Automated Microservice Identification in Legacy Systems with Functional and Non-Functional Metrics

1st Yukun Zhang
Nanjing University of Aeronautics and Astronautics
Nanjing, China
RISE, Southwest University
Chongqing, China
magiczhangyukun@gmail.com

2st Bo Liu*
RISE, Southwest University
Chongqing, China
liubocq@swu.edu.cn

3st Liyun Dai
RISE, Southwest University
Chongqing, China
dailiyun@swu.edu.cn

4st Kang Chen
RISE, Southwest University
Chongqing, China
yishanchuan@outlook.com

5st Xuelian Cao
RISE, Southwest University
Chongqing, China
xuelian7@email.swu.edu.cn

Abstract—Since microservice has merged as a promising architectural style with advantages in maintainability, scalability, evolvability, etc., increasing companies choose to restructure their legacy monolithic software systems as the microservice architecture. However, it is quite a challenge to properly partitioning the systems into suitable parts as microservices. Most approaches perform microservices identification from a function-splitting perspective and with sufficient legacy software artifacts. That may be not realistic in industrial practices and possibly results in generating unexpected microservices. To address this, we proposed an automated microservice identification (AMI) approach that extracts microservices from the execution and performance logs without providing documentation, models or source codes, while taking both functional and non-functional metrics into considerations. Our work firstly collects logs from the executable legacy system. Then, controller objects (COs) are identified as the key objects to converge strongly related subordinate objects (SOs). Subsequently, the relation between each pair of CO and SO is evaluated by a relation matrix from both the functional and non-functional perspective. We ultimately cluster classes(objects) into the microservices by optimizing the multi-objective of high-cohesion-low-coupling and load balance. The usefulness of the proposed approach is illustrated by applying to a case study.

Index Terms—microservice architecture, automated microservice identification (AMI), legacy system restructuring, non-functional metrics, object oriented programming.

I. Introduction

MICROSERVICE architecture has recently drawn widely attention from both academia and industrial communities. It describes a particular way to construct a single application as a set of small, well-defined, high cohesive, loosely coupled, independently deployable, distributed and autonomous services [1], namely, the microservices. Each microservice is designed to implement a single business capability which owns a relatively integrated domain model including data, logic, and behavior. It runs in its own process (e.g., a container like Tomcat), communicating with each other via lightweight mechanism(e.g., RESTish protocols like HTTP or lightweight message bus like RabbitMQ), and evolving independently (i.e., precisely updating and deploying an isolated microservice without bothering other ones). And the above features provide microservice based applications with the advantages of maintainability, scalability and evolvability over the monolithic software systems.

Although there seems be an overwhelming trend towards the microservice architectural style in industries, it is not yet the dominant form of software systems. The reasons are manifold, but a main reason is that many companies have already got running monolithic systems which are still of great value in business. While the legacy systems continuously bring values to the owners they suffer the inconveniences (e.g., a change made to a small part of a system requires the entire monolith to be rebuilt and redeployed). One feasible option for them is to re-engineer legacy monolithic applications by extracting microservices from the existing software artifacts. Many large companies like Netflix and Amazon have chosen this option to reap the benefits of the technological advances while conserving the current business value of legacy systems [2].

Inspired by the approach of migration from object-oriented systems to service oriented systems [3], there are two main phases to migrate a legacy monolith into microservices: (1) legacy analysis, and (2) microservices implementation. The first phase is to identify candidate microservices by analyzing the available legacy software artifacts. And the second phase is to code the identified

* Corresponding author: Bo Liu (liubocq@swu.edu.cn).

978-1-7281-4659-1/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSA47634.2020.00021

candidate microservices into real usable microservices. The first phase is vital because how properly to split the functions of a monolithic system as microservices, may dramatically affect the maintainability, scalability, and evolvability of the rebuilt microservice architectural system. However, many companies have to perform this task with mostly dependency on the experience of designers [4]. It is an obviously laboring-intensive and error-prone task [5]. Accordingly, the current paper tries to propose a solution for automated microservice identification (AMI) from legacy monolithic systems.

A great deal of literature has designed various function-splitting based approaches on the topic of AMI, following the principle of high cohesion and low coupling. Nevertheless, there are still drawbacks: (1) many of the proposed approaches assume the availability of sufficient documentation, analysis and design models, or source code. That is not realistic in industrial practices, because many legacy systems are usually built with poor documentation or even few modelling efforts. Moreover, many legacy system owners may refuse to provide source code for the consideration of business privacies. (2) majority of existed work mainly uses single-aspect metrics (e.g., from a function-splitting perspective) to identify candidate microservices. Accordingly, there may be a gap between identified microservices and expected ones (e.g., only functionality-oriented identification may result in unbalance in load among microservices).

In addressing the above challenges, we propose a new AMI approach that includes the following features: (1) we use runnable programs as the only required software artifact. And a systematic framework is designed to collaboratively analyse both the software execution log and performance-monitoring log, where the functional and non-functional information are generated as the input of the current approach. (2) we design an AMI algorithm that: (i) identifies the key objects, namely, the controller objects (COs)[1] [6] [7], to converge strongly related subordinate objects (SOs)[2]; (ii) quantifies the relationship between each pair of CO and SO as well as the related classes with both functional and non-functional metrics; (iii) optimally partitions the monolith into a suitable set of microservices considering both functionalities and performance.

The remainder of the paper is structured as follows: Section 2 presents a running example to motivate our work. The approach is introduced in Section 3. We evaluate our approach in Section 4 and conclude in Section 5.

[1]The concept of controller objects follows the idea of Controller Pattern: a controller object is a permanent object that is created before the execution of the related service and will not be destroyed before the end of the service. It acts as the facade which is responsible for handling external input events.

[2]The concept of subordinate objects (SOs) is relative to that of COs: An object that is not a CO is an SO.

II. Motivating Case: JPetStore

In order to explain our proposed AMI approach, we use JPetStore[3] as the legacy system. It is a simple but well-implemented web application, which contains main business activities in online pet transactions. Therefore, it is widely used as a benchmark in large number of software engineering related research [8] [9].

The use cases of JPetStore are as follows:

- Account Management: A user register with his personal information; then the registered user will get an account, with which the user can login the application with the registered username and password. A user can also browse the shopping history related to his account.
- Category Management: Categories of pets (e.g., Fishes, Dogs, Reptiles, Cats and Birds) are displayed on the electronic shelf. Each category contains a list of concrete products(pets), and each product has price and related description.
- Shopping Cart Management: A user can add any preferred pets to his shopping cart when he surfs the pet categories. Each item in the shopping cart includes the information of in-stock status, quantity to buy, price, subtotal, etc. A link to checkout is provided, but an order is not created before the related payment is conducted.
- Order Management: When a user is going to pay his shopping cart, a payment form is displayed. The payment form requires the information of credit card and shipping address. When payment is conducted, an order is created which includes payment and shipping details along with the shopping items.

