The Journal of Systems and Software 157 (2019) 110380

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A dataflow-driven approach to identifying microservices from monolithic applications

Shanshan Li[a], He Zhang[a], Zijia Jia[a], Zheng Li[b], Cheng Zhang[c,*], Jiaqi Li[a], Qiuya Gao[a], Jidong Ge[a], Zhihao Shan[d]

[a] State Key Laboratory of Novel Software Technology, Software Institute, Nanjing University, 22 Hankou Road, Gulou District, Nanjing City, 210093, Jiangsu Province, China
[b] Department of Computer Science, University of Concepción, Edmundo Larenas 219, Concepción 4070409, Chile
[c] School of Computer Science & Technology, Anhui University, 111 Jiulong Road, Hefei City, Anhui Province, 230601, China
[d] Tencent Building, Zhongquyi Road, Hi-tech Park, Shenzhen, Guangdong, 518057, China

ARTICLE INFO

Article history:
Received 1 June 2018
Revised 27 February 2019
Accepted 5 July 2019
Available online 5 July 2019

Keywords:
Software engineering
Microservices
Monolith
Decomposition
Data flow
Business logic(s)

ABSTRACT

Microservices architecture emphasizes employing multiple small-scale and independently deployable microservices, rather than encapsulating all function capabilities into one monolith. Correspondingly, microservice-oriented decomposition, which has been identified to be an extremely challenging task, plays a crucial and prerequisite role in developing microservice-based systems. To address the challenges in such a task, we propose a dataflow-driven semi-automatic decomposition approach. In particular, a four-step decomposition procedure is defined: (1) conduct the business requirement analysis to generate use case and business logic specification; (2) construct the fine-grained Data Flow Diagrams (DFD) and the process-datastore version of DFD (DFD_{PS}) representing the business logics; (3) extract the dependencies between processes and datastores into decomposable sentence sets; and (4) identify candidate microservices by clustering processes and their closely related datastores into individual modules from the decomposable sentence sets. To validate this microservice-oriented decomposition approach, we performed a case study on Cargo Tracking System that is a typical case decomposed by other microservices identification methods (Service Cutter and API Analysis), and made comparisons in terms of specific coupling and cohesion metrics. The results show that the proposed dataflow-driven decomposition approach can recommend microservice candidates with sound coupling and cohesion through a rigorous and easy-to-operate implementation with semi-automatic support.

1. Introduction

Monolithic architecture (MLA) (Richardson, 2018b) is the traditional high-level structure for constructing software, which used to be regarded as the widely accepted and employed software architecture by most Internet services including some Internet giants such as Netflix, Amazon and eBay, and it advocates encapsulating all functions in one single application. Less complicated monolithic applications have their own strengths, for instance, simple to develop, test, and deploy. Nevertheless, with time elapses a successful application always grows in size and eventually becomes a monstrous monolith after a few years. Once this happens, disadvantages of the monolithic architecture may outweigh its advantages, for example, overwhelmingly complex and incomprehensible code base of the monolith may hold back bug fixing and feature addition; the sheer size of the monolith can slow down the development and become an obstacle to continuous deployment because of the longer start-up time.

Microservice architecture (MSA), emerging from Agile developer communities, has become an alternative way in recent years that may effectively tackle the challenges of monolithic architecture, by decomposing the monolith into a set of small services and making them communicate with each other through light weight mechanisms, e.g., RESTful API (Fowler and Lewis, 2014).

The advantages of microservices are commonly accepted both in academia and industry, e.g., maintainability, reusability, scalability, availability and automated deployment (Francesco et al., 2017; Alshuqayran et al., 2016; Zimmermann, 2017; Richardson, 2015).

* Corresponding author.
E-mail addresses: stu.shanshan.li@gmail.com (S. Li), hezhang@nju.edu.cn (H. Zhang), stu_zijiajia@163.com (Z. Jia), imlizheng@gmail.com (Z. Li), cheng.zhang@ahu.edu.cn (C. Zhang), jachly3@gmail.com (J. Li), g_qiuya@foxmail.com (Q. Gao), gjdnju@163.com (J. Ge), cycle44@gmail.com (Z. Shan).

https://doi.org/10.1016/j.jss.2019.07.008
0164-1212/

Netflix, Amazon and eBay all have migrated their applications from monolithic architecture to microservice architecture, so as to benefit from many of its advantages.

Nevertheless, microservice architecture is not a panacea and it also brings many issues to be addressed (Jamshidi et al., 2018; Soldani et al., 2018). The most important one is how to effectively decompose a monolithic application into a suite of small microservices, for the reason that the decomposition process is often implemented manually (Kecskemeti et al., 2016). One of the critical challenges in decomposition is to identify the appropriate partitions of a monolithic system, because the microservice architecture with different granularity can significantly affects many quality attributes of the system, e.g., performance and testability (Heinrich et al., 2017). However, there is a lack of a systematic approach to determining and recommending microservice boundaries through rigorous approaches. Therefore, microservice-oriented design is usually performed in an intuitive manner and largely based on the architect's experience. The inappropriate partition of services may be costly (Newman, 2015). In addition, the absence of sound evaluation methodology exacerbates the challenges of microservice-oriented decomposition.

To address the above issues, we propose a dataflow-driven approach that realizes a systematic and easy-to-understand microservice-oriented decomposition. Correspondingly, this work makes four-fold contributions: first, this dataflow-driven approach essentially provides a systematic methodology for microservice-oriented decomposition; second, compared to the manual implementations (Kecskemeti et al., 2016), the algorithm supported semi-automated process effectively manages the uncertainty and effort on the decomposition practices; third, the typical case of Cargo Tracking System can act as an exemplar to demonstrate the use and effect of the microservice-oriented decomposition approach; finally, the essential metrics and the tool used for the assessment of the proposed approach suggests the evaluation solution for microservice-oriented decomposition with tool support.

This paper is an extended version of the conference paper (Chen et al., 2017) published at Asia-Pacific Software Engineering Conference (APSEC) in 2017. In comparison with the conference version, this paper is extended and updated in the following aspects: 1) we incorporate the construction of DFDs with different granularities (Level 0 DFD and Level 1 DFD), which enhances the quality of DFDs generated, especially for complex systems; 2) compared to the two business logic scenarios evaluated in the conference version, a more typical and complex case, Cargo Tracking System, is applied to evaluate the improved decomposition approach in this version; 3) we use essential coupling and cohesion metrics for evaluating the results of the decomposition approach with the comparison against other related studies.

The rest of this paper is organized as follows. Section 2 briefly summarizes the related work. Section 3 introduces the theoretical foundation of the dataflow-driven approach for microservice-oriented decomposition, including definitions, decomposition rules, and processes. In Section 4, we elaborate our case study on microservice-oriented decomposition by employing the dataflow-driven approach. The evaluation of the proposed decomposition approach and some limitations are discussed in Section 5 and 6 respectively. In Section 7, we draw conclusions and suggest potential future work directions.

2. Related work

2.1. Microservice-oriented decomposition

Microservice-oriented decomposition is a prerequisite in migration to microservice architecture. In the three dimensions to achieve scalability (Abbott and Fisher, 2009), the *Y*-axis of the scalable cube represents the decomposition process, i.e. spliting the application into small chunks to achieve higher scalability. More than scalability in theory, decomposing monolithic applications into microservices may result in benefits with several other quality attributes. however, the commonly manual and complex decomposition process (Kecskemeti et al., 2016) prevents the practices from achieving those benefits. Consequently, it is of great importance to effective tackle this challenge and realize the microservice-oriented decomposition in an efficient manner.

In the past three years, some researchers have devoted to addressing the problems of partitioning the application into microservices. Richardson (2018a) introduced four decomposition strategies, i.e. "decompose by business capability", "decompose by domain-driven design sub domain", "decompose by verb or use case" and "decompose by nouns or resources". To the best of our knowledge, the former two are the most abstract patterns which require human involvement and a decision making (Evans, 2002); while the latter two are easier to realize automation as long as criteria have been predefined. Moreover, as mentioned by Vresk and Cavrak (2016), an application might use a combination of verb-based and noun-based decomposition. Disappointingly, this method was not implemented in their study. Hassan et al. (2017) offered an architecture-centric modeling concept for microservitization, without mapping the microservices constituents in models to concrete microservices source code and verifying its feasibility. Gysel et al. (2016) proposed an service decomposition method on the base of 16 coupling criteria extracted from literature and industry experience. Clustering algorithms are applied to decompose an undirected, weighted graph, which is transformed from Software System Artifacts (SSAs). Yet Service Cutter 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. Some processes of the decomposition approach, especially score the edge of the aforementioned graph, lack objectivity. Besides, the transformation from SSAs to predefined, structured input increases the complexity in microservice-oriented decomposition.

Compared with the conference paper, this paper also finds another two studies related to microservice-oriented decomposition. The first one was conducted by Baresi et al. (2017), who proposed an automated process for finding microservices candidates through API specifications analysis. This approach relies on well-defined and described interfaces that provide meaningful names, and follow programming naming conventions such as camel casing and hyphenation. Unfortunately, this is not always the case and some situations are difficult to cope with (e.g., identifiers like op1, param or response). Mazlami et al. (2017) presented a formal microservice extraction model to allow algorithmic recommendation of microservice candidates in a refactoring and migration scenario. In order to circumvent the database decomposition challenge the extraction model assumes the presence of an ORM system represents data model entities as classes which are treated just like any ordinary class by the extraction algorithm. However, microservices in practice 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.

On the contrary, the microservice-oriented decomposition approach proposed in our work is driven by data flow diagrams from requirement and business logic analysis. Based on the objective operations and data extracted from the real-world business logics, our decomposition approach can deliver more rational, objective, and easy-to-understand results including the database design for each microservice.

2.2. Beyond microservice-oriented decomposition

Service-oriented (de)composition in traditional SOA has been a hot topic for decades (Mardukhi et al., 2013; Sun and Zhao, 2012), with two differences to decomposition under microservices architecture. One is that services are coarse-grained in SOA, but usually more fine-grained in microservices architecture. The other one is that decomposition in SOA of10 aims at selecting the optimal composed service from all possible service combinations regarding some quality requirements, which is a bottom-up process; while in MSA the decomposition procedure may cover the top-down partition first and then the bottom-up integration.

