毕业论文（设计）外文翻译

外文翻译之一

Extraction of Microservices from Monolithic Software Architectures

作者： Genc Mazlami; Jürgen Cito; Philipp Leitner

国籍： Switzerland

出处： 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 2017, pp. 524-531.

原文正文：

2017 IEEE 24th International Conference on Web Services

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*: $H_M = (h_1, h_2, \ldots, h_n)$. Each change event h_i is defined as a triple $h_i = (E_i, t_i, d_i)$, where $E_i = \{c_1, c_2, \ldots, c_n | c_k \in C_M\}$ 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.

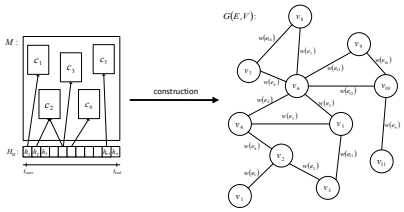

Figure 1. Construction Step

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.

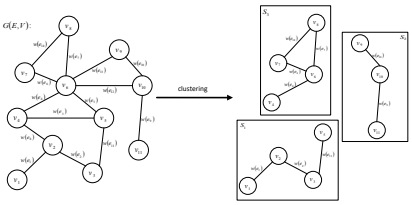

Figure 2. Clustering Step

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 $h_i \in H_M$:

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

Now let Δ be the *aggregated logical coupling*:

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

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 ∈ edges do
3:          e.weight ← 1 / e.weight
4:      end for
5:      edges_MST ← KRUSKAL(edges)
6:      edges_MST ← SORT(edges_MST)
7:      edges_MST ← REVERSE(edges_MST)
8:      n ← 1
9:      while n <= n_part do
10:         edges_MST[0].delete()
11:         n ← DFS(edges_MST)
12:     end while
13:     components ← 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.

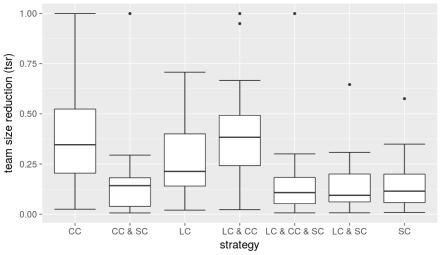

Figure 3. Boxplot of the team size reduction ratio (tsr) results

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.

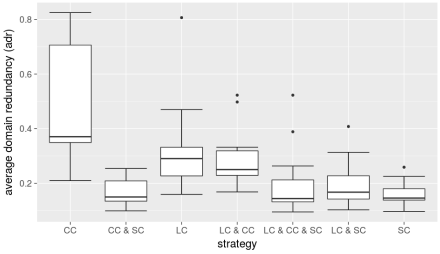

Figure 4. Boxplot of the average domain redundancy (adr) results

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.

从一体式软件架构中提取微服务

作者： Genc Mazlami; Jürgen Cito; Philipp Leitner
国籍： 瑞士

出处： 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 2017, pp. 524-531.

中文译文：

摘要——在移动计算、云计算基础设施、DevOps 和弹性计算等领域发展的推动下，微服务架构风格已经成为设计大型软件系统的单体架构风格的替代选择。工业中的单体架构时代遗留的应用程序正在向面向微服务的体系结构迁移。在这种情况下，关键的挑战是从现有的单体代码库中提取微服务。虽然存在非形式化的迁移模式和技术，但在这方面缺乏形式化的模型和自动化的支持工具。本文通过提出一个形式化的微服务抽取模型来解决这一问题，该模型允许在重构和迁移场景中对微服务集进行算法推荐。形式化模型是在一个基于 web 的原型中实现的。性能评估表明，该方法具有相当不错的性能。通过定制的微服务度量标准对推荐质量进行了定量评价。结果表明，产生的微服务候集选将平均开发团队规模降低到原始规模的一半或更低。此外，推荐的微服务规模符合实证调查报告的微服务规模，不同微服务之间的特定领域冗余保持在较低的水平。

索引词——微服务；提取；耦合；基于图的聚类；

1 引言

近年来，软件工程界出现了云计算的趋势[1]。不断变化的基础设施环境对架构风格提出了需求，这些架构风格利用云基础设施带来的机遇，并应对构建云本地应用程序的挑战。在这种情况下，一种在业界引起了大量关注的体系结构风格——例如在[2]、[3]中——就是微服务体系结构。

微服务带来了一些好处，例如服务是独立开发和可独立部署的，从而在 IaaS 环境中实现了更灵活的水平扩展，并在开发人员中实现了更高效的团队结构。因此，谷歌（Google）和 eBay[4]、Netflix[5]以及其他许多互联网行业的大公司都做出了认真的努力，从最初的单一架构转向面向微服务的应用程序环境，这一点也不足为奇。这些工作中的常见问题是，识别可以转换为内聚、独立服务的单体架构应用程序的组件是一项繁琐的手动工作，包括分析软件体系结构视图的许多维度，通常严重依赖于执行提取的专家的经验和专有技术。