III. AMI Approach for Microservices Identification

The main framework of our proposed AMI approach is illustrated in Figure 1. We use JPetStore as the running example. The open source tools, Kieker Monitoring Framework[4] and Java VisualVM[5], are used to generate the execution and the performance logs, respectively. We first analyse the execution log to find out all the involved objects and invocations between any pair of objects in legacy system (we assume the execution log is sufficiently collected). Identification for COs is initially performed, and the overall set of objects is partitioned into two sets of the COs and the SOs. Moreover, an object-to-object (O-O) relation evaluation matrix is generated to quantify the relation between each pair of CO and SO from a functional perspective. Then, the O-O relation matrix is reduced into

[3]It is a web application of online pet shop, which is built on the open source frameworks of MyBatis, Spring Framework and Stripes, and is implemented with Java Programming Language. The running application is located at https://jpetstore.cfapps.io/catalog; and the source code can be accessed at https://github.com/mybatis/jpetstore-6/

[4]http://kieker.sourceforge.net

[5]https://visualvm.github.io

TABLE I: The properties of a *line* (method call)

Name	Description
type	BeforeOperationEvent, AfterOperationEvent, BeforeConstructorEvent, and AfterConstructorEvent
generation time	indicates when the current line is generated
trace id	a globally unique ID labeling a request
order id	calling order of calling stack of the method
method identifier	complete signature of an invoked method
class identifier	complete signature of an invoked class
object id	a globally unique ID labeling a running object

a class-to-class (C-C) relation evaluation matrix, where the CPU and memory parameters are added by analysing the performance logs. Based on the C-C relation evaluation matrix, a genetic algorithm based approach is developed to optimise identification of microservices with the multi-objectives of high-cohesion-low-coupling and load balance of CPU and memory consumption.

A. Logs Generation

Instead of analysing source code or other software artifacts, the proposed approach uses the executable WAR package only. The open source tool Kieker is used to collect the execution log by inserting probes to the executable file, while VisualVM is configured to monitor the runtime CPU and memory consumptions of classes. Test cases are designed for black box testing. The tool Apache JMeter[6] is used to perform testing automatically. Both method invocation information as well as the CPU and memory consumptions are recorded into an execution log file and performance log files, respectively.

We first analyse the execution log. The elementary units of the execution *log* file are traces. A *trace* represents a series of method calls responding to a business request (e.g., click a button to browse item catalogs), and each method call is represented as a couple of a start *line* (indicates a BeforeOperationEvent or a BeforeConstructorEvent) and an end *line* (indicates an AfterOperationEvent or an AfterConstructorEvent) in a *trace*. Any couple of the start- and end- *line* has the same properties except their types (as shown in TABLE I). For example, if a user clicks a link of "Fish" on the main page of JPetStore, a trace will be generated to record the sequence of method invocations. We can simply formulate the related concepts in the generated log as follows:

$$\begin{aligned} Log =& \{Traces\} \\ Traces =& \{Lines\} \\ Line =& \{type, generation\ time, trace\ id, \\ & order\ id, method\ identifier, \\ & class\ identifier, object\ id\} \end{aligned}$$

The performance logs provide the following information:

a) CPU consumption of each method call of an object in a trace;
b) Average memory consumption of each class.

To simplify the addressed problem, we aggregate the raw CPU consumption of each object to obtain the average data of each class. Accordingly, both the obtained CPU and memory data are unified to reflect the performance of classes.

B. Data Preprocessing

Data preprocessing on log files is required to generate inputs for the proposed AMI algorithm. To this end, the information provided by the *lines* of *traces* are used to create a set of $\{MessageRecords\}$. Each record $MessageRecord$ within the set can be formulated as follows:

$$\begin{aligned} MessageRecord =& \{< senderOID, senderClassSig >, \\ & < receiverOID, receiverClassSig >, \\ & MethodSig\} \end{aligned}$$

Obviously, a record $MessageRecord$ provides information of the sender, receiver, and the method invocation between them. The set $\{MessageRecords\}$ is the input of the AMI algorithm.

C. Microservice Identification

1) Identification for controller objects: A universal set of all runtime objects $\{Objects\}$ can be obtained by scanning the $\{MessageRecords\}$. By following the idea of Controller Pattern [6], a CO acts as the facade which is responsible for handling external input events. The class related to a controller object may usually be identified as the interface of a candidate microservice. We need to identify the controller objects from the set $\{Objects\}$. A CO should be a permanent object [7]. That means it should have existed before the start of the execution traces and it will not be destroyed during the execution. Accordingly, we can obtain the set of COs by scanning the set $\{Objects\}$ with the following checking criteria:

a) all COs should only come from message senders;
b) any CO should never be a CREATE message receiver;
c) each CO should be of singleton pattern;

After obtaining the set $\{COs\}$, the set of subordinate objects $\{SOs\}$ is accordingly obtained by computing the complementary set of $\{COs\}$, such that $\{SOs\} = \{Objects\} \setminus \{COs\}$, where the symbol $A \setminus B$ indicates to delete the elements of the set B from the set A.

2) Construction of Relation Evaluation Matrixs: A OO relation evaluation matrix should be firstly constructed as shown in Table II , where a CO indicates a controller object, SO indicates a subordinate object, and R_{ij} indicates the evaluated relation between the $i-th$ CO and the $j-th$ SO.

[6]https://jmeter.apache.org/

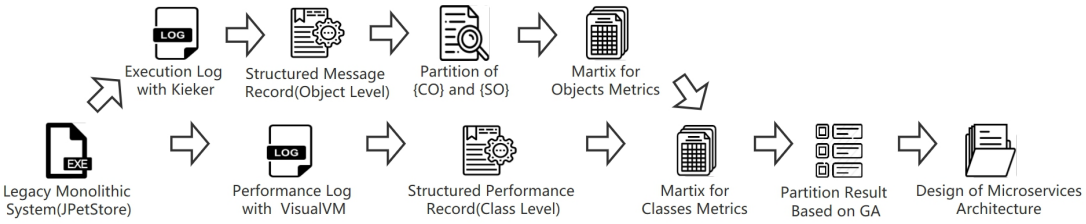

Fig. 1: Framework of AMI approach

TABLE II: O-O relation evaluation matrix

		Subordinate Object				
		SO_1	SO_2	SO_3	SO_4	...
Controller Object	CO_1	R_{11}	R_{12}	R_{13}	R_{14}	...
	CO_2	R_{21}	R_{22}	R_{23}	R_{24}	...
	CO_3	R_{31}	R_{32}	R_{33}	R_{34}	...
	CO_4	R_{41}	R_{42}	R_{43}	R_{44}	...
	...	...	...	...	...	...

We firstly use the functional metrics to quantify the relation in each pair of CO and SO. The functional metrics (Fct) include the invocation strength ($InvocStren$) and the invocation frequency ($InvocFreq$). We formulate them as follows:

$$R =< Fct >$$
$$Fct =< InvocStren, InvocFreq >$$

The invocation strength ($InvocStren$) is evaluated by various types of messages. We define three constants to represent different invocation strength for three types of messages:

a) $IStr_c$ - the invocation strength constant for CREATE messages
b) $IStr_u$ - the invocation strength constant for UPDATE messages
c) $IStr_r$ - the invocation strength constant for READ messages

We define the above constants because different types of messages should have different impacts on quantified relation evaluation. The CREATE message has the strongest relation between objects; the UPDATE message takes the second place; and the READ message indicates a weak relation.