As a top-down method, Data Flow Diagram (DFD) (Constantine and Yourdon, 1978) is a good choice for representing monolithic applications from the perspective of business processes (Zhao et al., 2009). Adler (1988) presented an algebra formalizing the decomposition procedure of data flow diagrams, however, it has some drawbacks like inefficiency and no guarantee on finding a good decomposition. Arndt and Guercio (1992) improved Adlers work by replenishing new criteria and implementing their own algorithm for the decomposition of data flow diagrams. Both of these two studies equally lack assessments on decomposition result except for the intuitive notion.

Compared with the aforementioned studies, we transformed the traditional DFD to a process-datastore and detailed version, rather than other SSAs (Gysel et al., 2016) that are difficult to be transformed objectively for decomposition. Moreover, we designed algorithms to support the two phases automation in our decomposition approach: (1) condensing the process-datastore version of DFDs to decomposable DFDs and (2) identifying microservice candidates from the decomposable DFD. We also conducted one typical case study of microservice-oriented decomposition, and demonstrated the effectiveness of our decomposition approach by comparison against another service-oriented decomposition tool.

3. Dataflow-driven decomposition approach

To address the challenge and reduce the complexity in microservice-oriented decomposition, we developed a semi-automatic approach to break business logics into microservice candidates based on the dataflow diagram decomposition. It is clear that DFD plays a fundamental role in our microservice-oriented decomposition work. By graphically displaying the flow of data within an information system, DFD has widely been used as one of the preliminary tools of requirement analysis in software engineering. Therefore, here we start from introducing relevant definitions about DFD theoretically, followed by explaining the decomposition rules and process of microservices using DFD defined in our study.

3.1. Theoretical foundation

3.1.1. Elements of DFD

From the data perspective, a DFD, which can be generated from business logics of a system, describes the flow of data through business functions or processes (Rosenblatt, 2013). In other words, it illustrates how data is processed by a business function or an operation of a system in terms of inputs and outputs. A DFD is composed of four basic elements, namely Process, Data Store, Data Flow and External Entity.

Furthermore, the traditional DFD mainly has two types of visual representations, which are proposed by Constantine and Yourdon (1978) and Gane (1978) respectively. However, there is no obvious distinction between these two visualizations, except the representation of processes. In our work, the former type from Constantine et al. is selected as the representation of DFDs.

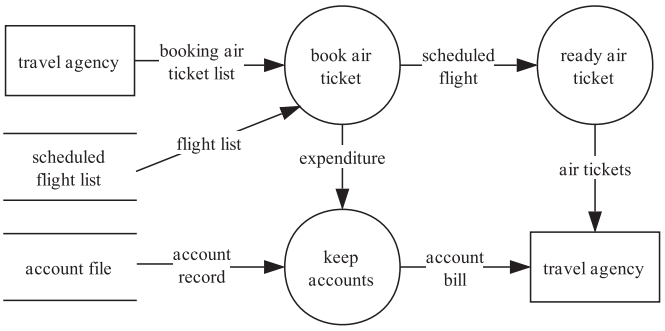

Fig. 1. An example DFD of air ticket booking system (Zhao et al., 2009).

To facilitate the understanding of the traditional DFD, an example of booking air ticket system (Zhao et al., 2009) (as shown in Fig. 1) displays the four basic elements, which are introduced as follows.

- **Process Notations** (or Operation Notations) refer to the activities that operate data of the system. An operation is depicted as a circle with a unique name in form of verb or verb phrase, for example, “book air ticket” and “keep accounts” in Fig. 1. Different from the circle type representation from Constantine et al., operations in DFDs defined by Gane et al. are squares with rounded corners.
- **Data Store Notations** represent the repository of data manipulated by operations, which can be databases or files (“scheduled flight list” and “account file” in Fig. 1). A data store is represented by a rectangle in a DFD with a name in the form of noun or noun phrase.
- **Data Flow Notations** are directed lines indicating the data from or to an operation with the information on the line of a data flow. At least one end of a data flow is linked to a circle of operation.
- **External Entity Notations** stand for the objects which are the sinks or sources of data outside the system. An example of external entity in Fig. 1) is “travel agency”. Similar to the representation of data store notation, an external entity is also depicted as a closed rectangle in DFDs.

Note that data cannot move without a process. In other words, data cannot go to or come from a data store or an external entity without having a process pushing it or pulling it.

3.1.2. Hierarchy of DFD

Different from the single-level DFD supporting for decomposition in our conference paper (Chen et al., 2017), we analyze DFDs in multiple hierarchies to identify the sets of processes and the relevant data stores with various degrees of detail and help derive microservices with different granularities. As mentioned by Rosenblatt (2013), most business processes of a system are too complex to be explained in one single DFD. Most process models are therefore composed of a set of DFDs. The first DFD provides a summary of the overall system, with additional DFDs gradually providing more detail about each part of the overall business process. Therefore, one important principle in process modeling with DFDs is the structured analysis of the business process into a hierarchy of DFDs, with decomposing each level from the top down and providing more detail on the subsequent level.

- **Context diagram** is the initial DFD in every business process model, either a manual system or a computerized system. As the name suggests, the context diagram is the highest-level

view of the system and shows the entire system in context with its environment. All process models have one context diagram (Rosenblatt, 2013).

- **Level-0 data flow diagram** is the next level DFD that shows all the processes at the first level of numbering (i.e. processes numbered from 1 through n), the data flows, the data stores, and external entities. The purpose of the Level-0 DFD is to depict all the major processes of the system and how they are interrelated. All process models have one and only one Level-0 DFD.
- **Level-1 data flow diagram** is generated per process by decomposing Level-0 DFD because the context diagram deliberately hides some of the system's complexity and Level-0 DFD only shows how the major high-level processes in the system interact. Level-1 DFD shows how it operates in an increasing level of detail denoted by the process numbers with one decimal point (e.g., 1.1, 1.2, 2.1).

It is convenient for analyst or architect to identify microservices at different granularities through processes with corresponding detail. However, it does not imply that the smaller the microservices, the better. Over fine-grained microservices may have negative influence on the performance of the entire system because of the communication overheads among them. To clearly describe our approach without introducing much complexity in limited space, we only analyze and construct Level-0 DFD and Level-1 DFD for the dataflow-driven decomposition (Section 3.2), because the context diagram shows the overall business process as just one process (i.e. the system itself) and often misses data stores (Rosenblatt, 2013), which are necessary for our dataflow-driven decomposition.

Note that how to construct the above diagrams at different levels is beyond the scope of this paper.

3.1.3. Conceptual definition and representation

To facilitate the application of rules (cf. Section 3.3) and simplify the decomposition process, we use the adapted definitions of DFD to conceptually represent the DFD and the transmission of data according to the studies (Ibrahim and Siow, 2011; Adler, 1988).

Definition 1. Let G be a data flow diagram, then

$G = \{P, D, S, E\}$

where

$P = \{P_1, P_2, P_3, \ldots, P_m\}$ indicates the set of processing (operation) nodes;

$D = \{D_1, D_2, D_3, \ldots, D_n\}$ indicates the set of data flows;

$S = \{S_1, S_2, S_3, \ldots, S_x\}$ indicates the set of data store nodes;

$E = \{E_1, E_2, E_3, \ldots, E_y\}$ indicates the set of external entity nodes.

Given the aforementioned air ticket booking system (Zhao et al., 2009), the conceptual DFD is displayed in Fig. 2 with the following elements:

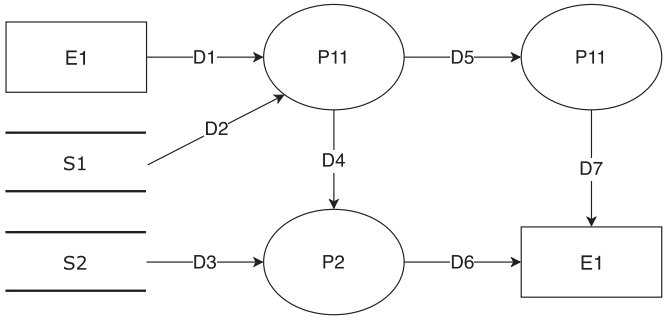

Fig. 2. Conceptual DFD for air ticket booking system (Zhao et al., 2009).

$P = \{P_1, P_2, P_3\}$

$D = \{D_1, D_2, D_3, D_4, D_5, D_6, D_7\}$

$S = \{S_1, S_2\}$

$E = \{E_1\}$

Sentence is used in our study for the interpretation of data flows in DFDs according to Adler (1988). As defined by Adler, *sentences* are transform lists including input elements, output elements, and transforms. They can be further interpreted as a decomposition DFD, which is the analyzed by our dataflow-driven decomposition approach (Section 3.2). The definition of *sentence* is displayed as follows.

Definition 2. A *sentence* of DFD includes start nodes, end nodes, and the data flow direction between them,

$(I \rightarrow O)$

where

$I, O \in (P \cup S \cup E)$ indicates the start nodes (P or S or E) and end nodes (P or S or E) respectively; $\rightarrow$ indicates the connection direction from start nodes to end nodes.

Note that there should be nine kinds of *sentences* by combing three start nodes and three end nodes and considering the data flow direction between them. Different types of *sentences* have the specific meanings, which are specified as below. Recall that data cannot go to or come from a data store or an external entity without having a process (cf. Section 3.1.1), therefore four types of *sentences* are impossible to be identified from DFDs, i.e. $(S_i \rightarrow S_j)$, $(E_i \rightarrow E_j)$, $(S_i \rightarrow E_j)$, and $(E_i \rightarrow S_j)$. Here we only explain the specific meanings of the other five possible kinds of *sentences* and use *sentence* examples extracted from Fig. 2 to help better understanding.

(1) $I \in E$ **and** $O \in P$: $(E_i \rightarrow P_j)$
Meaning: The entity E_i provides data for the process P_j.
Example: $(E_1 \rightarrow P_1)$

(2) $I \in P$ **and** $O \in E$: $(P_i \rightarrow E_j)$
Meaning: The process P_i outputs data for the entity E_j.
Example: $(P_2 \rightarrow E_1)$, $(P_3 \rightarrow E_1)$

(3) $I \in P$ **and** $O \in P$: $(P_i \rightarrow P_j)$
Meaning: The process P_i outputs data for the entity P_j.
Example: $(P_1 \rightarrow P_2)$, $(P_1 \rightarrow P_3)$

(4) $I \in S$ **and** $O \in P$: $(S_i \rightarrow P_j)$
Meaning: The process P_i reads data from the data store S_j.
Example: $(S_1 \rightarrow P_1)$, $(S_2 \rightarrow P_2)$

(5) $I \in P$ **and** $O \in S$: $(P_i \rightarrow S_j)$
Meaning: The process P_i writes data D_j into the data store S_k.
Example: No $(P_i \rightarrow S_j)$ *sentence* is found in Fig. 2.

