf-idf vectors for each class. The procedure continues to compute a vector $V \in \mathbb{R}^n$ for the word list W_j for the class c_j and analogously a vector $X \in \mathbb{R}^n$ for the class c_i and its word list W_i. The dimension n of the vectors V and X is equal to the size of the term list T computed in the previous step. The k-th element of the vectors is formed by computing the tf-idf (term-frequency inverse-document-frequency) value [16] of the k-th term in the term list T with respect to the corresponding word list. The elements of X are therefore computed by $\forall\ t_k \in T:\ \ x_k = tf(t_k, W_i) \cdot idf(t_k, W_{all})$ while V is computed analogously by replacing W_i with W_j. The variables v_k and x_k in the above equation denotes the k-th element of vectors V and X respectively. W_{all} is the set of all word lists of all class files in the current computation – in this example $W_{all} = \{W_i, W_j\}$.

The *raw term frequency* $f(t_k, W_i)$ of a term $t_k \in T$ with respect to a document or list of words W_i is defined as the number of times that the term t_k is found in W_i [17]. To avoid corruption of the result caused by unusually high raw frequency of a term in a specific document, the *logarithmic term frequency* [17] is used as a term frequency measure for the semantic coupling strategy. It is defined as:

$$tf(t_k, W_i) = \begin{cases} 0 & \text{if } f(t_k, W_i) = 0 \\ 1 + log_e(f(t_k, W_i)) & \text{otherwise} \end{cases} \tag{3}$$

The *inverse document frequency* measures how rare or common a given term t_k is across a set of documents or word lists $W_{all} = \{W_1, W_2, \ldots\}$. Assume that n is defined as the number of documents a term t occurs in: $n(t) = |\{W_i \in W_{all} : t \in W_i\}|$. The inverse document frequency is then defined as:

$$idf(t_k, W_{all}) = log\left(\frac{|W_{all}|}{n(t_k)}\right) \tag{4}$$

After the vector computation using the tf and idf measures, there are two vectors X and V, one for each word list of the respective class. By using the cosine between the two vectors, the similarity of the two original classes with respect to the semantics of their identifiers and contents can be computed. Let $e_k \in E$ be the edge between the class files c_i and c_j in the graph G representing the original monolith M. Also, let X and V denote the resulting $tf - idf$-vectors for the classes c_i and c_j. The weight $w(e_k)$ of the edge is defined by the cosine between the two vectors: $w(e_k) = cos(X, V)$.

3) Contributor Coupling Strategy: Prior work on reverse engineering has successfully employed team and organization information to recover relationships among software artifcats that stay undiscovered otherwise [18]. It is possible to recover an ownership architecture from version control systems [18]. Such ownership architectures reveal team structures and communication patterns between teams of different components of the software. Well-organized teams are a main concern when migrating a project to microservices. The microservice paradigm proposes cross-functional teams of developers organized around domain and business capabilities. One of the main objectives of the team and organization philosophy in the microservice paradigm is to reduce communication overhead to external teams and maximize internal internal communication and cohesion inside developer teams of the same service. The contributor coupling strategy aims to incorporate these team-based factors into a formal procedure that can be used to cluster class files class files according to the above-mentioned viewpoints. It does so by analyzing the authors of changes on the class files in version control history of the monolith.