The invocation frequency ($InvocFreq$) indicates how many times an invocation takes place between objects. We also expect different invocation frequencies could have different impacts on quantified relation evaluation, just as the invocation strength ($InvocStren$) does. We similarly define the following constants:

a) $IFre_c$ - the invocation frequency constant for CREATE messages
b) $IFre_u$ - the invocation frequency constant for UPDATE messages
c) $IFre_r$ - the invocation frequency constant for READ messages

We define a formulation as follow to evaluate the functional relation between objects:

$$\begin{aligned} eval(Fct) =& IStr_c \times (IFre_c)^{kc} \\ &+ IStr_u \times (IFre_u)^{ku} \\ &+ IStr_r \times (IFre_r)^{kr} \end{aligned} \quad (1)$$

where kc, ku, and kr indicate the respective frequency of the various messages.

The above formulation is defined in consideration of the following principles:

(1) The invocation strength should take a dominant place in the formulation compared with the invocation frequency. Accordingly, we suggest that the strength constant of a CREATE message $IStr_c$ should be set at least one magnitude more than that of a UPDATE message $IStr_u$; so does a UPDATE message to a READ message $IStr_r$ (e.g., $IStr_c = 100, IStr_c = 10, IStr_c = 1$). And, all the invocation frequency constants $IFre_c$, $IFre_u$, and $IFre_r$, should better be set slightly larger than 1.0 ($IFre_c > IFre_u > IFre_r$), such that these constants could not result in a faster speed in relation evaluation by comparing with the invocation strength constants.

(2) Moreover, although the invocation strength places a main role in functional relation evaluation, the invocation frequency could have a chance to be superior as the number of invocation is large enough.

We design the following algorithm to evaluate the relation between pairs of objects and that between pairs of classes.

Input: $\{MessageRecords\}$

Output: A C-C relation evaluation matrix

Step I: Initialize

a) Partition $\{Objects\}$ into $\{COs\}$ and $\{SOs\}$ then form the original O-O relation evaluation matrix as shown in TABLE II
b) Initialize each cell (R_{ij}) of the matrix with null value

Step II: Iteratively evaluate CREATE message between COs and SOs.

TABLE III: C-C relation evaluation matrix

		Subordinate Class		
		$SubClass_1$	$SubClass_2$	...
Controller Class	$CtlClass_1$	R'_{11}	R'_{12}	...
	$CtlClass_2$	R'_{21}	R'_{22}	...
	$CtlClass_3$	R'_{31}	R'_{32}	...
	...	...	...	...

a) For each CO in $\{COs\}$, search in $\{MessageRecords\}$ for the objects (denoted by $\{Level1Object\}$) that are directly created by the CO, and then update the corresponding R_{ij} with Equation (1) respectively.
b) For each object in $\{Level1Object\}$, search in the rest objects of the set $(\{MessageRecords\} \setminus (\{COs\} \cup \{Level1Object\}))$ (where the symbol $A \setminus B$ indicates to delete the elements of the set B from the set A), for the objects (denoted by $\{Level2Object\}$) that are directly created by the level 1 objects, and update R_{ij} between the current CO and the level 2 objects, respectively.
c) Recurse the search for $\{Level3Object\}$, $\{Level4Object\}$, ..., until rest set of $(\{MessageRecords\} \setminus (\{COs\} \cup \{Level1Object\} \cup \{Level2Object\} \cup \{Level3Object\} \cup ...))$ is empty, or no new object is found in the process.

Step III: Iteratively evaluate UPDATE message between COs and SOs in a similar way.

Step IV: Iteratively evaluate READ message between COs and SOs in a similar way.

Step V: Reduce O-O relation evaluation matrix by merging objects with same classes, and generate the C-C relation evaluation matrix accordingly (as shown in Table III)

a) In the new matrix, each R'_{ij} indicates the quantitative relation between the $i-th$ controller class (corresponding to a CO) and the $j-th$ subordinate class (corresponding to a set of SOs with same type). We formulate the R'_{ij} as follows by adding the non-functional metrics.
b) We simply obtain the Fct' by summing the corresponding Fct of the merged objects.
c) The evaluation data of CPU and $Memory$ are collected from the performance data which is described in the previous section of A. Logs Generation.

$$R' =< Fct', NonFct >$$
$$NonFct =< CPU, Memory >$$

3) Genetic Algorithm for Candidate Microservice Identification: Based on the C-C relation evaluation matrix, we can obtain a class set $\{classes\}$ which contains all controller classes $\{CC\}$ and subordinate classes $\{SC\}$. A candidate microservice can be defined as $< CC_i, \{SC\}_i >$, where $i \in [1, |\{CC\}|]$. Accordingly, a partition of the $\{classes\}$ can be a set of candidate microservices $\{microservices\}$ and there is no overlap between any pair of microservices.

To achieve a proper identification of candidate microservices, we optimise the following objectives from the perspectives of functional metrics, CPU consumption, and memory consumption, respectively.

$$max \quad FO = \sum_{1}^{n} Fct'(microservice_i) \tag{2}$$

$$min \quad CO = \sum_{i \neq j}^{n} ||CPU'(microservice_i) - CPU'(microservice_j)|| \tag{3}$$

$$min \quad MO = \sum_{i \neq j}^{n} ||Mem'(microservice) - Mem'(microservice_j)|| \tag{4}$$

The functional metrics related objective (FO) in Equation (2) optimises the overall functional relation evaluation of a partition on the set $\{classes\}$, where n denotes the number of microservices in the pratition. $Fct'(microservice_i)$ denotes the sum of $eval(Fct)$ between CC and SC in the $microservice_i$. The larger value of FO indicates the better high-cohesion-low-coupling in identification of microservices.

The CPU related objective (CO) in Equation (3) indicates the objective of CPU balance of a partition on the set $\{classes\}$. The smaller value of CO indicates the better balance of the CPU consumptions among microservices.

The memory related objective (MO) in Equation (4) indicates the objective of memory balance of a partition on the set $\{classes\}$. The smaller value of MO indicates the better balance of the memory consumptions among microservices.

It is a typical multi-objective optimisation problem [10] [11]. To search for a feasible solution, we design a genetic algorithm enlightened by the Non-dominated Sorting Genetic Algorithm [12]. A partition on the $\{classes\}$ can be encoded as an individual. The chromosome of an individual can be illustrated as Fig 2.

Given n controller classes and m subordinate classes obtained in JPetStore. As shown in Fig 2, an individual is encoded as an array with m genes. Each gene can be valued with an integer from the range $[1, n]$. An integer x on the $i-th$ gene of the chromosome indicates the $i-th$ class acts as a subordinate class belonging to the $x-th$ controller class, such that the $x-th$ controller class and all the belonged subordinate classes form a microservice candidate with the $x-th$ controller class acting as the facade (interface). For example, a chromosome of

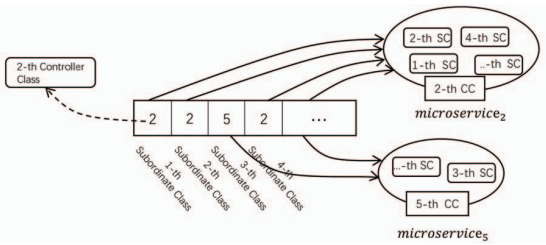

Fig. 2: Chromosome of an individual

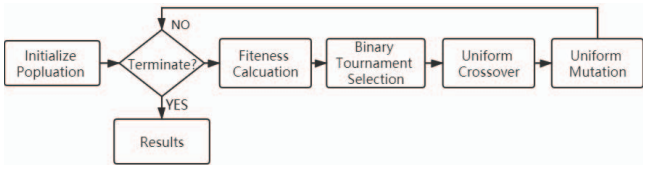

Fig. 3: Framework of genetic algorithm

$[2, 2, 5, 2....]$ indicate the first two and the fourth classes belong to the $2-th$ controller class, thereby forming a microservice candidate with the $2-th$ controller class as the interface and the first two as well as the fourth classes as the belonged subordinate classes. And so forth, the third class belongs to another microservice candidate with the $5-th$ controller class as its interface.