3.2. Decomposition process

The process of proposed DFD-driven mechanism is shown in Fig. 3, which comprises four steps on the basis of the theoretical foundation (cf. Section 3.1) and the decomposition rules (cf. Section 3.3). Fig. 3 displays that the first two steps, i.e. business logic analysis and detailed DFD construction including process-datastore version of DFDs (DFD_{PS}) are manually conducted by requirement analysts and data analysts respectively. During these two steps, use cases and business logics of system are analyzed to generate detailed DFDs at different levels conforming to Rule 1/2/3 and then detailed DFDs are converted to DFD_{PS} on the basis of Rule 4 (cf. Section 3.3). Given the detailed and the process-datastore version of DFDs, this paper provides two algorithms to automate the construction of decomposable DFDs using Rule 5/6 (cf. Section 3.3) and identify microservice candidates from

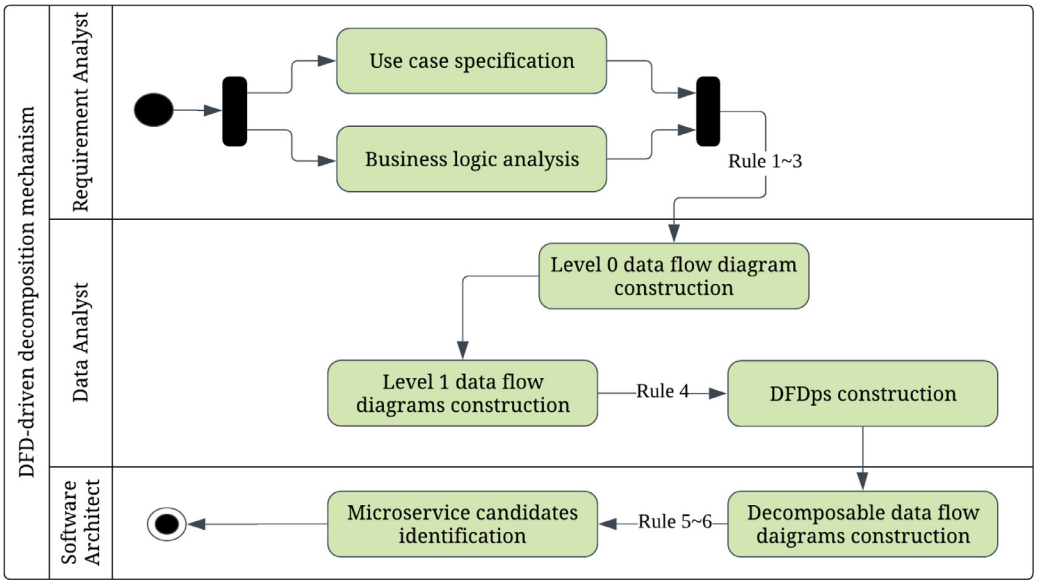

Fig. 3. The process of proposed dataflow-driven decomposition mechanism for microservices.

decomposable DFDs. The detail of these four steps are specified as follows.

Step-1. **Analyze requirements**
The proposed decomposition method begins with analyzing the business logics according to the use cases specifications and user's natural-language descriptions. Furthermore, domain context and entities (Evans, 2004) are also identified in this step to support constructing the DFDs and ensuring the entity cohesion of further decomposition.

Step-2. **Construct detailed DFDs and process-datastore version of DFDs**
The DFDs at different levels are constructed manually to illustrate the detailed data flow based on the use cases and business logic analysis in the last step. The construction of detailed DFDs can start from establishing the Level-0 DFD and then Level-1 DFD with fine detail according to our predefined rules (cf. Rule 1/2/3 in Section 3.3). Using Rule 4, the process-datastore version DFD (DFD_{PS}) is constructed from the detailed DFD. The DFD_{PS} focuses on the relationship between processes and the related data stores only, while excluding the side information like external entities and the data transmitted on the data flow (**Definition 3**).

Step-3. **Condense DFD_{PS} into decomposable DFDs.**
By extracting the *sentences* ($S_i \rightarrow P_j$) and ($P_i \rightarrow S_j$) (cf. Definition 2) in the constructed DFDs, Algorithm 1 can automatically condense the DFD_{PS} between processes and data stores specified in the DFD into a decomposable set, which is called *decomposable DFD* in this study (**Definition 4**). It is noteworthy that further decomposition based on these two kinds of *sentences* essentially decreases the size of microservices and reduces the complexity of ensuring the data consistency.

Step-4. **Identify microservice candidates from decomposable DFDs.**
Given a previously-generated decomposable DFD, Algorithm 2 can automatically group modules of fine-grained processes and the related data stores from the decomposable data diagrams according to Rule 5 and Rule 6 (cf. Section 3.3). The separation of the modules eventually completes and represents the dataflow-driven microservice-oriented decomposition, while each module indicates a potential microservice to be considered by software architects.

Algorithm 1 Decomposable DFD construction Algorithm.

Input: PDF: Sets representing a DFD_{PS}
Output: DDF: Sets representing a decomposable DFD

▷ identify sentences of processes and its related data store
$PS \leftarrow \emptyset$ ▷ PS : the process set of decomposable DFD
$SS \leftarrow \emptyset$ ▷ S : the data store set of decomposable DFD
$ST \leftarrow \emptyset$ ▷ ST : the sentence set of decomposable DFD
for $P_i \in PDF.process\_set$ **do**
▷ $PDF.process\_set$: the process set of the DFD_{PS}
  if $P_i \notin PS$ **then**
    $PS \leftarrow PS \cup P_i$
  end if
  for $S_j \in PDF.ds\_set$ **do**
▷ $PDF.ds\_set$: the data store set of the DFD_{PS}
    new ST_n ▷ S_n : new identified sentences
    if $S_j \in P_i.output$ **or** $P_i \in S_j.input$ **then**
      $ST_n.input \leftarrow ST_n.input \cup P_i$
      $ST_n.output \leftarrow ST_n.output \cup S_j$
    end if
    if $S_j \in P_i.input$ **or** $P_i \in S_j.output$ **then**
      $ST_n.input \leftarrow ST_n.input \cup S_j$
      $ST_n.output \leftarrow ST_n.output \cup P_j$
    end if
  end for
end for
for $S_j \in PDF.ds\_set$ **do**
  if $S_j \notin SS$ **then**
    $SS \leftarrow SS \cup S_j$
  end if
end for
new DDF ▷ DDF : new decomposable DFD
$DDF.ds\_set \leftarrow SS$
$DDF.process\_set \leftarrow PS$
$DDF.sentence\_set \leftarrow ST$
return DDF

Algorithm 2 Microservices Identification Algorithm.

Input: DDF: Sets representing the decomposable DFD
Output: MSS: the decomposed microservice list

```
▷ identify microservices from decomposable DFD
MSS ← ∅                          ▷ MS : the sentence sets of microservices
for S_i ∈ DDF.ds_set do
    new MS                                   ▷ MS : new service
    new MS.process                ▷ MS.process : the process set
    new MS.ds                       ▷ MS.ds : the data store set
    for ST_i ∈ DDF.sentence_set do
        if S_i = ST_i.input then
            MS ← MS ∪ ST_i
            if ST_i.output ∉ MS.process then
                MS.process ← MS.process ∪ ST_i.output
            end if
        end if
        if S_i = ST_i.output then
            MS ← MS ∪ ST_i
            if ST_i.input ∉ MS.process then
                MS.process ← MS.process ∪ ST_i.input
            end if
        end if
        MS.ds ← MS.ds ∪ S_i
    end for
    for MS_i ∈ MSS do
        if MS_i.process ∩ MS.process = ∅ then
            MSS ← MSS ∪ MS
        else
            MS_i ← MS_i ∪ MS
            MS_i.process ← MS_i.process ∪ MS.process
            MS_i.ds ← MS_i.ds ∪ MS.ds
        end if
    end for
end for
return MSS
```

3.3. Rules for decomposition

We improve the decomposition approach reported in our conference version through constructing DFDs at different levels, which gradually refines Level-0 DFD by specifying the process activities and introducing more side information for diagrams of next level conforming to rules predefined. Correspondingly, this paper revises and updates the rules to guide the construction of DFDs at different levels and identifying microservice candidates from these DFDs. Compared with the conference version, this paper includes six dominating rules that are specified in this subsection.

- **Rule 1.** When constructing DFDs, processes cannot consume or create data and data cannot move without a process. To be specific, a process cannot destroy input data; all processes must have outputs. Likewise, a process cannot create new data from nothing, i.e. it can transform data from one form to another, but it cannot produce output data without inputs. In terms of data, it cannot go to or come from a data store or an external entity without having a process to push or pull it.
- **Rule 2.** When constructing DFDs, processes should be verb phrases (starting with a verb and including a noun) that can reflect the semantic meaning of the corresponding data processing activities, while data should be named using semantically meaningful nouns or noun phrases occurred in the original business logic.
- **Rule 3.** The key principle in creating sets of DFDs is to ensure that all information presented in a DFD at one level is accurately represented in the next-level DFD. For example, Level-1 DFD replaces Level-0 DFD's processes (always numbered P_i) with multiple sub-processes, adds data stores, and includes additional data flows that were not in Level-0 DFD. The sub-processes and data flows added are contained within process of Level-0 DFD. They were not shown on Level-0 DFD because they are the internal components of process P_i. Only after understanding Level-0 DFD does the analyst "open up" process P_i to define the internal processes by decomposing Level-0 DFD into Level-1 DFD, which conveys more detail about the processes and the data flows inside the system.

These three rules are borrowed from Rosenblatt (2013) and used for DFD construction in the second step (cf. Section 3.2) of our decomposition approach (cf. Fig. 3). Given the detailed DFDs, Rule 4 is applied to construct the process-datastore version of DFDs (DFD_{PS}) for efficient decomposition by excluding irrelevant information.