本文旨在从算法上解决上述提取问题。为此，我们提出了三种形式化的耦合策略，并将其嵌入到基于图的聚类算法中。耦合策略依赖于来自单一代码基的

（元）信息来构造单一代码的图形表示，这些单一代码由聚类算法依次处理，从而为重构场景中潜在的微服务候选生成建议。此外，还对该方法的原型实现进行了评估。

本文的结构如下：第二节对相关工作进行了概述。第三节正式定义了提取模型、耦合策略和聚类算法。第四节对绩效和质量进行了评估。最后，第六节总结了本文的工作，并讨论了潜在的局限性和未来的工作。

2 相关工作

分解软件系统一直是软件工程研究领域的重要分支 - 从整体中提取微服务是分解系统最初挑战的下一个进化形式。 Parnas 等是最早系统地研究将系统分解为模块的准则的人之一[6]，重点是基于信息隐藏原理的分解。 Komondoor 等提出了一种从旧版企业业务应用程序中提取独立的在线服务的方法。作者指出，迁移整体旧版应用程序所带来的挑战是一项劳动密集型的手动任务，并且该领域的工具支持和自动化并不令人满意。他们提出了一种基于静态分析的方法，该方法针对诸如 COBOL 之类的传统语言中的批处理程序的命令性性质进行了量身定制，并使用向后数据流切片来提取高度内聚的服务。

微服务可以看作是基于服务的计算思想的最新体现。但是，特别是围绕微服务的工作是有限的。 Pahl 和 Jamshidi 进行了系统的二次研究，并对微服务的现有研究机构进行了归类并对其进行分类[7]。作者发现，在当前最新状态下，微服务的工具支持特别缺乏。微服务研究中很大一部分工作只是概念性的，而该综述揭示了比技术解决方案和工具支持研究更多的用例研究[7]。 Schermann 等通过对 42 家不同规模公司的服务计算实践进行实证研究，研究服务计算中的工业实践[8]，并特别关注微服务的发展趋势。在其他方面，对服务规模和复杂性进行了审查。结果表明服务的大小各不相同，但是在实践中很少会遇到极端的服务大小，例如，LOC 低于 100 的超小型服务或 LOC 超过 10000 的超大型服务。另一个有趣的结果是观察到服务是专用的。换句话说，受访者报告说，他们所涉及的服务通常专用于相对狭窄和简洁的任务[8]。最近，从整体中提取微服务已被认为是一种需求[9]。作者提出了一种系统的，但手动的方法，通过在客户端和服务器组件，数据库表和业务区域之间构建依赖图来识别大型整体企业系统中的

微服务。提供一种识别单体式架构代码库中微服务的结构化方法的显着尝试是ServiceCutter [10]。作者提出了一种工具，该工具支持通过图切割实现结构化服务分解。待分解系统的内部表示基于 16 种不同耦合标准的目录，这些标准是从文献和行业专有技术中抽象出来的。软件工程工件和文档（例如域模型和用例）充当生成生成图形表示形式的耦合值的输入。但是，ServiceCutter 无法从整体中挖掘或构造必要的结构信息，因此必须依靠用户以特定的预期模型提供软件工件。

本文试图弥补的研究空白主要有两个方面。一方面，有大量有关传统软件分解和模块化的先前工作，包括诸如传统软件度量，存储库挖掘或静态分析之类的技术。但在所有这些解决方案中，没有一个是专门为微服务提取方案中的应用程序而设计的。另一方面，在微服务迁移，从整体中提取微服务的主题方面，近期工作很少。在解决微服务提取问题的这些尝试中，我们是第一个提供半自动化或形式化方法的，该方法解决了无需大量用户输入即可推荐微服务的问题。本文介绍的方法一方面提供了世界上最好的传统分解技术，另一方面则提供了微服务提取方法和设计原则，以填补图示的研究空白。

3 一种微服务提取模型

本文概述的所有提取策略均嵌入到预定义的提取模型中。它包括三个提取阶段：整体阶段，图阶段和微服务阶段。这些阶段之间有两个转换：构造步骤将整体式转换为图表示，聚类步骤将整体式的图表示分解为候选微服务。根据使用的提取策略，在构建步骤中执行的转换会有所不同。

起点始终是某种形式的版本控制系统（例如 Git1）的实际代码库或已实现应用程序的存储库。对于整体提取过程很重要的代码库方面，已由整体表示形式捕获。整体 M 是三元组 M =（CM，HM，DM），其中 CM 是类文件的集合，HM 是更改历史，而 DM 是向整体 M 贡献代码的开发人员的集合。每个类文件 ck∈ CM 具有文件名，在整装 M 中被认为是唯一的文件路径以及以文本形式的文件内容。