As shown in Figure 3, we design the operators of binary tournament selection, uniform crossover and uniform mutation to evolve the offspring population in the genetic algorithm. The binary tournament selection operator randomly picks up two individuals from current population, evaluating them with a crowded-comparison operator [12], then keeping the better one. The better ones have chances to give birth offsprings by using crossover and mutation operators. The uniform crossover operator generates two offsprings by swapping a randomly selected genes from their parents (the above best ones). The uniform mutation operator randomly changes some genes to generate new offsprings.

When reaching a maximum number of generations, the acceptable approximately optimal individuals are located in the Pareto front. Each solution in the Pareto front is evaluated by the tuple $< FO, CO, MO >$, which indicates its finesses on functionality, CPU and Memory, respectively. We then sorted these individuals in descending order by their FO values, and the individual ranked first in the sorted collection is the selected optimal solution of microservice identification. The reason for ordering by FO values is that, in many real industrial practices, software architects usually regard the functionality related metrics as the primary criteria to identify microservices (to follow the principle of "high cohesion, low coupling"), while taking other metrics as the secondary criteria [13].

IV. Case Study

We illustrate the usefulness of the proposed AMI approach by applying it to a medium-sized system: Production SSM [7]. It is an open source ERP system implemented in Java. It provides supports for enterprise resource management, including production planning, inventory control, quality control, process control, equipment management and human resources management. Compared with JpetStore, it involves more domain concepts and complex business logic. Table IV illustrates the different scales between the two projects.

TABLE IV: Size of selected cases

Case	Class	LOC	Package
JPetStore	39	4932	4
Production SSM	227	38094	9

We choose Production SSM as the studied case to evaluate our approach because: (1) The project is a typical three-layer web application [14], and it is a much representative architectural style in existing legacy systems [15]. (2) The project related documents are insufficient, which is just the case that we say a typical challenge in refactoring into microservices.

A. Preparation

The following tools are initially used to generate logs:

1) Kieker: Kieker monitors runtime method invocations by inserting probes into the target executable programs. Various probes are provided by Kieker for different purposes of monitoring. In this case study, we use the probes of constructorExecutionObject and operationExecutionObject, to monitor calls on constructors and common methods, respectively. Ideally, all execution paths of the case project should be covered through the testing. It is quite a challenge that we have not enough documents specifying the requirements of the system. We therefore design test cases with the following strategies to improve the functional coverage with the best efforts.

a): Try to analyse the system operations from the perspective of end users by actually using the Web application from the Web pages, until all visible UI interfaces are accessed

b): Try to analyse interactions among objects from the sequence diagrams transformed from the execution logs (provided by the tool Kieker), in order to understand the possible functionalities

c): We access the running Web application by using different users simultaneously, and trying to cover each possible functionality and executing each system operations at least 100 times (by using the tool Apache JMeter).

2) VisualVM Monitor: The information of CPU and memory can be collected with the help of VisualVM tool. VisualVM is a tool that provides a visual interface for profiling and viewing performance information about Java technology-based applications. The CPU profiler profiles the performance of the target program by recording CPU

[7] https://github.com/megagao/production_ssm

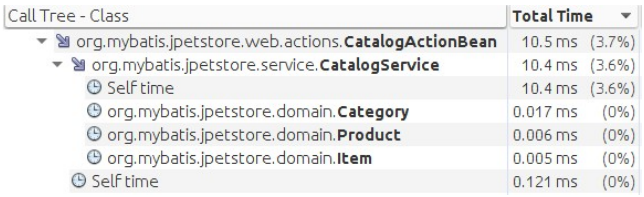

Fig. 4: Example of CPU profile

time of every method call and call relations between these calls. Fig 4 shows the performance record on the level of class, the method of CatalogService consumes 10.4 ms CPU time for the call from addItemToCart method.

Regarding the memory consumption, we choose to collect the Retained Size of an object [16]. Retained Size and Shallow Size are common concepts in the dump file analysis. The Shallow Size of an object is the allocated memory size to store the object itself without including its referenced objects; and the Retained Size of an object includes both its shallow size and that of its recursively referenced objects. We choose the Retained Size because it reflects more precisely on relation between objects.

B. Parameter Setting

In order to reflect the different impacts of invocations, we set $IStr_c$, $IStr_u$, $IStr_r$ with 100, 10, 1, respectively; and set $IFre_c$, $IFre_u$, $IFre_r$ with 1.03, 1.02 and 1.01, respectively. We set the respective values for the above parameters in accordance with the fact that we ran each test case at least 100 times. And it indicates that the weaker invocation types (e.g., READ messages) could have chance to weigh more than the stronger types (e.g., UPDATE messages) as the invocation times (frequency) reach some amounts.

We initially set all the parameters in this designed algorithm based on the general suggestions for standard genetic algorithms, and followed by calibration with multiple rounds of comparative pre-experiments performed on the motivating case of JPetStore. As a result, we obtained the parameter settings as follows: the crossover probability and mutation probability is set to 0.3 and 0.01 respectively; the population size is set to 100 individuals, and the maximum generation is limited to 200.

C. Experiments and Discussion

We performed three comparative experiments to evaluate the effectiveness of our proposed approach.

1) Experiment I: Manual Microservice Identification: We performed a manual identification of microservices from the legacy system. The authors have an average experiment of 7 years on system modelling and design. We identified the relatively independent functionalities from a perspective of end users by actually using the ERP system. As the source code can be easily obtained (although this artifact is not included in our proposed approach), we used a debug method to find out which classes are involved in responding to an identified functionality. After debugging

TABLE V: Results of Experiment I (the reference results)

No	Description	CC*	SC**
1	order management	C_1***	C_{30},C_{59},C_{87},C_{115}
2	custom management	C_2	C_{31},C_{60},C_{88}
3	product management	C_3	C_{32},C_{61},C_{89}
4	job management	C_4	C_{33},C_{62},C_{90},C_{116}
5	production planning management	C_5	C_{34},C_{63},C_{91},C_{117}
6	task management	C_6	C_{35},C_{64},C_{92}
7	device management	C_7	C_{36},C_{65},C_{93},C_{118}
8	device type management	C_8	C_{37},C_{66},C_{94},C_{119}
9	inspection record management	C_9	C_{38},C_{67},C_{95},C_{120}
10	malfunction record management	C_{10}	C_{39},C_{68},C_{96},C_{121}
11	maintenance record management	C_{11}	C_{40},C_{69},C_{97},C_{122}
12	technology management	C_{12}	C_{41},C_{70},C_{98}
13	technological requirement management	C_{13}	C_{42},C_{71},C_{99},C_{123}
14	technological plan management	C_{14}	C_{43},C_{72},C_{100},C_{124}
15	process management	C_{15}	C_{44},C_{73},C_{101}
16	material management	C_{16}	C_{45},C_{74},C_{102}
17	warehousing management	C_{17}	C_{46},C_{75},C_{103},C_{125}
18	consumption management	C_{18}	C_{47},C_{76},C_{104},C_{126}
19	substandard product management	C_{19}	C_{48},C_{77},C_{105},C_{127}
20	product measurement inspection	C_{20}	C_{49},C_{78},C_{106},C_{128}
21	product count inspection	C_{21}	C_{50},C_{79},C_{107},C_{129}
22	process measurement inspection	C_{22}	C_{51},C_{80},C_{108},C_{130}
23	process count inspection	C_{23}	C_{52},C_{81},C_{109},C_{131}
24	department management	C_{24}	C_{53},C_{82},C_{110}
25	employee management	C_{25}	C_{54},C_{83},C_{111},C_{132}
26	user management	C_{26}	C_{55},C_{84},C_{112},C_{133}
27	authorization service	C_{27}	C_{56},C_{85},C_{113},C_{134}, C_{138},C_{139},C_{140},C_{142}, C_{143},C_{144},C_{145}, C_{146},C_{141}
28	login service	C_{28}	C_{57}
29	file management	C_{29}	C_{58},C_{86},C_{114}