- **Rule 4.** To achieve more effective microservice-oriented decomposition, we are only concerned with the relationship between processes and data stores in DFD_{PS}, without the need of information of external entities E and the data D on the data flow.
  Reason 1: It is whether there is the dependency (read or write) between processes and data stores that depicts the cohesion of processes in our decomposition method, other than what specific data D they transmit.
  Reason 2: All external entities E are merely the provider or consumer of specific input data or output data outside of the system and have no influence on the dataflow-driven decomposition, which identifies the microservice candidates based on the relationship between operations and data stores within the system. For example, $(E_1 \rightarrow P_1)$, $(P_2 \rightarrow E_1)$, and $(P_3 \rightarrow E_1)$ in the DFD of air ticket booking system (cf. Fig. 2).
  Reason 3: Similar to *Reason 2*, processes outside the cohesive process set related to specific data stores should request or post data through APIs exposed, therefore the dependency among processes (P_i, P_j) can be neglected in our decomposition method.

After generating DFD_{PS}, Algorithm 1 can be used to construct the decomposable DFDs in Step-3 (cf. Section 3.2), then Rule 5 and Rule 6 are used by Algorithm 2 to identify microservice candidates from the constructed DFDs (cf. Fig. 3) based on the *Database-per-Service* pattern (Messina et al., 2016) and the performance consideration (Richards, 2015).

- **Rule 5.** The decomposition of DFD is to cluster *sentences* including the same data stores into the identical set for one microservice candidate.
  Reason: According to the *Database-per-Service* pattern (Messina et al., 2016), each microservice has its own database for loose coupling and scalability. Its associated database is an effective part of the implementation of that service. It cannot be accessed directly by other services. Therefore, this rule can ensure that processes write data into or read data from the same data stores are grouped into single microservice candidate.
  Fig. 4 is a conceptual representation of Rule 5. It displays three kinds of circumstances in which *sentences* are clustered when applying Rule 5 as follows.
  a) Part of processes write the data into related data store, while part of processes read data from it. Then all related processes of this data store should be clustered into the same sentence set (Fig. 4a):
  Given $S_k, S'_k \in S, \{P_1, \ldots, P_n\} \in P$,
  $\{(P_1 \rightarrow S_k)\}, \ldots, \{(S'_k \rightarrow P_n)\}$,
  if $S_k = S'_k$,
  then $\{(P_1 \rightarrow S_k)\}, \ldots, \{(S'_k \rightarrow P_n)\} \Rightarrow$
  $\{(P_1 \rightarrow S_k), \ldots, (S_k \rightarrow P_n)\}$

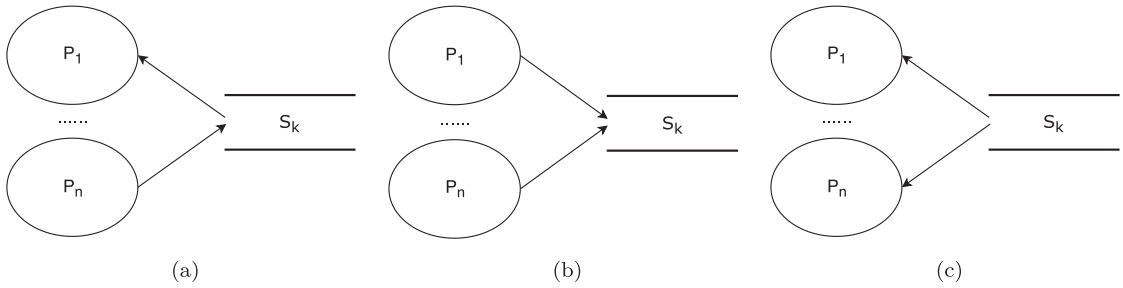

Fig. 4. A conceptual representation of Rule 5.

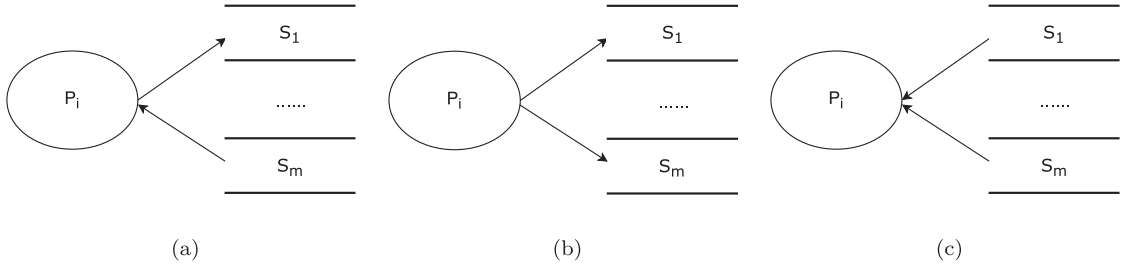

Fig. 5. A conceptual representation of Rule 6.

b) All processes that write data into the identical data store should be clustered into the same sentence set (Fig. 4b):
Given $S_k, S'_k \in S, \{P_1, \ldots, P_n\} \in P$,
$\{(P_1 \rightarrow S_k)\}, \ldots, \{(P_n \rightarrow S'_k)\}$,
if $S_k = S'_k$,
then $\{(P_1 \rightarrow S_k)\}, \ldots, \{(P_n \rightarrow S'_k)\} \Rightarrow$
$\{(P_1 \rightarrow S_k), \ldots, (P_n \rightarrow S_k)\}$
c) Processes only read data from the identical data store should be clustered into the same sentence set (Fig. 4c):
Given $S_k, S'_k \in S, \{P_1, \ldots, P_n\} \in P$,
$\{(S_k \rightarrow P_1)\}, \ldots, \{(S'_k \rightarrow P_n)\}$,
if $S_k = S'_k$,
then $\{(S_k \rightarrow P_1)\}, \ldots, \{(S'_k \rightarrow P_n)\} \Rightarrow$
$\{(S_k \rightarrow P_1), \ldots, (S_k \rightarrow P_n)\}$
Note that Fig. 4a is the general circumstance in which a specific data store and the related processes writing data into or reading data from it are clustered, while Fig. 4b and Fig. 4c are special cases of Fig. 4a including processes which only have write or read operations respectively.

- **Rule 6.** The decomposed sets after applying Rule 5 should be merged when there are duplicate *processes* among them.
Reason: Once there are duplicate *processes* among two different decomposed sets, it means that the overall performance may be increased because more remote calls are needed (Richards, 2015). Considering the performance improvement, Rule 6 should be applied to reduce the possibile distributed calls among microservices.
There may also be three kinds of circumstances in which *sentence* sets are merged when applying Rule 6, which is shown in Fig. 5.
a) If one process reads data from one data store and writes data into another data store, then the related sentence sets including this process should be merged together (cf. Fig. 5a).
Given $\{S_1, \ldots, S_m\} \in S, P_i, P'_i \in P$
$\{(P_i \rightarrow S_1), \ldots\}, \ldots, \{(S_m \rightarrow P'_i) \ldots\}$,
if $P_i = P'_i$,
then $\{(P_i \rightarrow S_1), \ldots\}, \ldots, \{(S_m \rightarrow P'_i), \ldots\} \Rightarrow$
$\{(P_i \rightarrow S_1), \ldots, (S_m \rightarrow P_i)\}$
b) If one specific process writes data into multiple data stores, then the related sentence sets including this process should be merged together (cf. Fig. 5b):
Given $\{S_1, \ldots, S_m\} \in S, P_i, P'_i \in P$
$\{(P_i \rightarrow S_1), \ldots\}, \ldots, \{(P'_i \rightarrow S_m), \ldots\}$,
if $P_i = P'_i$,
then $\{(P_i \rightarrow S_1), \ldots\}, \ldots, \{(P'_i \rightarrow S_m), \ldots\} \Rightarrow$
$\{(P_i \rightarrow S_1), \ldots, (P_i \rightarrow S_m)\}$
c) If one specific process reads data from multiple data stores, then the related sentence sets including this process should be merged together (cf. Fig. 5c):
Given $\{S_1, \ldots, S_m\} \in S, P_i, P'_i \in P$
$\{(S_1 \rightarrow P_i), \ldots\}, \ldots, \{(S_m \rightarrow P'_i), \ldots\}$,
if $P_i = P'_i$,
then $\{(S_1 \rightarrow P_i), \ldots\}, \ldots, \{(S_m \rightarrow P'_i), \ldots\} \Rightarrow$
$\{(S_1 \rightarrow P_i), \ldots, (S_m \rightarrow P_i)\}$
Fig. 5a is the general circumstance in which the *sentence* sets are merged, as long as there are duplicate process P_i in those *sentence* sets, no matter the process has reading or writing operation on data stores. While Fig. 5b and Fig. 5c are special cases of Fig.5a, in which the merged sets have the same processes that only read data from or write data into related data stores respectively.

Among the six rules, Rule 1/2/3 are basic rules for DFD construction which are borrowed from Rosenblatt (2013). Rule 1 and Rule 2 are basic requirements of DFD construction, while Rule 3 specifies how to draw DFDs at different levels. In addition, we improve and adapt the DFD_{PS} construction rule (Rule 4) and decomposition rules (Rule 5 and Rule 6) to DFDs at different levels in this paper. In particular, Rule 5 and Rule 6 can be automatically generated. Construction and decompostion based on DFDs according to these rules are demonstrated through a typical case in Section 4.