The presented procedure for computing the contributor coupling is applied to all class files $c_i \in C_M$ in the monolith M. The first step involves finding all history change events $h_k \in H_M$ that have modified the current class c_i. Let $\gamma(h_k)$ denote a function that returns the set of changed class files E_k from the change event $h_k = (E_k, t_k, d_k)$. Then the set of change events where class c_i was involved is denoted by $H(c_i) = \{h_k \in H_M | c_i \in \gamma(h_k)\}$. Let $\sigma(h_k)$ denote a function that returns the uniquely identifiable author $d_k \in D_M$ that contributed the change event h_k to the monolith M. Then, the set of all developers that contributed to the current class file c_i are denoted by $D(c_i) = \{d_x \in D_M \mid \forall\ h_k \in H(c_i) : d_x = \sigma(h_k)\}$. After computing the set $D(c_i)$ for all classes $c_i \in C_M$ in the monolith M, the coupling can be computed. In the graph G representing the original monolith M, the weight on any edge e_l is equal to the contributor coupling between two classes c_i and c_j which are connected by e_l in the graph. The weight is defined as the cardinality of the intersection of the sets of developers that contributed to class c_i and c_j: $w(e_l) = |D(c_i) \cap D(c_j)|$.

B. Clustering Algorithm

The next step in the extraction process is to cut the graph G in (connected) components that will represent the recommended microservice candidates. Formally, this means deleting edges $e_k \in E$ from the graph in such a manner that the remaining connected components converge The partitioning of the graph must take the weights $w(e)$ on the edges into account when deciding on which edges to delete. In the presented extraction model, a large weight $w(e_k)$ on an edge e_k between two class vertices c_i and c_j implies that the classes c_i and c_j belong to the same service candidate according to the utilized strategies. Consequently, the algorithm should primarily favor edges with low weights for deletion. To ensure that the deletion of a single edge always guarantees a partition, not the entire graph is considered during extraction, but only the minimum spanning tree (MST). Every time an edge in a MST is deleted, it causes the MST to partition into two connected components. It is not possible to automatically determine when to stop partitioning the graph. Thus, the algorithm takes the number of targeted partition steps as an input parameter.

As presented in Algorithm listing 1, the first step is the weight inversion of the edges. The goal is for class nodes that have a high weight on the edges between them to remain in the same microservice candidates or connected components. By directly computing the minimum spanning tree, this reasoning would be defeated. Hence, the weights $w(e_k)$ on the edges $e_k \in E$ in the graph $G = (V, E)$ have to be inverted with $w(e_k) = \frac{1}{w(e_k)}$. The minimum spanning tree $MST(G)$ of the graph G will hence preserve the most important edges according to the computed couplings or weights. The MST edge set $edges_{MST}$ is computed by the KRUSKAL algorithm [19]. The set $edges_{MST}$ is sorted according to their inverted weight, and then reversed such that edges with the highest inverted weight – and thus the lowest coupling – occur first in the list.

Algorithm 1 MST Clustering Algorithm

```
 1: function CLUSTER($edges, n_{part}, s$)
 2:     for $e \in edges$ do
 3:         $e.weight \leftarrow \frac{1}{e.weight}$
 4:     end for
 5:     $edges_{MST} \leftarrow$ KRUSKAL($edges$)
 6:     $edges_{MST} \leftarrow$ SORT($edges_{MST}$)
 7:     $edges_{MST} \leftarrow$ REVERSE($edges_{MST}$)
 8:     $n \leftarrow 1$
 9:     while $n <= n_{part}$ do
10:         $edges_{MST}[0].delete()$
11:         $n \leftarrow$ DFS($edges_{MST}$)
12:     end while
13:     $components \leftarrow$ REDUCECLUSTERS($edges_{MST}, s$)
14:     return $components$
15: end function
```

The algorithm then enters the iterative edge deletion loop, as shown on the lines 9 - 12 in algorithm listing 1. The number of partitions that the algorithm attempts is defined by the external parameter n_{part}. In each of the iteration steps, the edge with the lowest coupling is deleted from the MST and the remaining connected components are computed by traversing the remaining edges in Depth-First-Search (DFS). After n_{part} partitions have been performed, the remaining edges are passed to the *ReduceClusters* procedure which ensures that there are no unusually large clusters of class nodes. This is necessary due to the observation that there are class files in monoliths that exhibit extraordinary high degrees of coupling because of their special role in the frameworks and languages used. We treat such files as outliers and handle them in *ReduceClusters*: All connected components whose number of contained class nodes is larger than a predefined parameter s are divided further by deleting the class node with the highest degree. This reduces the size of the clusters while keeping internal coupling inside the remaining connected components high.