* CC is controller class;
** SC is subordinate class;
*** Definition of each C_i is shown in Appendix A.

for all the functionalities, we got the sets of classes. It is possible that there is overlap between some class sets. In this case, we manually analysed the code, and discussed whether the overlap classes should be attributed to one of the sets or attributed to a new create class set. Ultimately, we obtained the microservice candidates as shown in the Table V, where the description of functionalities and the related class sets of microservice candidates are included. Moreover, we manually identified a class in each class set as the controller class for the microservice candidate, thereby being convenient to act as the reference to contrast the results of the designed experiments II and III.

2) Experiment II: Microservice Identification with the Proposed Approach: After applying the proposed approach, we obtained the results of refactoring Production SSM into microservice candidates as shown in Table VI. Each row of the table indicates a microservice candidate, where a controller class, a set of subordinate classes, CPU and memory proportion are included.

3) Experiment III: Microservice Identification with Functional Metrics Only: We kept Equation (2) as the only objective for the genetic algorithm. The only kept objective function is related to the functional metrics. We implemented this experiment to compare with the state-of-the-art approach FoSCI [2], which is also an code-free approach based on execution log analysis. By applying

our proposed approach, we obtained the microservice candidates as also shown in Table VI.

4) Experiment IV: Functional Atom Identification with FoSCI: We implemented the approach of FoSCI [2] in the current experiment to generate functional atoms. Although the FoSCI approach has different opinion on the granularity of microservices, the functional atoms are essentially the same concept as the microservice in our approach. The results of functional decomposition are as shown in Table VI.

5) Discussion: We further summarized the results of the above experiments in Table VII, where the similarity indicates how the identified microservices candidates are similar as the reference results (shown in Experiment I). We evaluate the similarity by computing the ratio of the related subordinate classes in the same place as those in reference results.

We manually identified 29 microservice candidates in experiment I while the AMI approach automatically partitioned legacy system into 31 candidates in both experiments in Exp. II (with consideration of both functional and non-functional metrics) and Exp III (with consideration of only functional metrics). We obtained a least similarity of 84.65% between the manual and automatical approaches. Obviously, the majority part of the results of AMI approach is acceptable. We found the margin in the rest part of the results is caused by the ascription of five classes C_{27},C_{85},C_{146} and C_{29}, C_{86}. In Exp. I, we manually assigned the first three classes C_{27},C_{85},C_{146} to the $Mircoservice_{27}$ for authorization service while the last two classes C_{29}, C_{86} were assigned to the $Mircoservice_{28}$ for file uploading. By contrast, the Exp. II and Exp. III automatically identified the five classes as controller classes. The manually obtained results should be better than the automatically obtained ones. However, it is easy to investigate some working hours to revise the latter results manually. Even so, our tallied results showed that about 40 working hours(about one week) were used for debugging, analyzing and decomposing the legacy system in experiment I while only 8 working hours in total to analyze and revise the result of AMI. We can carefully draw out a conclusion that the AMI approach is usefulness and takes an advantage over the manual approach on cost-efficiency.

Moreover, 20 functional atoms are generated by the FoSCI approach as shown in Table VI. The comparatively experimental results in Table VII provide an evidence that our proposed approach outperforms the FoSCI in two ways: Firstly, when taking only functional decomposition into consideration, our approach in experiment III got higher similarity than that in the FoSCI approach . The reason is that, when evaluating the functional relation between each pair of classes, we consider not only how frequently one class is related to another but also what kind of relation (i.e., creation, updating, and reading) between them. By contrast, the FoSCI approach clusters the related classes into a functional atom by only measuring how frequently the classes appeared in the same traces, without differentiating various types of relation. Obviously, a creation invocation indicates a stronger relation than an updating invocation, and a much stronger relation than a reading invocation. Ignoring this difference may accordingly contribute to a deviation in functional decomposition of a system. Secondly, when bringing non-functional metrics into account, our approach shows an obvious advantage in load balance over the FoSCI. That is because our approach jointly uses both functional and non-functional metrics in automatic microservice identification.

Additionally, we observe that the Exp. II has a decrease of 4.4% in similarity by comparing with the Exp. III. At the same time, we also observe a better load balance appearing in Exp. II than Exp. III. By further analyzing the results in the both two experiments, we found that the classes for file uploading (e.g., SC_{58} and SC_{114}) consume relatively more resources in CPU and memory. That is why the AMI approach with considering both functional and non-functional objectives re-located them in order to achieve a better balance. It indicates that the non-functional metrics do have impact (effectiveness) on the results of microservice identification, and that provides a choice to users of this AMI approach if they want to properly take into consideration of load balance in development and operation of their microservices.

Ultimately, we have to admit that the AMI approach still has a large space to improve so as to better the similarity with the manual reference results.

V. Related Work

Microservice identification is essentially a process of suitably partitioning an overall system into serval small, functionally independent, manageable parts. There are two main categories on the topic of microservice identification, that is, identifying microservices from greenfield and from legacy monolithic systems.

These approaches on microservice identification from greenfield take user requirements as input, and split the planned overall systems based on different criteria by analyzing the requirement specifications. For example, Ahmadvand and Ibrahim [17] decompose a system into microservices based on the criteria of functionalities, security and scalability. Michael et al. [18] implement a tool that supports structured service decomposition with graph cutting, where the 16 different coupling criteria are used in the method by summarizing both academia suggestions and industrial practices.