整体 M 的变化历史 HM 被定义为变化事件的有序序列：HM =（h1，h2，...，hn）。每个更改事件 hi 定义为三元组 hi =(Ei，ti，di)，其中 Ei = {c1，c2，...，

cn ｜ ck∈CM}是来自集合 CM 的一组类别，这些类别相加或由更改事件 hi 修改，ti 是发生更改事件的时间戳，di∈DM 是提交相应更改的开发人员。

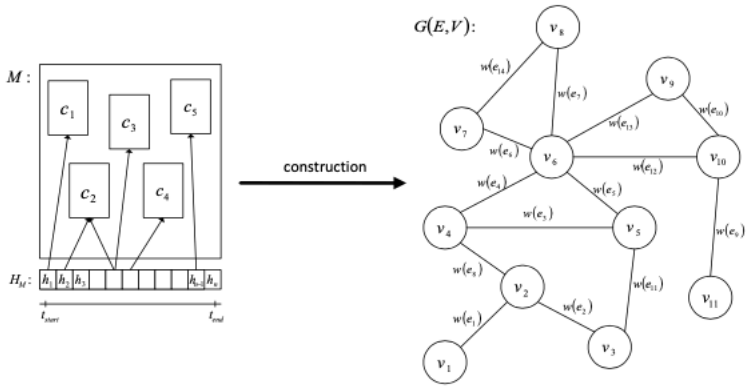

Figure 1. Construction Step

如图 1 所示，通过构建步骤将整块图转换为图形表示。构建步骤采用一种耦合策略来挖掘整块图上的信息，并创建无方向的边缘加权图 G。如果图 G =（E，V），则顶点 vi∈V 对应于整体中的 ci∈C 类。每个图边 ek∈E 具有由权重函数定义的权重。权重函数根据使用中的耦合策略确定相邻边缘之间的耦合强度。较高的重量值表示较强的耦合。请注意，并非来自原始整体的所有类别 ck∈C 都将具有其相应顶点 vi∈V 的边。使用耦合策略时，某些类可能不会表现出与任何其他类的任何耦合，这将导致该类从图中的进一步处理中被丢弃。

然后使用下面描述的图聚类算法将图表示切成小块，以获得微服务的候选对象。

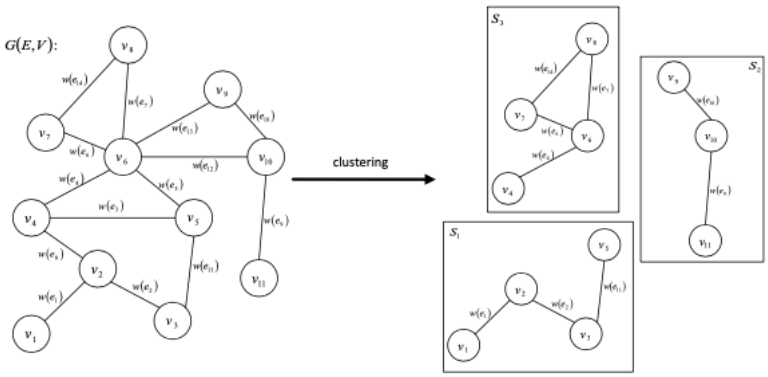

Figure 2. Clustering Step

聚类产生了针对整体 M 的微服务推荐 RM。形式上，微服务推荐是由至少 1 个连通的组件 Sj 组成的森林 RM。森林中的连通组件 Sj 称为微服务，而 N 表示推荐 DM 中微服务的数量。

每个微服务 Sj 是一个有向的有向图 Sj =（CSj，RSj），其中图的顶点由微服务 Sj 中的类的集合 CSj 组成。集合 CSj 是整体中类集合的适当子集。

A.提取策略

1）逻辑耦合策略：单一责任原则[11]指出，软件组件应该只有一个更改的理由。因此，遵循该原理的软件设计应将出于相同原因而更改的软件元素汇集在一起。此外，微服务概念背后的主要好处和设计目标之一就是强制实现强大的模块边界[2]。强大的模块边界和对单一职责原则的遵守在发生更改时会带来好处：如果开发人员必须对系统进行更改，则他们只需要找到要更改的模块，并且只需要了解受限的模块即可。上面提到的两个论点的共同点是，它们基于更改软件元素（例如类文件）的事实。因此，要获得提供上述好处的微服务，必须分析原始整体的变化行为。因此，一起更改的类文件也应该属于同一微服务。这构成了逻辑耦合策略的基本原理。