IV. Evaluation

We implemented the presented extraction model and strategies as a proof-of-concept in an open source Java project[2]. We also wrote a web-based frontend in AngularJS[3]. We evaluate our approach with respect to two research questions:

RQ1: What is the performance of the implemented prototype with respect to execution time?

RQ2: What is the quality of the microservice recommendations generated by the prototype?

To that end, an evaluation is designed with a specific set of sample code bases from open-source projects and a series of performance measurement experiments is conducted for **RQ1**, while **RQ2** is answered through a series of quality measurements using custom metrics. The sample set of monolithic code bases consists of open source repositories using the Git VCS. The sample projects are web applications written in Java, Python or Ruby and use ORM-technology for data management. Since ORM systems use classes to represent database tables, such applications lend themselves optimally for a class-based extraction model such as in this paper. The sample repositories have from 200 to 25000 commits in their version history, while the size of the projects in LOC is in the range of 1000 LOC to 500000 LOC. Furthermore, all sample projects have from 5 to 200 authors that contributed changes in the version history.

A. Performance

Table I lists the execution times (in seconds) of the sample projects for the different coupling strategies in relation to the repository properties commit count (h), contributor count (c) and code size in LOC (s).

Semantic Coupling: The execution times measured on the sample projects confirm the intuition that the performance of the semantic coupling strategy is mainly and directly impacted by the total size of the code in the repository in LOC. This comes at no surprise since the outlined tf-idf

[2] https://github.com/gmazlami/microserviceExtraction-backend

[3] https://github.com/gmazlami/microserviceExtraction-frontend

Table I
EXECUTION TIMES FOR LOGICAL COUPLING (LC), CONTRIBUTOR COUPLING (CC) AND SEMANTIC COUPLING (SC)

Project	**h**	**c**	**s**	**LC**	**CC**	**SC**
DemoSite	1477	22	1301	91	3	1729
mayocat	1670	4	33216	136	173	149166
TNTConc.	216	15	118356	80	172	281075
petclinic	545	32	1564	53	5	545
sunrise-java	1859	11	18820	62	129	69646
helpy	1650	42	9230	71	19	17127
Spina	500	38	2468	7	4	1765
sharetribe	14940	46	53845	34	134	540303
Hours	796	21	4268	11	3	5567
rstat.us	2260	64	6807	51	4	5326
kandan	868	52	1553	81	5	1375
fulcrum	697	44	3085	6	7	2217
redmine	13009	6	87529	200	1136	643179
chiliproject	5532	39	65171	319	1004	562731
django-fiber	848	20	4686	98	7	3710
mezzanine	4964	238	11780	302	20	18845
wagtail	6809	205	48971	269	15	227307
mayan-edms	5049	25	39225	270	76	199418
django-shop	3451	62	8281	58	1	15841
django-oscar	6881	170	32272	540	5	149249
django-wiki	1589	64	8113	246	13	7752

similarity computation needs to process the contents of the class files line by line.

Logical Coupling: The execution time rises with the number of commits. Nevertheless, there are some large variations of the execution times with spikes towards higher execution time values. Examples are the *LC* execution times for the *django-oscar* and *chiliproject* samples. The computation of the logical couplings involves a power set of all class files in the change set of a certain change interval in the history H_M of the analyzed monolith M. In those cases, the average size of this change set dominates the history size and hends leads to spikes.

Contributor Coupling: While the impact of the history size can be observed in the Table I, there are extreme outliers attributed to sample projects where the number of class files $|C|$ is unusually high and therefore dominates the effect of the history size and leads to higher execution times. This is due to the fact that computing the contributor coupling involves both traversing the entire history but also iterating through pair-wise permutations of the classes in the monolith. Therefore, outliers are explained by cases where the number of classes dominates the growth of execution time.