Despite of building microservice based systems from greenfield, much more cases are related to migration from legacy monoliths into a set of fine-designed microservices. Majority of the existing research on this category is based on the analysis of software artifacts, such as documentations, source codes, version control history in

TABLE VI: Experiment II , III and IV result

No	Exp II			Exp III			Exp IV		
	CC	SC	CPU(%) /Mem(%)	CC	SC	CPU(%) /Mem(%)	CC	Functional Atom	CPU(%) /Mem(%)
1	C_1	C_{30},C_{59},C_{115} ,C_{87}	2.32/7.48	C_1	C_{30},C_{59},C_{115} ,C_{87}	2.32/7.48	C_1,C_3	C_{30},C_{32},C_{59},C_{115},C_{61},C_{89},C_{87}	5.16/14.30
2	C_2	C_{60},C_{31},C_{88}	2.64/7.45	C_2	C_{60},C_{31},C_{88}	2.64/7.45	C_2,C_7, C_5,C_{12}	C_{36},C_{60},C_{34},C_{41},C_{31},C_{88},C_{65},C_{93},C_{63}, C_{91},C_{104},C_{70},C_{98},C_{114},C_{118},C_{117}	10.39/20.87
3	C_3	C_{148},C_{32},C_{61} ,C_{89}	2.85/6.88	C_3	C_{32},C_{61},C_{89}	1.85/6.82	—	—	—/—
4	C_4	C_{33},C_{62},C_{90} ,C_{116}	2.51/2.29	C_4	C_{136},C_{33},C_{113} ,C_{62},C_{90},C_{143} ,C_{116}	9.90/2.84	C_4	C_{33},C_{62},C_{90},C_{116}	2.51/2.29
5	C_5	C_{136},C_{34},C_{63} ,C_{91},C_{117}	4.25/2.63	C_5	C_{34},C_{63},C_{91} ,C_{117}	2.23/1.58	—	—	—/—
6	C_6	C_{35},C_{64},C_{92}	2.20/0.52	C_6	C_{35},C_{64},C_{92}	2.20/0.52	C_6	C_{35},C_{64},C_{99}	2.19/0.52
7	C_7	C_{137},C_{36},C_{65} ,C_{93},C_{73},C_{118}	3.37/7.25	C_7	C_{135},C_{36},C_{65} ,C_{93},C_{118}	1.37/6.42	—	—	—/—
8	C_8	C_{37},C_{66},C_{94} ,C_{119}	4.01/3.32	C_8	C_{137},C_{37},C_{66} ,C_{94},C_{119}	7.01/3.38	C_8,C_{86}	C_{37},C_{97},C_{66},C_{119}	6.00/3.32
9	C_9	C_{38},C_{67},C_{95} ,C_{120}	2.99/2.46	C_9	C_{38},C_{67},C_{95} ,C_{120}	2.99/2.46	C_9	C_{38},C_{67},C_{96},C_{120}	2.98/2.46
10	C_{10}	C_{39},C_{68},C_{96} ,C_{121}	2.84/3.50	C_{10}	C_{39},C_{68},C_{96} ,C_{121}	2.84/3.50	C_{10}	C_{39},C_{95},C_{68},C_{121}	2.84/3.50
11	C_{11}	C_{40},C_{69},C_{97} ,C_{122}	2.44/1.78	C_{11}	C_{40},C_{69},C_{97} ,C_{122}	2.44/1.78	C_{11}	C_{40},C_{69},C_{94},C_{122}	2.44/1.78
12	C_{12}	C_{41},C_{71},C_{98}	3.15/4.61	C_{12}	C_{41},C_{70},C_{98}	4.15/3.61	—	—	—/—
13	C_{13}	C_{42},C_{70},C_{99} ,C_{123}	2.67/2.06	C_{13}	C_{42},C_{71},C_{99} ,C_{123},C_{147}	2.67/5.10	C_{13}	C_{42},C_{92},C_{71},C_{123}	2.67/2.06
14	C_{14}	C_{43},C_{72},C_{124} ,C_{142},C_{100}	2.63/3.24	C_{14}	C_{43},C_{72},C_{124} ,C_{100}	1.62/2.72	C_{14}	C_{43},C_{72},C_{124},C_{100}	2.62/2.72
15	C_{15}	C_{44},C_{101},C_{139}	3.25/2.26	C_{15}	C_{44},C_{73},C_{101}	3.26/2.06	C_{15}	C_{44},C_{73},C_{101}	3.25/2.06
16	C_{16}	C_{45},C_{74},C_{102}	2.66/3.96	C_{16}	C_{45},C_{74},C_{102}	2.66/3.96	—	—	—/—
17	C_{17}	C_{46},C_{75},C_{103} ,C_{125},C_{127}	2.33/0.77	C_{17}	C_{46},C_{75},C_{103} ,C_{125}	3.33/0.77	—	—	—/—
18	C_{18}	C_{47},C_{76},C_{104} ,C_{126}	2.31/0.92	C_{18}	C_{47},C_{76},C_{104} ,C_{126}	2.31/0.92	C_{18},C_{17}, C_{16}	C_{47},C_{45},C_{46},C_{107},C_{74},C_{76},C_{102},C_{75},C_{125}	7.28/4.79
19	C_{19}	C_{48},C_{77},C_{105}	2.32/1.66	C_{19}	C_{48},C_{77},C_{105} ,C_{127}	1.32/1.66	C_{19}	C_{48},C_{77},C_{105},C_{127}	2.32/1.66
20	C_{20}	C_{49},C_{78},C_{106} ,C_{128}	2.27/2.12	C_{20}	C_{148},C_{49},C_{78} ,C_{106},C_{128}	3.27/2.18	C_{20}	C_{49},C_{78},C_{106},C_{128}	2.27/2.12
21	C_{21}	C_{58},C_{79},C_{107}	16.38/1.49	C_{21}	C_{50},C_{79},C_{107} ,C_{129}	8.72/2.35	C_{21}	C_{50},C_{79},C_{103},C_{129}	2.72/2.35
22	C_{22}	C_{51},C_{80},C_{108} ,C_{130}	2.48/1.43	C_{22}	C_{51},C_{80},C_{108} ,C_{130}	2.48/1.43	C_{22}	C_{51},C_{80},C_{126},C_{130}	2.48/1.49
23	C_{23}	C_{52},C_{81},C_{109}	2.00/1.06	C_{23}	C_{52},C_{81},C_{109} ,C_{131}	2.00/2.35	C_{23}	C_{52},C_{81},C_{109},C_{131}	2.00/2.35
24	C_{24}	C_{53},C_{82},C_{110} ,C_{131}	2.76/3.50	C_{24}	C_{53},C_{82},C_{110}	2.75/2.21	—	—	—/—
25	C_{25}	C_{54},C_{83},C_{111} ,C_{143},C_{132},C_{147}	2.76/3.67	C_{25}	C_{54},C_{83},C_{111} ,C_{132}	2.69/2.21	C_{25},C_{24}	C_{53},C_{54},C_{82},C_{110},C_{83},C_{111},C_{132}	5.44/4.41
26	C_{26}	C_{55},C_{84},C_{112} ,C_{133}	2.72/3.12	C_{26}	C_{55},C_{84},C_{112} ,C_{133}	2.72/3.12	C_{27},C_{85}, C_{146},C_{26}	C_{57},C_{28},C_{56},C_{58},C_{113},C_{55},C_{144},C_{145}, C_{140},C_{141},C_{142},C_{143},C_{84},C_{112},C_{134},C_{138},C_{139}	31.39/22.61
27	C_{27}	C_{56},C_{144},C_{145}	3.71/0.17	C_{27}	C_{56},C_{145},C_{142}	0.38/1.43	—	—	—/—
28	C_{29}	C_{28},C_{135},C_{141}	3.03/2.35	C_{29}	C_{58},C_{134},C_{138}	16.28/0.06	—	—	—/—
29	C_{85}	C_{134},C_{138}	3.04/1.12	C_{85}	C_{28}	0.05/1.18	C_{29}	C_{133},C_{108}	0.01/0.06
30	C_{86}	C_{50},C_{140}	2.70/0.06	C_{86}	C_{57},C_{114},C_{144} ,C_{141}	0.54/15.42	—	—	—/—
31	C_{146}	C_{57},C_{114},C_{113}	2.42/14.85	C_{146}	C_{140},C_{139}	0.01/1.03	—	—	—/—