4. Case study

To demonstrate and validate our microservice-oriented decomposition approach, we conducted a case study, in which a

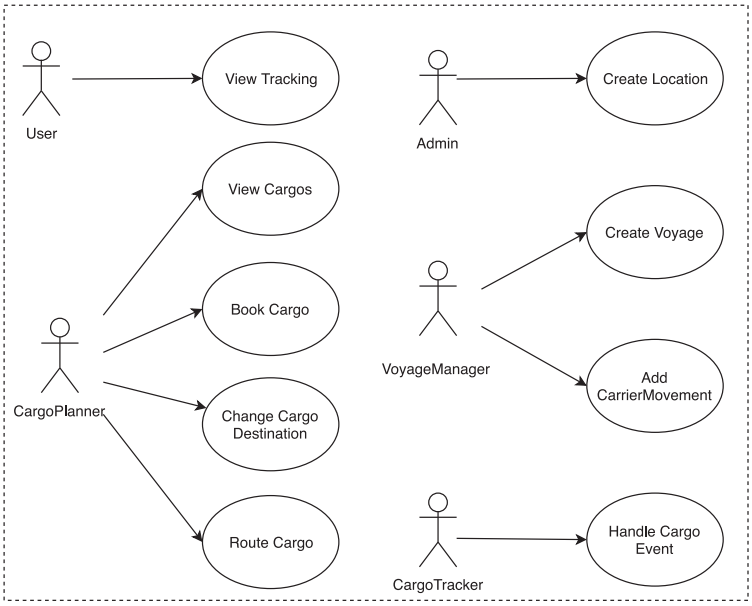

Fig. 6. The use case for the high-level functional requirements of Cargo Tracking System.

monolithic application Cargo Tracking System was decomposed through our dataflow-driven approach. Cargo Tracking System is a typical case used by Evans (2004) to illustrate the Domain-Driven Design (DDD) and open sourced on Github[1]. Since the decomposition and refactoring of a monolithic application at enterprise level into microservices can be a complicated and time-consuming task, we considered this sample application manageable in size and concentrated on the decomposition stage only. Although the complexity of Cargo Tracking System might not be mentioned as the systems from enterprises, this case is a complete system and more complex to some extent than the small and separate business logics analyzed in the conference version (Chen et al., 2017). Moreover, two related studies (Gysel et al., 2016; Baresi et al., 2017) also used it to evaluate their decomposition approaches. Hence, Cargo Tracking System is a favorable choice for us to demonstrate the complexity of decomposition close to reality as well as enable the comparison of our approach to other related studies with the same case.

4.1. Requirements analysis

The decomposition approach proposed in this paper starts with analyzing functional requirements, and then constructing detailed DFDs (cf. Section 4.2) according to those requirements and business logics' analysis. There are many analysis methods of gathering and assessing application requirements, e.g., use case analysis. Use case analysis identifies the actors in a system and the operations they may perform. The requirements of this case study come from the Cargo Tracking System described by Evans (2004) and the use case analysis given by Gysel et al. (2016).

Fig. 6 depicts the high-level system functional requirements of Cargo Tracking System, which includes five roles (User, CargoPlanner, CargoTracker, VoyageManager and Admin).

- Users view tracking information.
- CargoPlanners view cargos' information and manage cargos, for example *Book Cargo*.
- CargoTrackers track the progress of each Cargos itinerary through *Handle Cargo Event*.
- VoyageManagers manage the voyages through *Create Voyage* and *Add CarrierMovement*
- Admins are responsible for *Create Location*.

To support the dataflow-driven decomposition method focusing the dependency of opetations and data stores, we identify nine data stores from Cargo Tracking System according to Gysel et al. (2016), which are briefly explained in Table 1.

On the basis of the requirement analysis from Gysel et al. (2016), we correspond the nine use cases in Fig. 6 to relatively coarse-grained operations and analyze the interactions among them with specific data stores through Table 2.

Table 1
The brief explanation of data stores in Cargo Tracking system.

No.	Data stores	Explanation
1	Cargo	Storing the unique identifier of cargos (*TrackingId*).
2	RouteSpecification	Storing the expected itinerary information of cargos, e.g., *Origin, Destination*, and *arrivalDeadline*.
3	Itinerary	Storing the unique identifier of a cargo's expected route (*ieneraryNumber*).
4	Leg	Storing information related to every step in Itinerary, such as loading location (*loadLocation*), unloading location (*unloadLocation*), loading time (*loadTime*), and unloading time (*unloadTime*).
5	Location	Storing information about locations, e.g., the unique identifier *unLocode* and the name (*name*) of a location.
6	HandlingEvent	Storing information about the historical operation events of cargos, such as operation type (*upload* etc.), location (*location*), completion time (completionTime), registration time (*registrationTime*).
7	Delivery	Storing the information related to the current delivery status of cargos, including transport status, *misdirected* or not, unloaded (*isUnloadedAtDestination*), estimated arrival time (*estimatedArrivalDate*), route status (*routingStatus*).
8	Voyage	Storing the unique identifier of voyages (*voyageNumber*).
9	CarrierMovement	Storing voyages' information such as *departureLocation,arrivalLocation, departureTime* and *arrivalTime*

[1] https://github.com/citerus/dddsample-core.

Table 2
The interaction between coarse-grained operations and specific data stores in Cargo Tracking system.

Use cases	Explanation
View Tracking	Users can view all cargos' tracking information based on the inputted *Cargo.trackingId*, including the route planning information in *RouteSpecification*, the voyage information in *Voyage*, the hand-off event information in *HandlingEvent*, the delivery information in *Delivery*, etc.
View Cargos	Cargo planners can view all cargos' information, which includes the route information in *RouteSpecification*, the delivery information in *Delivery*, the voyage planning information in *Itinerary*, etc.
Book Cargo	Cargo planners can book cargos, in which the generated *Cargo.trackingId* and location information are written into *RouteSpecification* and *Location* respectively.
Change Cargo Destination	Cargo planners can change a cargo's destination through modifying *RouteSpecification* based on the *Cargo.trackingId* and the new destination input. Meanwhile, it writes the route status in *Delivery*.
Route Cargo	Cargo planners can arrange transportation routes for a cargo. The arrangement of transportation routes is based on the cargo's route planning information from *RouteSpecification*, the voyage planning information from *Itinerary*, the *Leg*, and the specific voyage information from *Voyage* and *CarrierMovement*.
Create Location	Admins can add new location into *Location*.
Create Voyage	Voyage managers can add new voyage into *Voyage*.
Add CarrierMovement	Voyage managers can also add the docking information into *CarrierMovement*. Each *CarrierMovement* includes the information of one docking, i.e. docking location and arrival time. Each voyage may contain one or more CarrierMovements.
Handle Cargo Event	Cargo trackers can track the progress of each cargo's itinerary through *HandlingEvent* and *Delivery*.

4.2. Constructing detailed DFDs and DFD_{PS}

Further construction of data flow diagrams starts with the information provided by the use cases and the requirements definition (Rosenblatt, 2013). Given the above requirement analysis and the DFD construction rules (cf. Section 3.1.2), the detailed DFDs and DFD_{PS} with different levels are constructed step by step.

1) **Detailed DFDs**

Level-0 DFD: In this step, the top level use case (cf. Fig. 6) is converted into Level-0 DFD according to requirements analysis introduced (cf. Section 4.1). When converting the use case into a DFD, we need to follow the formal Rule 1 and Rule 2 to naming processes on the DFD first. Then we add the data flows and data stores to the DFD according to the process analysis of business logics, which is omitted by the use case. The information about the major sub-processes that make up each use case is ignored at this point and it is used in a later step. After this step, the Level-0 DFD is constructed as shown in Fig. 7. It shows that the name of the nine processes is identical to the name of nine use cases in 6 with the adherence to Rule 1

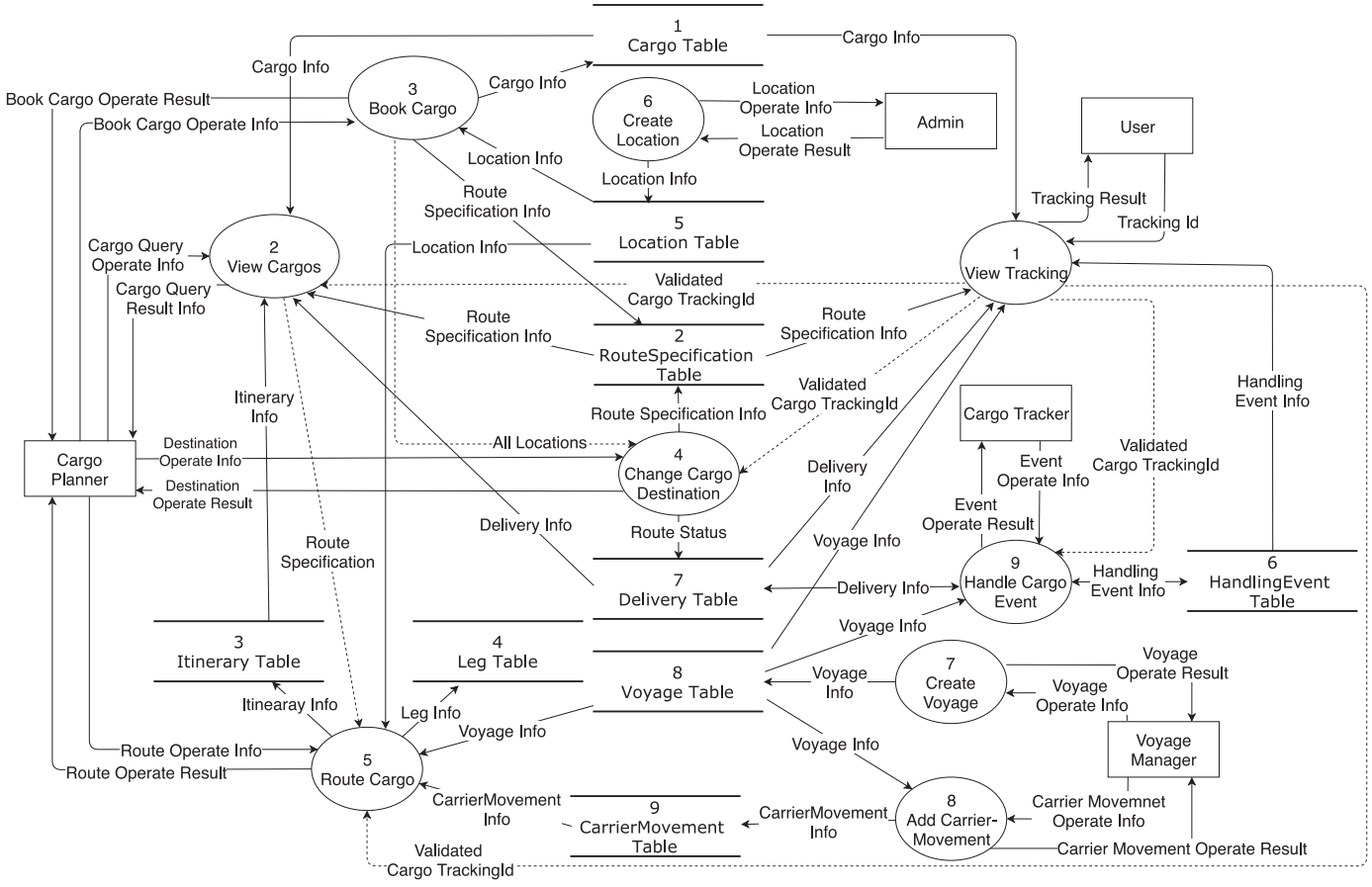

Fig. 7. Level-0 DFD of Cargo Tracking System.