Gall 等根据应用程序源代码的修订历史[12]，提出了术语逻辑耦合作为隐式耦合的回顾性度量。对于整体 M 的更改历史 HM 中的每个更改事件 hi =（Ei，ti，di），一组更改的类 Ei 包含在从 ti 开始的预定义会话期间更改的所有类文件。令类别 c1，c2∈Ei 是两个在各自的更改事件中更改的不同类别。令 δ 为函数，它表示这些类 c1，c2 是否在某个 commit hi∈HM 中一起改变了：

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

现在，让 Δ 为合计逻辑耦合：

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

在构造步骤中，该策略使用聚合逻辑耦合 Δ 来计算图 G 的边缘上的权重。令 el =（ci，ck）是图 G 上代表原始整体的边缘。 然后通过 w（el）=Δ（ci，ck）

计算权重 w（el）。 通过计算边缘权重来构造图 G，如针对整体 M：=（CM，HM，DM）的所有成对的不同类文件 ck∈C 所示。

2）语义耦合策略：在微服务文学中，源于域驱动设计的有限上下文的概念被提出为微服务及其边界的有前途的设计基础[13]，[2]。根据该原理，每个微服务都应对应于问题域中一个单独定义的有界上下文。这导致可伸缩且可维护的微服务专注于一种责任。因此，从所述概念推断提取策略是有前景的。能够从问题域中正式识别实体的先前工作的一种方案是通过信息检索技术检查源代码文件的内容和语义[14]，[15]。

基本上，该策略将包含关于相同“事物”的代码的类（即领域模型实体）耦合在一起。如先前的工作[14]所示，代码中的标识符和表达式（变量名，方法名等）可用于标识源代码中的高级主题或领域概念。语义耦合将这些表达式作为术语频率逆文档频率方法（tf-idf）的输入[16]。给定预定义的单词集，它为文档计算标量向量。通过计算整体中类文件的这些向量，然后计算成对的不同类的向量之间的余弦相似度，语义耦合策略可以计算出一个分数，该分数表明两个文件在域概念或表达的“事物”方面的相关性在代码和标识符中。

语义耦合策略查看整体 M 中所有成对的不同类 ci，cj∈CM，其中 i != j。对于每对类 ci，cj，执行以下描述的过程。

首先，对每个类文件进行标记，从而为每个类 cj 生成一组单词 Wj = {w1，w2，...，wn}。在标记化过程中，将滤除类中使用的特定于编程语言的所有特殊字符和符号。现在，集合 W 将包含来自相应类代码的所有标识符。类文件中存在与应用程序的任何实际域概念或实体都不相关的单词或标识符，但属于所使用的编程语言或框架的标识符。对于两个类文件 cj，ci，这些停用词都从集合 Wj，Wi 中过滤掉。在项提取期间，项列表 T = {t1，t2，...，tk} 通过组合来自第一类的所有词 wl∈Wj 和第二类的所有词 wl∈Wi 构成。该术语列表用作计算每个类别的 tf-idf 向量的基础。该过程继续为类别 cj 的单词列表 Wj 计算向量 V∈Rn，并类似地为类别 ci 及其单词列表 Wi 计算向量 X∈Rn。向量 V 和 X 的维度 n 等于在上一步中计算出的项列表 T 的大小。向量的第 k 个元素是通过计算相对于相应单词列表的单词列表 T 中第 k 个单词的 tf-idf（文档频率逆文档频率）值[16]形成的。因此，X 的元素可通过∀tk∈T 来计算：xk = tf（tk，Wi）·idf（tk，

Wall），而 V 是类似地通过将 Wi 替换为 Wj 来计算的。上式中的变量 vk 和 xk 分别表示向量 V 和 X 的第 k 个元素。 Wall 是当前计算中所有类文件的所有单词列表的集合 - 在此示例中，Wall = {Wi，Wj}。

相对于文档或单词列表 Wi 而言，术语 tk∈T 的原始术语频率 f（tk，Wi）定义为在 Wi [17]中找到术语 tk 的次数。为了避免由于特定文档中术语的原始频率过高而导致的结果损坏，对数术语频率[17]用作语义耦合策略的术语频率度量。它定义为：

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

反向文档频率测量给定术语 tk 在一组文档或单词列表中的稀有度或共同度 Wall = {W1，W2，... }。 假设将 n 定义为术语 t 在以下项中出现的文档数：n（t）= | {Wi∈Wall：t∈Wi} |。 反向文档频率定义为：