TABLE VII: Measurement result

Strategy	Similarity	CPU Balance	Memory Balance
Exp II	84.65%	0.0015	0.0008
Exp III	89.05%	0.0020	0.0009
Exp IV	87.50%	0.0044	0.0042

code repositories, etc [3] [19] [20] [21] [22] [23] [24]. The relation between each pair of classes or objects is widely evaluated to achieve a properly functions splitting. For example, Adjoyan et al. [3] propose an automatic service identification from source code by optimally clustering classes based on the service quality metrics. Eski et al. [19] use evolutionary and static code coupling information as the criteria where the relation between classes is evaluated, and perform a service identification by graph based classes clustering. Mazlami et al. [20] present a microservice extraction model by analyzing the source codes and the related version control system, where the class dependencies, business domain semantics, and code version information are jointly considered and quantified to splitting legacy systems. Baresi et al. [21] grouped the system API interfaces as service candidates based on the semantic similarity of foreseen/available functionality described through OpenAPI specifications. Levcovitz et al. [22] present a technique that groups database tables into several subsystems where each represents a business

domain of the organization, and then clustering the classes into microservices that access the same subsystem. Selmadji et al. [23] propose a quality function that measures both the structural validity and data autonomy of classes, then they design a hierarchical agglomerative clustering algorithm to optimally identify service candidates. Escobar et al. [24] decomposes Enterprise Java Bean (EJB) into service candidates through the relations between session bean and entity bean by analyzing the source code.

The above literature is mainly based on the underlying assumption of availability of sufficient legacy software artifacts, e.g., documentation, analysis and design models, and source code. That is not realistic in industrial practices. Accordingly, many researchers have engaged in exploring some document-free or code-free approaches. For example, Wuxia Jin et al. [2] present an execution log based approach without any other software artifacts. They split the legacy monolith into service candidates from a functional perspective. Based on the formal definition of system, Alwis et al. [5] propose a function splitting approach that identifies service candidates based on object subtypes and common execution fragments across software. However, only function related factors are considered in their methods, which may result in a gap between identified microservices and expected ones.

Therefore, our proposed approach aims to combined use both execution and performance logs, and perform automated microservice identification for monolithic legacy system from both the functional and non-functional perspectives. Compared with the approaches of document or code based static analysis [3] [19] [20] [21] [23] [24], executable logs that reflect runtime system business processes are considered to be a more reasonable source of data for service extraction [2]. Furthermore, comparing with these execution logs based methods [2] [5], our approach considers both invocation frequency and invocation strength in functional relation evaluation, along with bringing non-functional metrics into account. All of these have contributed to a higher accuracy in function decomposition and better performance in load balance. Our paper is the first work that jointly considering functional and non-functional metrics in automated microservices identification to the best of our knowledge.

VI. Conclusion and Future Work

This paper tackles two weaknesses in the previous literature: (1)impractical assumption of the availability of sufficient documentation, analysis and design models, or source code in legacy monolithic system migration, and (2) mainly uses single-aspect metrics (e.g., from a function-splitting perspective) to identify candidate microservices. We propose an AMI approach that extracts microservices from the execution and performance logs without providing documentation, models or source codes, while taking both the functional and non-functional metrics into considerations. In this approach, we design an AMI algorithm to identify the controller objects as the key objects to converge their strongly related subordinate objects. A quantified O-O relation matrix and a quantified C-C relation matrix are successively created to evaluate the relation of each pair of CO and SO (their related classes, respectively). A genetic algorithm is also designed to optimise the multi-objective of high-cohesion-low-coupling and load balance. A legacy ERP system is used as a case to demonstrate the applicability of our proposed approach. As a part of our future work, we plan to improve the approach by better the accuracy of AMI and introducing more metrics, as well as applying to lager and more complex cases.

Acknowledgments

This paper is funded in part by the Special Foundation for Basic Science and Frontier Technology Research Program of Chongqing (cstc2017jcyjAX0295), the Capacity Development Grant of Southwest University (SWU116007), and the National Natural Science Foundation of China (61732019, 61672435, 61811530327). We would like to thank Professor Zhiming Liu for the enlightening discussions that form the main idea of this paper. We also want to thank the anonymous reviewers for their constructive suggestions that help us significantly improve the manuscript.

References

[1] "Microservices," https://martinfowler.com/articles/microservices.html.

[2] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service Candidate Identification from Monolithic Systems based on Execution Traces," IEEE Transactions on Software Engineering, pp. 1–1, 2019.

[3] S. Adjoyan, A.-D. Seriai, and A. Shatnawi, "Service Identification Based on Quality Metrics Object - Oriented Legacy System Migration Towards SOA," in SEKE: Software Engineering and Knowledge Engineering. Knowledge Systems Institute Graduate School, Jul. 2014, pp. 1–6.

[4] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, "Identifying Microservices Using Functional Decomposition," in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2018, vol. 10998 LNCS, pp. 50–65. [Online]. Available: http://link.springer.com/10.1007/978-3-319-99933-3_4

[5] A. A. C. De Alwis, A. Barros, A. Polyvyanyy, and C. Fidge, "Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems," in Service-Oriented Computing, C. Pahl, M. Vukovic, J. Yin, and Q. Yu, Eds. Cham: Springer International Publishing, 2018, vol. 11236, pp. 37–53.

[6] C. Larman, Applying UML and Patterns, 3rd ed. Prentice Hall.

[7] D. Li, X. Li, Z. Liu, and V. Stolz, "Interactive transformations from object-oriented models to component-based models," pp. 97–114, 2011.

[8] N. Tiwari and K. C. Nair, "Performance extrapolation that uses industry benchmarks with performance models," in Proceedings of the 2010 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '10), pp. 301–305.

[9] F. Fittkau and W. Hasselbring, "Elastic Application-Level Monitoring for Large Software Landscapes in the Cloud," in Service Oriented and Cloud Computing, ser. Lecture Notes in Computer Science, S. Dustdar, F. Leymann, and M. Villari, Eds. Springer International Publishing, pp. 80–94.

[10] M. Harman and B. F. Jones, "Search-based software engineering," p. 7.

[11] A. Ramírez, J. R. Romero, and S. Ventura, "A survey of many-objective optimisation in search-based software engineering," vol. 149, pp. 382–395. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121218302759

[12] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[13] L. Carvalho, A. Garcia, W. K. G. Assun??o, R. de Mello, and M. J. de Lima, "Analysis of the criteria adopted in industry to extract microservices," in Proceedings of the Joint 7th International Workshop on Conducting Empirical Studies in Industry and 6th International Workshop on Software Engineering Research and Industrial Practice, ser. CESSER-IP '19. IEEE Press, pp. 22–29. [Online]. Available: https://doi.org/10.1109/CESSER-IP.2019.00012

[14] "Multitier architecture - Wikipedia," https://en.wikipedia.org/wiki/Multitier_architecture.

[15] M. Richards, Software Architecture Patterns. O'Reilly Media, Inc.

[16] YourKit Java Profiler Help - Shallow and retained sizes. [Online]. Available: https://www.yourkit.com/docs/java/help/sizes.jsp

[17] M. Ahmadvand and A. Ibrahim, "Requirements Reconciliation for Scalable and Secure Microservice (De)composition," in 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW), Sep. 2016, pp. 68–73.