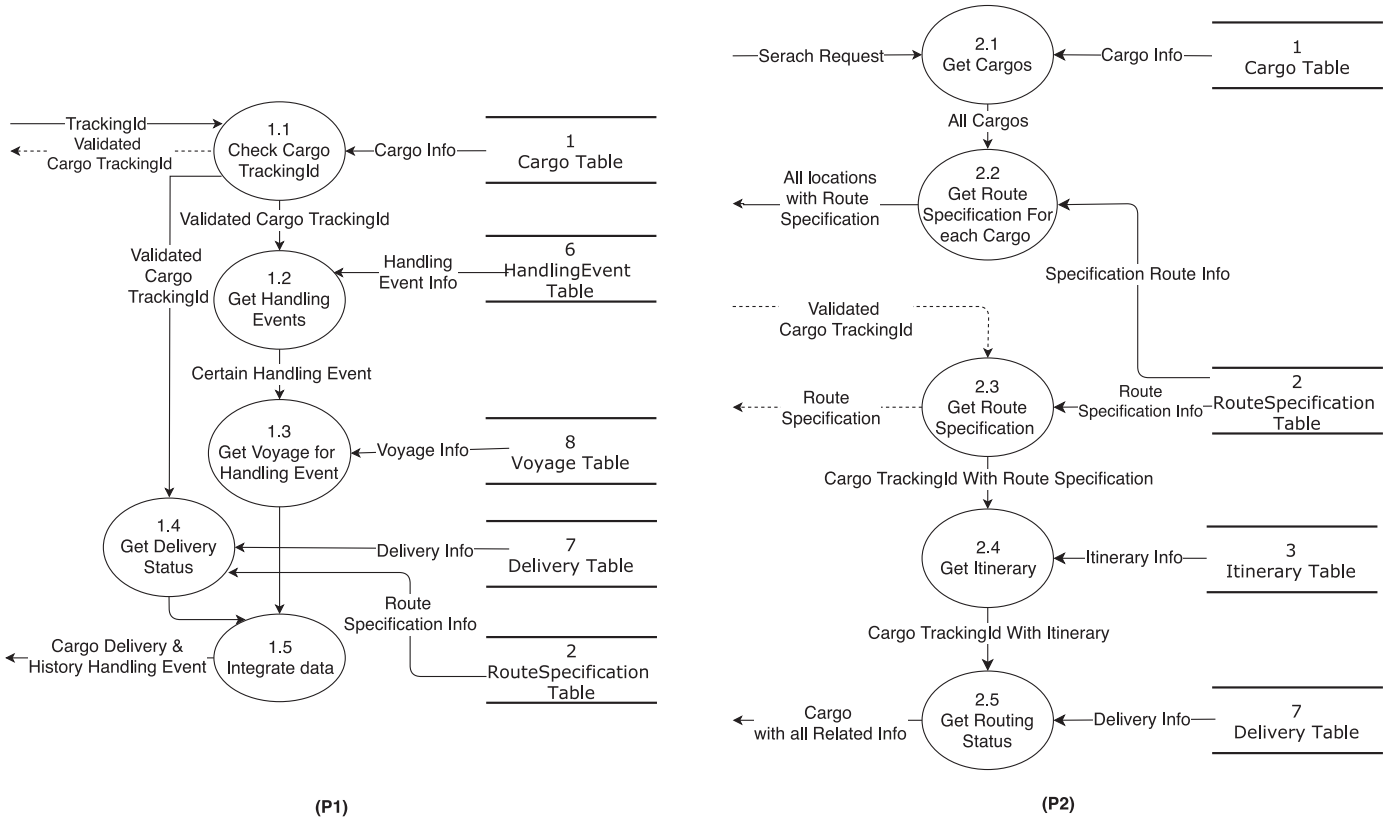

Fig. 8. Level-1 DFD of Cargo Tracking System (P1-P2) (*Fragments for P3-P9 are displayed in Appendix.*).

and Rule 2. Moreover, we add data flows among external entities, processes and data stores to the diagram on the basis of business logics as well.

Level 1 DFD: After the last step, we create lower-level DFDs for each process in the Level-0 DFD based on requirements and business logics analysis. The process for creating the Level-1 DFDs is to take the steps as written on the use cases and convert them into a DFD in much the same way as for the Level-0 DFD, such as introducing input data flows that were not included in the use case. According to Rule 1, Rule 2 and Rule 3 (cf. Section 3.3), the Level-1 DFDs of the Cargo Tracking system are constructed and Fig. 8 shows the fragment related to Process 1 (View Tracking) and Process 2 (View Cargos) in the Level-0 DFD. DFDs at this level for the rest six processes are displayed in the Appendix (Fig. A.1).

Given the Level-0 DFD and Level-1 DFDs, the team validates the set of DFDs to make sure that they are complete and correct, and contain no syntactic or semantic errors. Analysts seldom create DFDs perfectly at the first time, and iteration is important in ensuring that both single-page and multipage DFDs are clear and easy to read.

2) ***DFD_{PS}***

Given the detailed DFDs at Level 0 and Level 1, we manually constructed the DFD_{PS} using Rule 4. Recall that the DFD_{PS} excludes the irrelevant information such as external entity and the data on the data flow from the detailed DFDs. The process-datastore versions of detailed DFDs are conceptually displayed in Figs. 9, 10, and A.2 (Appendix) respectively. Note that the processes at Level-1 DFD_{PS} corresponding to P_i are represented by $\{P_{i1}, P_{i2}, \ldots, P_{in}\}$, which means that their parent process at Level-0 DFD is the process P_i.

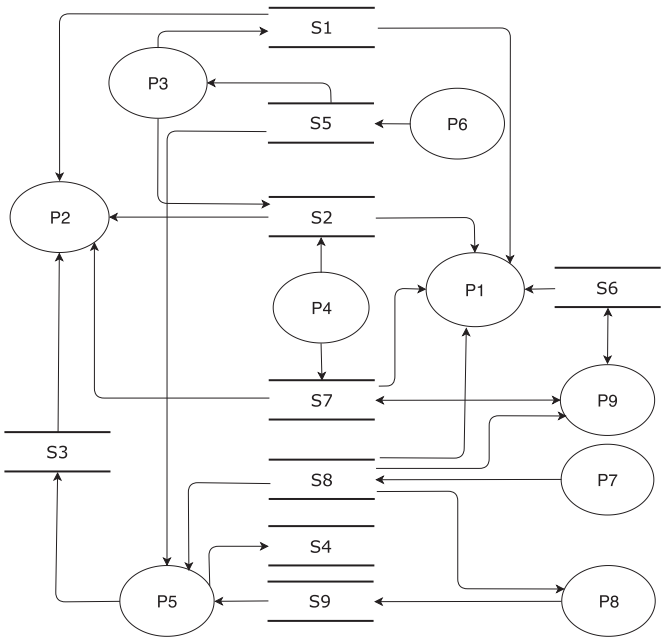

Fig. 9. Level-0 DFD_{PS} of Cargo Tracking System.

4.3. *Condensing* DFD_{PS} *into decomposable DFDs*

After constructing DFD_{PS}, the regularity should be checked based on Rule 4 to ensure that all entities as well as the dependency related to them are removed. Moreover, the data on the data

Table 3
Decomposable Level-0 DFD identified by Algorithm 1.

Element	Value
Process	$\{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$
Data Store	$\{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9\}$
Sentence	$\{S_1 \rightarrow P_1, S_2 \rightarrow P_1, S_6 \rightarrow P_1, S_7 \rightarrow P_1, S_8 \rightarrow P_1, S_1 \rightarrow P_2, P_2 \rightarrow S_2, S_3 \rightarrow P_2, S_7 \rightarrow P_2, P_3 \rightarrow S_1, P_3 \rightarrow S_2, S_5 \rightarrow P_3, P_4 \rightarrow S_2, P_4 \rightarrow S_7, P_5 \rightarrow S_3, P_5 \rightarrow S_4, S_5 \rightarrow P_5, S_8 \rightarrow P_5, S_9 \rightarrow P_5, P_6 \rightarrow S_5, P_7 \rightarrow S_8, S_8 \rightarrow P_8, P_8 \rightarrow S_9, P_9 \rightarrow S_6, S_6 \rightarrow P_9, P_9 \rightarrow S_7, S_7 \rightarrow P_9, S_8 \rightarrow P_9\}$

Table 4
Decomposable Level-1 DFD identified by Algorithm 1.

Element	Value
Process	$\{P_{11}, P_{12}, P_{13}, P_{14}, P_{15}, P_{21}, P_{22}, P_{23}, P_{24}, P_{25}, P_{31}, P_{32}, P_{33}, P_{41}, P_{42}, P_{43}, P_{51}, P_{52}, P_{53}, P_{61}, P_{71}, P_{81}, P_{91}, P_{92}, P_{93}, P_{94}\}$
Data Store	$\{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9\}$
Sentence	$\{S_1 \rightarrow P_{11}, S_6 \rightarrow P_{12}, S_8 \rightarrow P_{13}, S_2 \rightarrow P_{14}, S_7 \rightarrow P_{14}, S_1 \rightarrow P_{21}, S_2 \rightarrow P_{22}, S_2 \rightarrow P_{23}, S_3 \rightarrow P_{24}, S_7 \rightarrow P_{25}, S_5 \rightarrow P_{31}, P_{32} \rightarrow S_1, P_{33} \rightarrow S_2, P_{42} \rightarrow S_2, P_{43} \rightarrow S_7, S_5 \rightarrow P51, S_8 \rightarrow P_{52}, S_9 \rightarrow P_{52}, P_{53} \rightarrow S_3, P_{53} \rightarrow S_4, P_{61} \rightarrow S_5, P_{71} \rightarrow S_8, S_8 \rightarrow P_{81}, P_{82} \rightarrow S_9, P_{92} \rightarrow S_6, S_8 \rightarrow P_{92}, S_6 \rightarrow P_{93}, S_7 \rightarrow P_{93}, P_{94} \rightarrow S_7\}$

Table 5
Decomposition results of Algorithm 2 for decomposable Level-0 DFD.