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

在使用 tf 和 idf 度量进行向量计算之后，存在两个向量 X 和 V，每个向量对应于相应类别的每个单词列表。通过使用两个向量之间的余弦值，可以计算出两个原始类相对于其标识符和内容的语义的相似性。令 ek∈E 是表示原始整体 M 的图形 G 中的类文件 ci 和 cj 之间的边缘。此外，让 X 和 V 表示类 ci 和 cj 的结果 tf-idf -vector。边缘的权重 w（ek）由两个向量之间的余弦定义：w（ek）= cos（X，V）。

3）贡献者耦合策略：逆向工程的先前工作已经成功地利用了团队和组织信息来恢复软件工件之间的关系，而这些东西在其他情况下是未被发现的[18]。可以从版本控制系统中恢复所有权体系结构[18]。这样的所有权体系结构揭示了软件的不同组件的团队之间的团队结构和通信模式。将项目迁移到微服务时，组织良好的团队是主要关注的问题。微服务范式建议围绕领域和业务功能组织开发人员的跨功能团队。微服务范式中团队和组织理念的主要目标之一是减少与外部团队的通信开销，并最大程度地提高内部内部通信以及同一服务的开发团队内部的

凝聚力。贡献者耦合策略旨在将这些基于团队的因素合并到一个正式过程中，该过程可用于根据上述观点将类文件分类为类文件。它是通过分析整体版本控制历史中类文件的更改的作者来实现的。

所提出的计算贡献者耦合的过程应用于整体 M 中的所有类别文件 ci∈CM。第一步涉及查找所有修改了当前类别 ci 的历史更改事件 hk∈HM。令γ（hk）表示一个函数，该函数从更改事件 hk =（Ek，tk，dk）返回更改的类文件 Ek 的集合。然后，涉及类 ci 的一组变化事件用 H（ci）= {hk∈HM ｜ ci∈γ（hk）}表示。令σ（hk）表示一个函数，该函数返回唯一可识别的作者 dk∈DM，该作者将变化事件 hk 贡献给整体 M。然后，贡献给当前类文件 ci 的所有开发人员的集合用 D（ci）= {dx∈DM|∀hk∈H（ci）：dx =σ（hk）}表示。在为整体 M 中的所有类别 ci∈CM 计算集合 D（ci）之后，可以计算耦合。在表示原始整体 M 的图 G 中，任意边缘 el 上的重量等于在图中由 el 连接的两个类别 ci 和 cj 之间的贡献者耦合。权重定义为贡献给 ci 和 cj 的一组开发人员的交集的基数：w（el）= ｜ D（ci）∩D（cj）｜。

B.聚类算法

提取过程的下一步是在表示所建议的微服务候选对象的（连接的）组件中剪切图 G。从形式上来说，这意味着从图中删除边缘 ek∈E，以使其余连接的分量收敛。确定要删除的边缘时，必须考虑图的划分，并考虑边缘上的权重 w（e）。在提出的提取模型中，两个类别顶点 ci 和 cj 之间的边缘 ek 上的权重 w（ek）很大，这意味着类别 ci 和 cj 根据所采用的策略属于同一服务候选。因此，该算法应主要偏爱具有低权重的边缘以进行删除。为了确保删除单个边缘始终保证分区，在提取过程中不考虑整个图，而仅考虑最小生成树（MST）。每次删除 MST 中的边缘时，都会导致 MST 划分为两个连接的组件。无法自动确定何时停止对图形进行分区。因此，该算法将目标分区步骤的数量作为输入参数。

如算法清单 1 所示，第一步是边缘的权重反转。目标是使类节点之间的边缘具有较高的权重，以使其保留在相同的微服务候选或连接的组件中。通过直接计算最小生成树，这种推理将失败。因此，删除权重。为了确保删除单个边缘始终保证分区，在提取过程中不考虑整个图，而仅考虑最小生成树（MST）。每次删

除 MST 中的边缘时，都会导致 MST 划分为两个连接的组件。无法自动确定何时停止对图形进行分区。因此，该算法将目标分区步骤的数量作为输入参数。

如算法清单 1 所示，第一步是边缘的权重反转。目标是使类节点之间的边缘具有较高的权重，以使其保留在相同的微服务候选或连接的组件中。通过直接计算最小生成树，这种推理将失败。因此，在图 G =（V，E）的边缘 ek∈E 上的权重 w（ek）可通过 $w(e_k) = \frac{1}{w(e_k)}$ 求逆。因此，图 G 的最小生成树 MST（G）将根据计算的耦合或权重保留最重要的边缘。 MST 边缘集 edgeMST 由 KRUSKAL 算法计算得出[19]。设置的 edgeMST 根据其倒置权重进行排序，然后倒置，以使倒置权重最高（因此耦合度最低）的边缘首先出现在列表中。