Generally, all the performance experiments show satisfying performance levels. They indicate that our approach can be used for different scenarios (e.g., recommendation systems for software architects or in continuous integration).

B. Quality

While there are clearly established and well-known quality metrics in fields such as object-oriented design [20], microservice research exhibits a scarce amount of such efforts. Therefore, we use custom metrics as proxies to capture and compare recommendation quality. We present the evaluation of the *team size reduction ratio* metric (tsr) and the *average domain redundancy* metric (adr). A more detailed overview over the rationale behind the metrics and the results can be found in [21].

1) Team Size Reduction: One of the factors that is often cited as a main benefit of microservices is the improved team structure and reduced team complexity and size [13]. Reduced team size translates to a reduced communication overhead and thus more productivity and focus of the team can be directed towards the actual domain problem and service it is responsible for. We use the *tsr* metric as a proxy for this team-oriented quality aspect. Let R_M be a microservice recommendation for a monolith M. The *tsr* is computed as the average team size across all microservice candidates in R_M divided by the team size of the original monolith M.

Figure 3 shows the corresponding results for the different coupling strategies (LC, CC, SC) or combinations thereof for all 21 projects in the study. A tsr value of 1 would imply that the new team size per microservice is as large as the original team size for the monolith, while a lower value would mean a reduction of the team size.

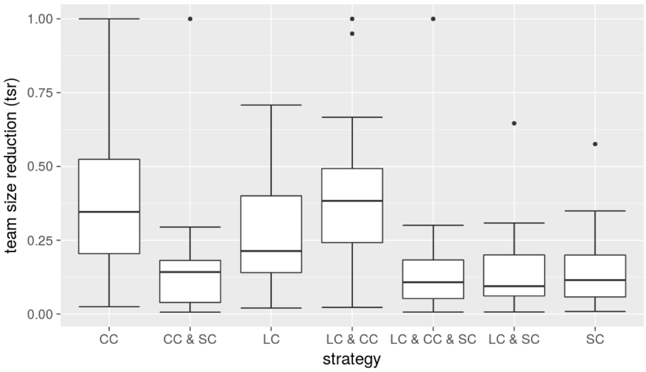

Figure 3. Boxplot of the team size reduction ratio (tsr) results

Figure 3 shows that all of the experiment series exhibit median values below 0.5, independent of the strategy combination used. All but one of the combinations also have the upper quartile below a tsr of 0.5. These observations indicate that the extraction approach and strategies perform very well with respect to team size improvement; the team size shrinks down half of the original size or even lower in the majority of the experiments. Taking a look at the discrepancies between the results of the different basic extraction strategies, it is clear that the semantic coupling strategy shows the best results in this metric, by significantly outperforming both the logical and contributor coupling strategies. We performed Welch t-tests to support this claim: The test comparing the LC and SC distributions indicates significant differences (p-value $<$ 0.05), while the comparison of the distributions for the CC and SC experiments shows an even more significant difference (p-value < 0.01). The strategy combinations where the *SC* strategy is involved appear to consistently yield better tsr than the rest. Our t-tests show for instance that the distributions for the *LC & CC* and the *LC & CC & SC* are significantly different (p-value < 0.01). Similar differences are implied by the p-values of the tests involving the other combinations of the *SC* strategy (all t-test results can be found in [21]).