[18] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service Cutter: A Systematic Approach to Service Decomposition," in Service-Oriented and Cloud Computing, ser. Lecture Notes in Computer Science. Springer, Cham, Sep. 2016, pp. 185–200.

[19] S. Eski and F. Buzluca, "An Automatic Extraction Approach: Transition to Microservices Architecture from Monolithic Application," in Proceedings of the 19th International Conference on Agile Software Development: Companion, ser. XP '18. New York, NY, USA: ACM, 2018, pp. 25:1–25:6.

[20] G. Mazlami, J. Cito, and P. Leitner, "Extraction of Microservices from Monolithic Software Architectures," in 2017 IEEE International Conference on Web Services (ICWS), Jun. 2017, pp. 524–531.

[21] L. Baresi, M. Garriga, and A. De Renzis, "Microservices Identification Through Interface Analysis," in Service-Oriented and Cloud Computing, ser. Lecture Notes in Computer Science, F. De Paoli, S. Schulte, and E. Broch Johnsen, Eds. Springer International Publishing, 2017, pp. 19–33.

[22] A. Levcovitz, R. Terra, and M. T. Valente, "Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems," arXiv:1605.03175 [cs], May 2016.

[23] A. Selmadji, A.-D. Seriai, H. L. Bouziane, C. Dony, and R. O. Mahamane, "Re-architecting OO Software into Microservices," in Service-Oriented and Cloud Computing, ser. Lecture Notes in Computer Science, K. Kritikos, P. Plebani, and F. de Paoli, Eds. Springer International Publishing, 2018, pp. 65–73.

[24] D. Escobar, D. Cardenas, R. Amarillo, E. Castro, K. Garces, C. Parra, and R. Casallas, "Towards the understanding and evolution of monolithic applications as microservices," in 2016 XLII Latin American Computing Conference (CLEI). Valparaíso, Chile: IEEE, Oct. 2016, pp. 1–11.

Appendix A
The Classes in Production SSM

No.	Class	No.	Class
C_1	OrderServiceImpl	C_{75}	MaterialReceive
C_2	CustomServiceImpl	C_{76}	MaterialConsume
C_3	ProductServiceImpl	C_{77}	UnqualifyApply
C_4	WorkServiceImpl	C_{78}	FinalMeasuretCheck
C_5	ManufactureServiceImpl	C_{79}	FinalCountCheck
C_6	TaskServiceImpl	C_{80}	ProcessMeasureCheck
C_7	DeviceServiceImpl	C_{81}	ProcessCountCheck
C_8	DeviceTypeServiceImpl	C_{82}	Department
C_9	DeviceCheckServiceImpl	C_{83}	Employee
C_{10}	DeviceFaultServiceImpl	C_{84}	SysUser
C_{11}	DeviceMaintainServiceImpl	C_{85}	RoleServiceImpl
C_{12}	TechnologyServiceImpl	C_{86}	PictureServiceImpl
C_{13}	TechnologyRequirement ServiceImpl	C_{87}	COrderVO
C_{14}	TechnologyPlanServiceImpl	C_{88}	CustomExample
C_{15}	ProcessServiceImpl	C_{89}	ProductExample
C_{16}	MaterialServiceImpl	C_{90}	WorkExample
C_{17}	MaterialReceiveServiceImpl	C_{91}	ManufactureExample
C_{18}	MaterialConsumeServiceImpl	C_{92}	TaskExample
C_{19}	UnqualifyServiceImpl	C_{93}	DeviceExample
C_{20}	MeasureServiceImpl	C_{94}	DeviceTypeExample
C_{21}	FCountCheckServiceImpl	C_{95}	DeviceCheckExample
C_{22}	PMeasureCheckServiceImpl	C_{96}	DeviceFaultExample
C_{23}	PCountCheckServiceImpl	C_{97}	DeviceMaintainExample
C_{24}	DepartmentServiceImpl	C_{98}	TechnologyExample
C_{25}	EmployeeServiceImpl	C_{99}	TechnologyRequirementExample
C_{26}	UserServiceImpl	C_{100}	TechnologyPlanVO
C_{27}	PermissionServiceImpl	C_{101}	ProcessExample
C_{28}	LoginController	C_{102}	MaterialExample
C_{29}	FileServiceImpl	C_{103}	MaterialReceiveExample
C_{30}	OrderController	C_{104}	MaterialConsumeExample
C_{31}	Custom	C_{105}	UnqualifyApplyExample
C_{32}	ProductController	C_{106}	FinalMeasuretCheckExample
C_{33}	WorkController	C_{107}	FinalCountCheckExample
C_{34}	ManufactureController	C_{108}	ProcessMeasureCheckExample
C_{35}	TaskController	C_{109}	ProcessCountCheckExample
C_{36}	DeviceListController	C_{110}	DepartmentExample
C_{37}	DeviceTypeController	C_{111}	EmployeeExample
C_{38}	DeviceCheckController	C_{112}	SysUserExample
C_{39}	DeviceFaultController	C_{113}	RoleController
C_{40}	DeviceMaintainController	C_{114}	PictureController
C_{41}	TechnologyController	C_{115}	COrderExample
C_{42}	TechnologyRequirement Controller	C_{116}	WorkVO
C_{43}	TechnologyPlanController	C_{117}	ManufactureVO
C_{44}	ProcessController	C_{118}	DeviceVO
C_{45}	MaterialController	C_{119}	DeviceTypeVO
C_{46}	MaterialReceiveController	C_{120}	DeviceCheckVO
C_{47}	MaterialConsumeController	C_{121}	DeviceFaultVO
C_{48}	UnqualifyApplyController	C_{122}	DeviceMaintainVO
C_{49}	FMeasurementController	C_{123}	TechnologyRequirementVO
C_{50}	FCountCheckController	C_{124}	TechnologyPlanExample
C_{51}	PMeasureCheckController	C_{125}	MaterialReceiveVO
C_{52}	PCountCheckController	C_{126}	MaterialConsumeVO
C_{53}	DepartmentController	C_{127}	UnqualifyApplyVO
C_{54}	EmployeeController	C_{128}	FinalMeasuretCheckVO
C_{55}	UserController	C_{129}	FinalCountCheckVO
C_{56}	PermissionController	C_{130}	ProcessMeasureCheckVO
C_{57}	FirstController	C_{131}	ProcessCountCheckVO
C_{58}	FileController	C_{132}	EmployeeVO
C_{59}	COrder	C_{133}	UserVO
C_{60}	CustomController	C_{134}	SysUserRole
C_{61}	Product	C_{135}	DeviceController
C_{62}	Work	C_{136}	AuthorityJudgeController
C_{63}	Manufacture	C_{137}	StringTrimConverter
C_{64}	Task	C_{138}	SysUserRoleExample
C_{65}	Device	C_{139}	RoleVO
C_{66}	DeviceType	C_{140}	SysRole
C_{67}	DeviceCheck	C_{141}	SysRoleExample
C_{68}	DeviceFault	C_{142}	SysRolePermission
C_{69}	DeviceMaintain	C_{143}	SysRolePermissionExample
C_{70}	Technology	C_{144}	SysPermission
C_{71}	TechnologyRequirement	C_{145}	SysPermissionExample
C_{72}	TechnologyPlan	C_{146}	SysServiceImpl
C_{73}	Process	C_{147}	CustomRealm
C_{74}	Material	C_{148}	CustomDateConverter