Algorithm 1 MST Clustering Algorithm

```
1: function CLUSTER(edges, n_part, s)
2:     for e ∈ edges do
3:         e.weight ← 1 / e.weight
4:     end for
5:     edges_MST ← KRUSKAL(edges)
6:     edges_MST ← SORT(edges_MST)
7:     edges_MST ← REVERSE(edges_MST)
8:     n ← 1
9:     while n <= n_part do
10:        edges_MST[0].delete()
11:        n ← DFS(edges_MST)
12:    end while
13:    components ← REDUCECLUSTERS(edges_MST, s)
14:    return components
15: end function
```

然后，该算法进入迭代边缘删除循环，如算法清单 1 中的第 9-12 行所示。该算法尝试的分区数由外部参数 npart 定义。在每个迭代步骤中，从 MST 中删除耦合度最低的边，并通过在深度优先搜索（DFS）中遍历其余边来计算其余连接的组件。执行完 npart 分区后，其余边缘将传递给 ReduceClusters 过程，该过程确保没有异常大的类节点簇。由于观察到整体结构中存在类文件，这些类文件由于在所使用的框架和语言中具有特殊作用，因此具有非常高的耦合度，因此这是必要的。我们将这些文件视为离群值，并在 ReduceClusters 中处理它们：包含的类节点的数量大于预定义参数 s 的所有连接的组件将通过删除具有最高程

度的类节点来进一步划分。这样可以减小群集的大小，同时保持其余连接组件内部的内部耦合较高。

4 评估

我们在开源 Java 项目 2 中将提出的提取模型和策略实现为概念验证。我们还在 AngularJS 3 中编写了一个基于 Web 的前端。我们针对两个研究问题评估了我们的方法：

RQ1：就执行时间而言，已实现原型的性能如何？

RQ2：原型生成的微服务建议的质量如何？

为此，设计了一个评估，其中包含一组来自开源项目的特定示例代码库，并针对 RQ1 进行了一系列性能评估实验，而 RQ2 通过使用自定义指标进行一系列质量评估得到了回答。整体代码库的样本集由使用 Git VCS 的开放源代码存储库组成。示例项目是用 Java，Python 或 Ruby 编写的 Web 应用程序，并使用 ORM 技术进行数据管理。由于 ORM 系统使用类来表示数据库表，因此此类应用程序很适合用于基于类的抽取模型，如本文所述。样本存储库的版本历史记录中包含 200 到 25000 个提交，而 LOC 中的项目大小在 1000 LOC 到 500000 LOC 范围内。此外，所有示例项目都有 5 至 200 名作者，这些人贡献了版本历史记录中的更改。

A. 表现

表 I 列出了关于不同耦合策略的示例项目的执行时间（以秒为单位），与存储库属性的提交数（h），贡献者数（c）和 LOC 中的代码大小（s）有关。

语义耦合：在示例项目上测得的执行时间证实了直觉，即语义耦合策略的性能主要且直接受到 LOC 中存储库中代码总大小的影响。这一点不足为奇，因为概述的 tf-idf 相似度计算需要逐行处理类文件的内容。

逻辑耦合：执行时间随着提交次数的增加而增加。但是，执行时间会有一些较大的变化，并且会随着执行时间值的增加而增加。例如，django-oscar 和 chiliproject 样品的 LC 执行时间。逻辑耦合的计算涉及所分析的整体结构 M 的历史 HM 中某个更改间隔的更改集中的所有类文件的幂集。在那些情况下，此更改集的平均大小将主导历史记录大小，并且趋向于导致峰值。

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

贡献者耦合：虽然在表 I 中可以看到历史记录大小的影响，但由于示例项目的类文件数| C |的存在，存在一些极端的异常值。异常高，因此支配了历史记录大小的影响，并导致执行时间增加。这是由于以下事实：计算贡献者耦合既涉及遍历整个历史，又涉及遍历整块中的类的成对排列。因此，离群值是由类数主导执行时间增长的情况来解释的。

通常，所有性能实验均显示令人满意的性能水平。它们表明我们的方法可以用于不同的场景（例如，针对软件架构师的推荐系统或持续集成中）。

B. 质量

尽管在诸如面向对象设计等领域中已经建立了公认的质量度量标准[20]，但是微服务研究却显示出很少的此类努力。因此，我们使用自定义指标作为代理来捕获和比较推荐质量。我们介绍了团队规模缩减比率指标（tsr）和平均域冗余指标（adr）的评估。可以在[21]中找到有关度量依据和结果的更详细的概述。

1）减少团队规模：经常被认为是微服务的主要好处之一是改善了团队结构，降低了团队的复杂性和规模[13]。减少团队规模可减少通信开销，因此，可以将更高的生产力和团队关注重点放在实际的领域问题和它所负责的服务上。我们将

tsr 指标用作此面向团队的质量方面的代理。令 RM 为整体 M 的微服务推荐。tsr 计算为 RM 中所有微服务候选者的平均团队规模除以原始整体 M 的团队规模。