An observation that stands out is the higher upper quartile value and the significantly higher upper whisker for the *CC* strategy. Upon detailed inspection of the experiment results, it becomes apparent that there are certain sample projects where the tsr is equal to 1, meaning that there has been no reduction whatsoever. This occurs only in experiments where the contributor coupling strategy is involved. The explanation for this is found by looking in the coupling graph before the MST clustering algorithm is applied. In all of the cases where the tsr resulted in a value of 1, the coupling graph consisted of only one large connected component with neighboring nodes arranged in a star shape. The high-degree node in the center corresponds to a class file that has been changed by every single one of the contributors that participated in the history of the monolith. Following the definition of the contributor coupling, this file will be coupled to every other class file in the monolith. The weights of the edges towards the outside of the star are weaker than the ones near to the center. This causes a degenerate behavior of the clustering algorithm where despite deleting the lowest-weight edge in every step, the number of components does not increase but stays at 1 connected component. Only the outer nodes are iteratively cut away from the star shape. At the end, the remaining graph will still always contain the central class node and it will be the only remaining connected component and hence the only remaining microservice candidate. This remaining candidate will by definition have the same team size as the original monolith since the central node exhibits all of the contributors as the original monolith.

2) Average Domain Redundancy: The problem domain of an application can be viewed as a set of bounded contexts, with each of the bounded contexts having clear responsibilities and clearly defined interface that defines which model entities are to be shared with other bounded contexts [13]. Also, a favorable microservice design avoids duplication of responsibilities across services. As Schermann et al. report in their empirical study [8], microservices are dedicated to relatively narrow and concise tasks. Thus, we compute the *average domain redundancy* metric as a proxy to indicate the amount of domain-specific duplication or redundancy between the services on a normalized scale between 0 and 1, where we favor service recommendations with lower adr values.

Figure 4 indicates that 6 out of the 7 strategy combinations deliver good domain redundancy distributions. The median of 4 out of those 6 experiment series is significantly lower than 0.25, indicating that considerably less than a quarter of the domain content in the microservice source code is redundant between the recommended services. Furthermore, one might intuitively assume that the semantic coupling strategy will perform very well on this metric – and rightfully so – because it optimizes the extraction graph for domain-specific clustering. A look at the results in Figure 4 supports this intuition: The adr values produced by the *SC* strategy in isolation have the lowest median. The t-test results partially confirm this observation (p-value < 0.05 for 3 out of 5 comparisons involving the isolated *SC* experiments). The two cases where the p-value is above 0.05 involve the *SC* strategy in both comparison data sets, once isolated and once in combination with other strategies. This explains the less significant difference in these t-tests.

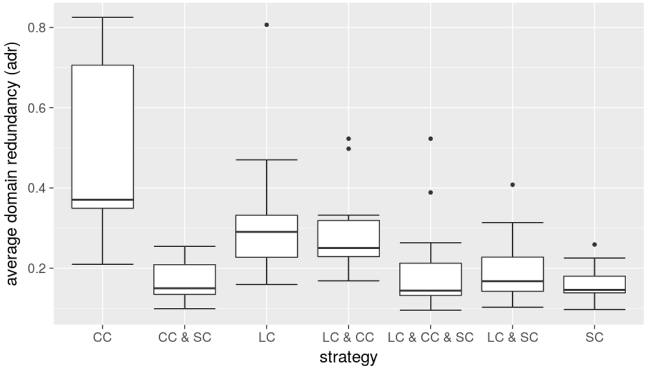

Figure 4. Boxplot of the average domain redundancy (adr) results

Despite the median of the adr for the CC experiments being well below the hard threshold of 0.5, the upper quartile is extremely high and the results show generally a stronger skew to higher values than for the other strategies. This phenomenon can be attributed to the fact that the skill distribution among developers in open-source projects is not domain-oriented but rather technology-dependent. This means, front-end developers tend to contribute to the same files independent of the domain content of those files. The same holds analogously for back-end or database developers. Due to the way the contributor coupling is computed, this situation leads to class files being coupled into the same microservice despite having low semantic and domain-specific cohesiveness. Consequently, this leads to higher domain redundancy among different service candidates.