MS No.	Sentence set	Process set	Data store set
MS_1	$\{S_1 \rightarrow P_1, S_2 \rightarrow P_1, S_6 \rightarrow P_1, S_7 \rightarrow P_1, S_8 \rightarrow P_1, S_1 \rightarrow P_2, P_2 \rightarrow S_2, S_3 \rightarrow P_2, S_7 \rightarrow P_2, P_3 \rightarrow S_1, P_3 \rightarrow S_2, S_5 \rightarrow P_3, P_4 \rightarrow S_2, P_4 \rightarrow S_7, P_5 \rightarrow S_3, P_5 \rightarrow S_4, S_5 \rightarrow P_5, S_8 \rightarrow P_5, S_9 \rightarrow P_5, P_6 \rightarrow S_5, P_7 \rightarrow S_8, \}$ $S_8 \rightarrow P_8, P_8 \rightarrow S_9, P_9 \rightarrow S_6, S_6 \rightarrow P_9, P_9 \rightarrow S_7, S_7 \rightarrow P_9, S_8 \rightarrow P_9\}$	$\{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$	$\{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9\}$

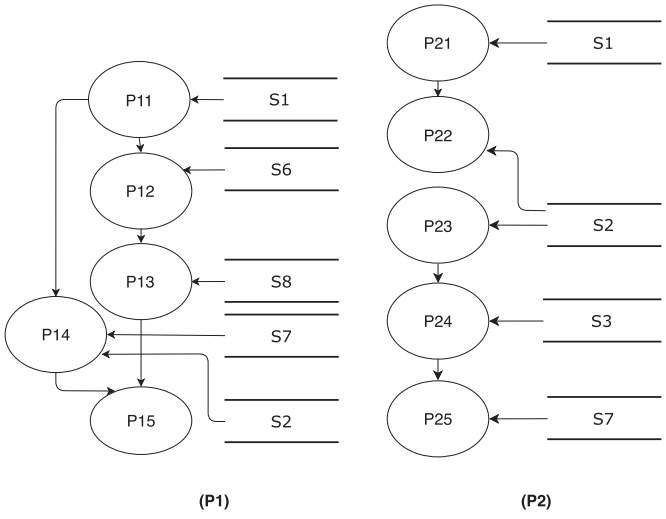

Fig. 10. Level-1 DFD_{PS} of Cargo Tracking System (P1-P2) (*Fragments for P3-P9 are displayed in Appendix*).

flows should be cleared and only the flow with direction are retained. It is clear that the DFD_{PS}(cf. Figs. 10 and A.2) conforms to Rule 4.

To condense the DFD_{PS} into a decomposable DFD automatically, we need to extract the process nodes, the data store nodes and the dependencies between processes and data stores from all DFD_{PS} as the matrix input of Algorithm 1 (cf. Section 3.2). Algorithm 1 that is implemented on the basis of Rule 5 identify the decomposable *sentence* sets of DFDs at Level 0 and Level 1 as displayed in Tables 3 and 4 respectively.

4.4. Identifying microservice candidates from decomposable DFDs

Given a decomposable DFD generated from the DFD_{PS}, the modules of processes and data stores can be automatically extracted through Algorithm 2 based on Rule 6. These modules indicate the potential microservice candidates that can be developed independently. Tables 5 and 6 show the decomposition results of Level-0 DFD and Level-1 DFD after the execution of Algorithm 2. Both of them display the identified microservices candidates with the corresponding processes and data stores. Table 5 shows that only one microservice candidate is obtained by analyzing Level-0 DFD, which is too coarse-grained and of no help for the decomposition. By contrast, we derive four candidates from the analysis of the Level-1 DFDs, which are *CargoService, PlanningService, LocationService* and *TrackingService* shown in Table 6.

5. Evaluation

The high cohesion and loose coupling are most concerned in microservice-oriented decomposition. As a principle for individual microservices, high cohesion requires every single microservice to implement a relatively independent piece of business logic. On the contrary, loose coupling is a crucial principle for involving multiple microservices, which requires the involved microservices to depend barely on each other (Newman, 2015). Recall that "processes and the related data store" modules can be extracted from a decomposable DFD into microservice candidates. By clustering processes tightly related to the same data stores, each "processes and data stores" module with high cohesion essentially emphasizes its own business logics of data processing. On the other side, processes which are not tightly related to a specific data store are excluded from the corresponding module.

In terms of appropriate metrics for measuring the high cohesion and loose coupling aspects, *Afferent Coupling* (*Ca*), *Efferent Coupling* (*Ce*), *Instability* (*I*), and *Relational Cohesion* (*RC*), were adopted in this study as recommended by Fritzsch (2018), who has conducted experiments to evaluate the decomposition results of two related studies (Gysel et al., 2016; Baresi et al., 2017). These four metrics were explained in detail in Table 7 and can be automatically gathered by Sonargraph Architect[2]. It is a powerful tool that allows the

[2] http://t.cn/E51WJn9.

Table 6
Decomposition results of Algorithm 2 for decomposable Level-1 DFDs.

MS No.	MS Name	Sentence set	Process set	Data store set
MS_1	*CargoService*	$\{S_1 \to P_{11}, S_1 \to P21, P_{32} \to S_1\}$	$\{P_{11}, P_{21}, P_{32}\}$	$\{S_1\}$
MS_2	*PlanningService*	$\{S_3 \to P_{24}, P_{53} \to S_3, P_{53} \to S_4\}$	$\{P_{24}, P_{53}\}$	$\{S_3, S_4\}$
MS_3	*LocationService*	$\{S_5 \to P_{31}, S_5 \to P_{51}, P_{61} \to S_5\}$	$\{P_{31}, P_{51}, P_{61}\}$	$\{S_5\}$
MS_4	*TrackingService*	$\{S_6 \to P_{12}, S_8 \to P_{13}, S_2 \to P_{14}, S_7 \to P_{14}, S_2 \to P_{22}, S_2 \to P_{23}, S_7 \to P_{25}, P_{33} \to S_2, P_{42} \to S_2, P_{43} \to S_7, S_8 \to P_{52}, S_9 \to P_{52}, P_{71} \to S_8, S_8 \to P_{81}, P_{82} \to S_9, P_{92} \to S_6, S_8 \to P_{92}, S_6 \to P_{93}, S_7 \to P_{93}, P_{94} \to S_7\}$	$\{P_{12}, P_{13}, P_{14}, P_{22}, P_{23}, P_{25}, P_{33}, P_{42}, P_{43}, P_{52}, P_{71}, P_{81}, P_{82}, P_{92}, P_{93}, P_{94}\}$	$\{S_2, S_6, S_7, S_8, S_9\}$

Table 7
Basic metrics for the evaluation of microservice candidates (Fritzsch, 2018).

Aspects	Metrics	Explanation
Coupling	Afferent Coupling (*Ca*) (Martin, 2002)	It is defined as the number of classes in other packages (services) that depend upon classes within the package (service) itself, as such it indicates the package's (service's) responsibility.
	Efferent Coupling (*Ce*) (Martin, 2002)	It measures the number of classes in other packages (services), that the classes in a package (service) depend upon, thus indicates its dependence on others.
	Instability (*I*) (Martin, 2002)	It measures a package's (service's) resilience to change through calculating the ratio of *Ce* and $Ce + Ca$. *I=0* indicates a completely stable package (service), whereas *I=1* a completely unstable package (service).
Cohesion	Relational Cohesion (*RC*) (Larman, 2012)	It is defined as the ratio between the number of internal relations and the number of types in a package (service). Internal relations include inheritance between classes, invocation of methods, access to class attributes, and explicit references like creating a class instance. Higher numbers of *RC* indicate higher cohesion of a package (service).

Table 8
Metrics of microservices identified by DFD driven approach.

Metrics	Cargo	Planning	Location	Tracking	Avg.
Ca	13	10	15	16	13.5
Ce	4	3	1	5	3.3
I	0.2	0.2	0.1	0.2	0.2
RC	1.8	11.5	21.5	14.1	12.2

Table 9
Metrics of microservices identified by Service Cutter (Fritzsch, 2018).

Metrics	Location	Tracking	Voyage & Planning	Avg.
Ca	14	11	15	13.3
Ce	1	3	4	2.7
I	0.1	0.2	0.2	0.2
RC	21.5	4.3	16.7	14.2

Table 10
Metrics of microservices identified by API analysis (Fritzsch, 2018).

Metrics	Planning	Product	Tracking	Trip	Avg.
Ca	16	13	15	8	13
Ce	2	4	5	2	3.3
I	0.1	0.2	0.3	0.2	0.2
RC	20.3	1.8	14.9	0	9.3

simulation of active refactorings without touching the code and assesses quality by analyzing metrics and code structure. With Sonargraph Architect, classes can be moved to different packages easily, which means assigning classes to the imagined microservices.

Since other decomposition methods (Kecskemeti et al., 2016; Mazlami et al., 2017; Tyszberowicz et al., 2018) have not been applied to the Cargo Tracking System in a timely manner, we evaluate the decomposition approach proposed in this paper by comparing its results against two representative related studies, Service Cutter (Gysel et al., 2016) and API analysis (Baresi et al., 2017), decomposing the same case. Fritzsch (2018) has evaluated the decomposition results of these two related studies (Gysel et al., 2016; Baresi et al., 2017) using Sonargraph Architect considering the four metrics in Table 7. Therefore we adopt this tool for further evaluation of the microservice candidates generated from the dataflow-driven method in this paper (cf. Table 6). Then we compare the evaluation result of our method with Service Cutter (Gysel et al., 2016) and API (Baresi et al., 2017) evaluated by Fritzsch (2018), which is shown in Tables 8, 9, and 10.