图 3 显示了研究中所有 21 个项目的不同耦合策略（LC，CC，SC）或其组合的相应结果。 tsr 值为 1 表示每个微服务的新团队规模与整体组件的原始团队规模一样大，而较低的值则意味着团队规模的减小。

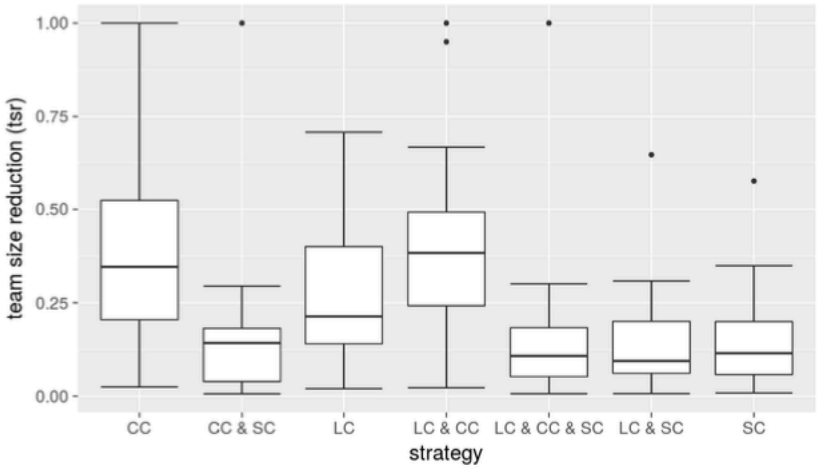

Figure 3. Boxplot of the team size reduction ratio (tsr) results

图 3 显示，所有实验系列的中位值均低于 0.5，与所使用的策略组合无关。除其中一种组合外，所有组合的上四分位数都低于 0.5 的 tsr。这些观察结果表明，在团队规模改善方面，提取方法和策略表现良好。在大多数实验中，团队规模缩小为原始规模的一半甚至更低。查看不同的基本提取策略的结果之间的差异，很明显，语义耦合策略通过显着胜过逻辑耦合和贡献者耦合策略，在该度量标准中显示出最佳结果。我们进行了 Welch t 检验以支持这一说法：比较 LC 和 SC 分布的测试表明存在显着差异（p 值<0.05），而 CC 和 SC 实验的分布比较则显示出甚至更大的差异（p -值<0.01）。涉及 SC 策略的策略组合似乎始终比其他策略产生更好的 tsr。例如，我们的 t 检验显示 LC 和 CC 以及 LC 和 CC 和 SC 的分布存在显着差异（p 值<0.01）。涉及 SC 策略其他组合的测试的 p 值也暗示了类似的差异（所有 t 检验的结果都可以在[21]中找到）。

一个引人注目的观察结果是 CC 策略的较高四分位数值和较高的较高晶须。在仔细检查实验结果后，很明显，某些样本项目的 tsr 等于 1，这意味着没有任何降低。这仅在涉及贡献者耦合策略的实验中发生。通过在应用 MST 聚类算法之

前查看耦合图可以找到对此的解释。在所有 tsr 的值为 1 的情况下，耦合图仅由一个大的连接分量组成，并且相邻节点呈星形排列。中心的高度节点对应于一个类文件，该类文件已由参与整体历史的每个贡献者之一进行了更改。按照贡献者耦合的定义，此文件将与整体中的每个其他类文件耦合。朝向恒星外部的边缘的权重比靠近中心的边缘的权重弱。这会导致聚类算法的退化行为，尽管尽管在每个步骤中都删除了权重最低的边缘，但组件的数量并没有增加，而是保持在 1 个相连的组件上。仅外部节点被迭代地从星形中切除。最后，其余图仍将始终包含中心类节点，并且它将是唯一剩余的连接组件，因此也是唯一剩余的微服务候选者。根据定义，由于中央节点将所有贡献者都显示为原始整体，因此该剩余候选人的定义将与原始整体具有相同的团队规模。

2）平均域冗余：应用程序的问题域可以看作是一组有界上下文，每个有界上下文具有明确的职责和明确定义的接口，该接口定义了哪些模型实体要与其他有界上下文共享[13]。而且，有利的微服务设计可以避免跨服务重复职责。如 Schermann 等在他们的实证研究中报告[8]，微服务致力于处理相对狭窄和简洁的任务。因此，我们将平均域冗余度量标准作为代理来计算，以指示服务之间在 0 到 1 之间的归一化规模上特定于域的重复或冗余的数量，其中我们倾向于使用具有较低 adr 值的服务建议。