V. Limitations and Future Work

One limitation is the fact that the extraction model is based on classes as the atomic unit of computation in the strategies and the graph. While this premise lends itself nicely to our graph-based extraction model, it limits the available leeway

when refactoring monoliths. Using methods, procedures or functions as atomic units of extraction might potentially improve the granularity and precision of the code rearrangement and reorganization and will hence be considered in future work.
Our extraction model circumvents the database decomposition challenge by assuming the presence of an ORM system that represents data model entities as classes that are treated just like any ordinary class by the extraction algorithm. Of course, microservices in practice will often lack such a component, and hence the unsolved problem of how to share or assign pre-existing databases to different services remains a limitation of this paper and a challenge for future work in the area.

VI. Conclusion

We introduced a formal model that enables extracting microservices from monolithic software architectures. We designed and implemented extraction strategies on top of that model and evaluated performance and quality of our approach based on 21 open source projects in Java, Ruby, and Python. The performance evaluation shows that, for the most part, our approach scales with respect to the size of the revision history (logical- and contributor coupling). The quality evaluation shows that our approach can reduce the microservice team size to a quarter of the monolith's team size or even lower.

Most strategy combinations perform very well with respect to lowering *average domain redundancy*. The median values for domain redundancy are consistently at 0.3 or even lower for most cases. Only recommendations generated with the contributor coupling alone without any other strategy show more widely distributed results that are skewed towards higher redundancy.

The quality evaluation can guide software architects to use our approach accordingly based on their needs (i.e., reducing team size and lower domain redundancy of extracted services). For the future, we are considering more fine-granular software artifacts (e.g., methods) as an input to our approach that might potentially improve the granularity and precision of the code rearrangement and reorganization.

VII. Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802 (CloudWave).

References

[1] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications: An empirical study on software development for the cloud," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, 2015.

[2] M. Fowler, "Microservices: a definition of this new architectural term," http://martinfowler.com/articles/microservices.html, 2014, accessed: 2016-08-16.

[3] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[4] T. Hoff, "Deep lessons from google and ebay on building ecosystems of microservices," http://highscalability.com/blog/2015/12/1/deep-lessons-from-google-and-ebay-on-building-ecosystems-of.html, 2015, accessed: 2016-08-16.

[5] T. Mauro, "Adopting microservices at netflix: Lessons for architectural design," https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, 2015, accessed: 2016-08-16.

[6] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972. [Online]. Available: http://doi.acm.org/10.1145/361598.361623

[7] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, 2016.

[8] G. Schermann, J. Cito, and P. Leitner, "All the services large and micro: Revisiting industrial practice in services computing," in *International Conference on Service-Oriented Computing*. Springer, 2015, pp. 36–47.

[9] A. Levcovitz, R. Terra, and M. T. Valente, "Towards a technique for extracting microservices from monolithic enterprise systems," *arXiv preprint arXiv:1605.03175*, 2016.

[10] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2016, pp. 185–200.

[11] R. C. Martin, "The single responsibility principle," *The Principles, Patterns, and Practices of Agile Software Development*, pp. 149–154, 2002.

[12] H. Gall, M. Jazayeri, and J. Krajewski, "Cvs release history data for detecting logical couplings," in *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*. IEEE, 2003, pp. 13–23.

[13] S. Newman, *Building Microservices*. " O'Reilly Media, Inc.", 2015.

[14] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*. IEEE, 2001, pp. 107–114.

[15] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems." in *ICSM*, vol. 6, 2006, pp. 469–478.

[16] J. Ramos, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, 2003.

[17] M. Lan, C. L. Tan, J. Su, and Y. Lu, "Supervised and traditional term weighting methods for automatic text categorization," *IEEE transactions on pattern analysis and machine intelligence*, vol. 31, no. 4, pp. 721–735, 2009.

[18] I. T. Bowman and R. C. Holt, "Software architecture recovery using conway's law," in *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1998, p. 6.

[19] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.

[20] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[21] G. Mazlami, "Algorithmic extraction of microservices from monolithic code bases," Master Thesis,Software Evolution and Architecture Lab, Department of Informatics, University of Zurich, 2017.