图 4 表明 7 种策略组合中的 6 种提供了良好的域冗余分布。在这 6 个实验系列中，有 4 个的中位数显着低于 0.25，这表明在推荐的服务之间，微服务源代码中域内容的不到四分之一是多余的。此外，人们可能会凭直觉地认为语义耦合策略将在此度量标准上执行得很好-的确如此-因为它针对特定领域的聚类优化了提取图。看一下图 4 中的结果，就可以证明这种直觉：孤立地由 SC 策略产生的 adr 值的中位数最低。 t 检验结果部分证实了这一观察结果（涉及独立的 SC 实验的 5 个比较中，有 3 个比较的 p 值<0.05）。 p 值大于 0.05 的两种情况都涉及两个比较数据集中的 SC 策略，一次是孤立的，一次是与其他策略的组合。这解释了这些 t 检验的较小差异。

尽管 CC 实验的 adr 中位数远低于硬阈值 0.5，但四分位数的上限非常高，结果显示，与其他策略相比，偏斜更大，值更高。 这种现象可以归因于这样一个事实，即开源项目中开发人员之间的技能分配不是面向领域的，而是取决于技术

的。 这意味着，前端开发人员倾向于对相同文件做出贡献，而与这些文件的域内容无关。 后端或数据库开发人员也是如此。 由于贡献者耦合的计算方式，尽管语义和领域特定的内聚性较低，但这种情况仍导致类文件耦合到同一微服务中。 因此，这导致不同服务候选之间更高的域冗余。

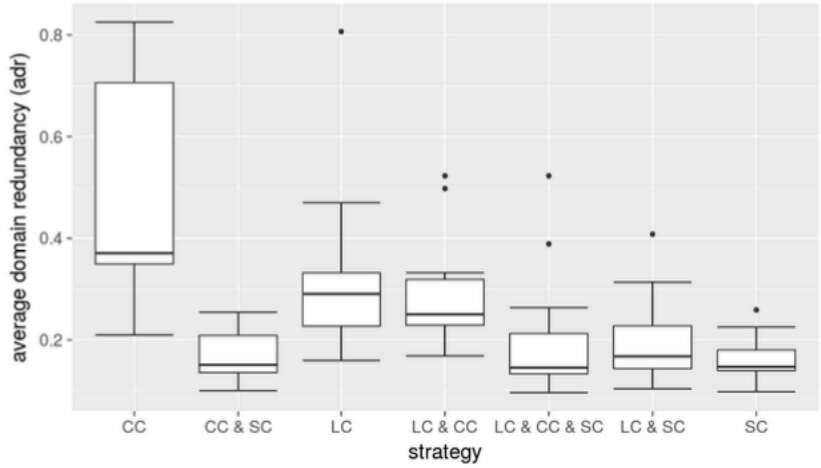

Figure 4. Boxplot of the average domain redundancy (adr) results

5 局限性与未来工作

一个局限性是这样一个事实，即提取模型基于作为策略和图形中计算的原子单位的类。 尽管此前提很适合我们的基于图的提取模型，但它限制了重构整料时的可用余地。 使用方法，过程或功能作为提取的原子单位可能会提高代码重新排列和重组的粒度和精度，因此将在以后的工作中加以考虑。

我们的提取模型通过假设存在一个 ORM 系统来规避数据库分解难题，该系统将数据模型实体表示为类，提取算法将其视为与任何普通类一样的类。 当然，在实践中微服务通常会缺少这样的组件，因此如何共享或将现有数据库分配给不同服务的未解决问题仍然是本文的局限性，也是对该领域未来工作的挑战。

6 总结

我们引入了一个正式模型，该模型能够从单体式架构软件体系结构中提取微服务。我们基于该模型设计并实施了提取策略，并基于 21 个 Java，Ruby 和 Python 开源项目对方法的性能和质量进行了评估。绩效评估表明，在大多数情况下，我

们的方法相对于修订历史记录的大小（逻辑和贡献者耦合）进行缩放。质量评估表明，我们的方法可以将微服务团队的规模减小到整体团队的规模的四分之一甚至更低。

大多数策略组合在降低平均域冗余方面表现良好。在大多数情况下，域冗余的中值始终为0.3甚至更低。只有单独使用贡献者耦合生成的建议而没有任何其他策略，才会显示分布更广泛的结果，这些结果倾向于更高的冗余度。

质量评估可以指导软件架构师根据他们的需要相应地使用我们的方法（即减少团队规模和降低提取服务的域冗余）。对于未来，我们正在考虑将更多细粒度的软件工件（例如方法）作为我们方法的输入，这可能会改善代码重新排列和重组的粒度和精度。

7 致谢

导致这些结果的研究已获得欧洲共同体第七框架计划（FP7/2007-2013）的资助，协议号为No.610802（CloudWave）。