From a comparative perspective, both the dataflow-driven method (Table 8) and API analysis (Table 10) approaches yield four microservice candidates, smaller than the three microservices identified by Service Cutter (Table 9) approach. "Small" and "autonomous" are considered to be the most important characteristics of microservices (Jaramillo et al., 2016). Moreover, the "small" microservices obtained by the dataflow-driven method are observed to be "appropriate" when taking coupling and cohesion metrics into account. As displayed by Tables 8, 9 and 10, microservices identified by the dataflow-driven method, Service Cutter and API analysis approaches all have similar Afferent Coupling (*Ca*) on average. Both the dataflow-driven method and the API analysis deliver microservice candidates with the same average Efferent Coupling (*Ce*), which is slightly higher than the ones from Service Cutter. In addition, the Instability (*I*) of microservices from these three approaches is also of no much difference, for it is calculated using the value of *Ca* and *Ce*. As for the Relational Cohesion (*RC*) metric, the dataflow-driven method performs better than API analysis method, while less desirable than Service Cutter. The unfavourably low cohesion value of decomposition by API analysis method might indicate a limitation of the API analysis decomposition approach, which relies on well defined and described interfaces with meaningful names.

Although the dataflow-driven method does not recommend microservices with the same cohesion as the candidates from Service Cutter, it is considered to be reasonable and shows superiorities in the following aspects:

Replicable. Service Cutter much relies on the criteria pre-defined to determine the characteristics of each entity, and it also uses an undirected weighted graph for decomposition. However, the weight of the graph is generated in a subjective manner as well, which may lead to inaccurate description of the relationships

between services. Compared with this decomposition method, our approach minimizes the dependence on designer's decision, for it is based on DFDs which strictly describes the real business logics and data flow of systems.

Easier to operate. The dataflow-driven approach is based on the DFD_{PS} construction which is easy to build from the detailed DFD of systems' requirements and business logics with the help of reverse engineering technologies, e.g., MoDisco[3]. Moreover, algorithms are also provided to enable automated condensing of the DFD_{PS} into the decomposable DFD and then identifying microservice candidates. However, Service Cutter requires a detailed and exhaustive specification of the system, together with ad-hoc and subjective specification artifacts associated with the coupling criteria (Gysel et al., 2016).

Easier to understand. The decomposition approach proposed in this paper returns the microservice candidates with the style of "processes and the related data stores" and even the data interaction among processes and data stores, which is clear and easy to understand, and facilitates the further development. More importantly, by largely inheriting relevant semantic meanings from the original dataflow, our decomposition approach can conveniently generate lightweight descriptions for the resulting microservices (Fernández-Villamor et al., 2010). On the contrary, from Service Cutter's result, it is difficult to understand the content of microservices and know how to design them, particularly with data from different entities binding together.

6. Discussion

The dataflow-driven approach we proposed has shown a series of advantages over other methods (cf. Section 5). However, this approach is not perfect and also has a few potential limitations which may threaten the service decomposition. This section discusses some limitations of our work at this stage.

Since our method heavily rely on the detailed DFDs at different levels, ensuring the quality of DFDs is critical for the decomposition results. To overcome this threat, analysts must carefully and painstakingly review every process, external entity, data flow, and data store on all DFDs by hand or using some automatic tools, to make sure that they have a consistent viewpoint and the decomposition are appropriately balanced.

Apart from the relationships between processes and data stores, there may be other factors impacting whether processes should be merged into one microservice, e.g., the frequency of communication among processes, the execution time of processes. However, these kind of factors can only be collected through running the system for a period of time. In the future, we plan to implement the microservice-based Cargo Tracking System designed by this study and improve the current method by considering other factors based on runtime data monitoring and collection. Also, few Non-Functional Requirements (NFRs) are clearly claimed in this study except reusability. To eliminate this limitation, other important factors such as performance, will be incorporated into the integration of microservices later on the base of the small-grained results of our approach.

Efficiency may be another limitation of our work in this stage, though efficiency of design does not attract much attention of researchers when addressing the decomposition problems of microservices. For example, Service Cutter requires a detailed and exhaustive specification of the system, together with ad-hoc and subjective specification artifacts associated with coupling criteria (Gysel et al., 2016). This process may take a long time and be inefficient. However, we still treat this as part of the limitation of our work at the current stage. To migrate the limitation and improve the efficiency of our method, the reverse engineering tool, MoDisco, is used to help automatically generate the DFDs required. Moreover, algorithms are also provided to help automatically condensing the DFD_{PS} into the decomposable DFD and identifying microservice candidates from it. In the future, we aim to provide a fully automated dataflow-driven decomposition mechanism, including the automated generation, analysis and decomposition of DFDs.

Limitations not specific to the dataflow-driven approach in this paper are twofold. First, there is no comprehensive, well-known metrics and (simulation) methods to support the evaluation of the decomposition results, especially for the design phase. To the best of our knowledge, Sonargraph Architect is a promising tool for this purpose. However, there are also aspects that may affect the validity of the evaluation, for example, dependencies between packages might not correlate 1:1 with the interfaces to be defined between the services. Second, we lack appropriate microservice cases as benchmarks to run experiments and compare the results. Although an industry case study in a large organization is important for validation of a single approach, decomposition and evaluation of such systems are time-consuming and costly. Under this circumstance, an open-source large dataset of microservices can act as a gold-standard for current and future research in the field.

7. Conclusion

In order to tackle the challenges in migrating monolith to microservices architecture, we proposed a top-down decomposition approach driven by dataflow of business logics. We defined our decomposition as three phases: first, use case specification and business logics are analyzed on the basis of requirements; second, the detailed DFDs at different levels and the corresponding DFD_{PS} are constructed from business logics on the basis of requirement analysis; third, we designed an algorithm to automatically condense the DFD_{PS} to a decomposable DFD, in which the *sentences* between processes and data stores are combined; last but not least, microservice candidates were identified and extracted automatically from the decomposable DFD with the style of "processes + data stores". We also selected the Cargo Tracking System, one typical case decomposed by other related studies, for the application and validation of our decomposition approach. After getting the decomposition results of the Cargo Tracking System, we refer to a set of evaluation metrics and also provide a comparison against an existing microservice-oriented decomposition methods–Service Cutter (Gysel et al., 2016) and API analysis (Baresi et al., 2017), for evaluation. The evaluation and comparison all illustrate that our dataflow-driven approach is convenient to deliver reasonable, replicable and understandable microservice candidates.

Although our dataflow-driven approach has demonstrated the intended effect on the microservice-oriented decomposition, some limitations may also threaten the decomposition procedure (cf. Section 6). In future, we will conduct research on possible directions to make up those limitations, e.g., exploring more factors that determine whether a processes should be included or excluded from one microservice and further adjusting the granularity of microservices by taking more NFRs into consideration.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 61572251).

[3] http://www.eclipse.org/MoDisco/.

Appendix A

Figs. A.1 and A.2

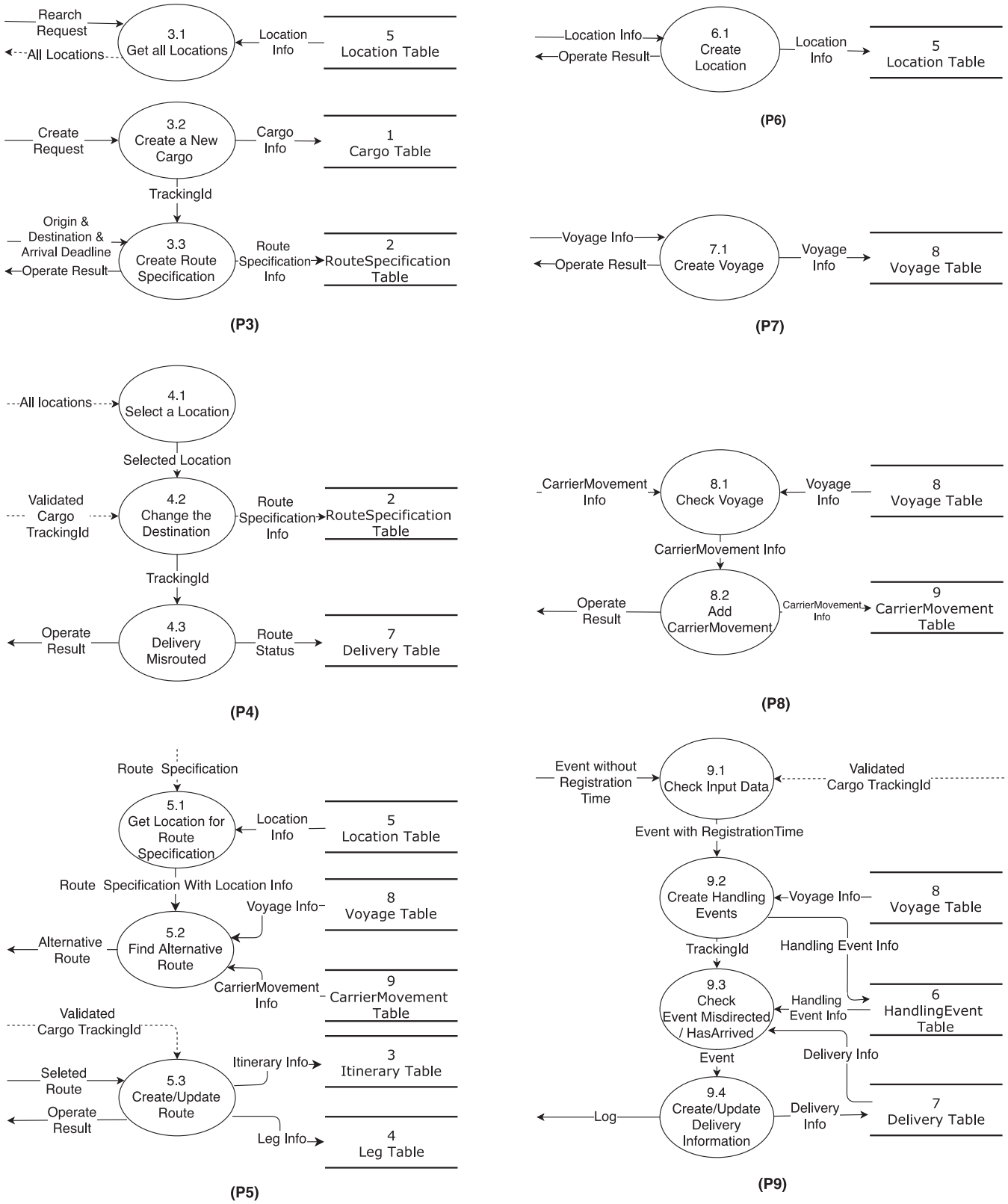

Fig. A.1. The level 1 DFD of the Cargo Tracking System (P3-P9).

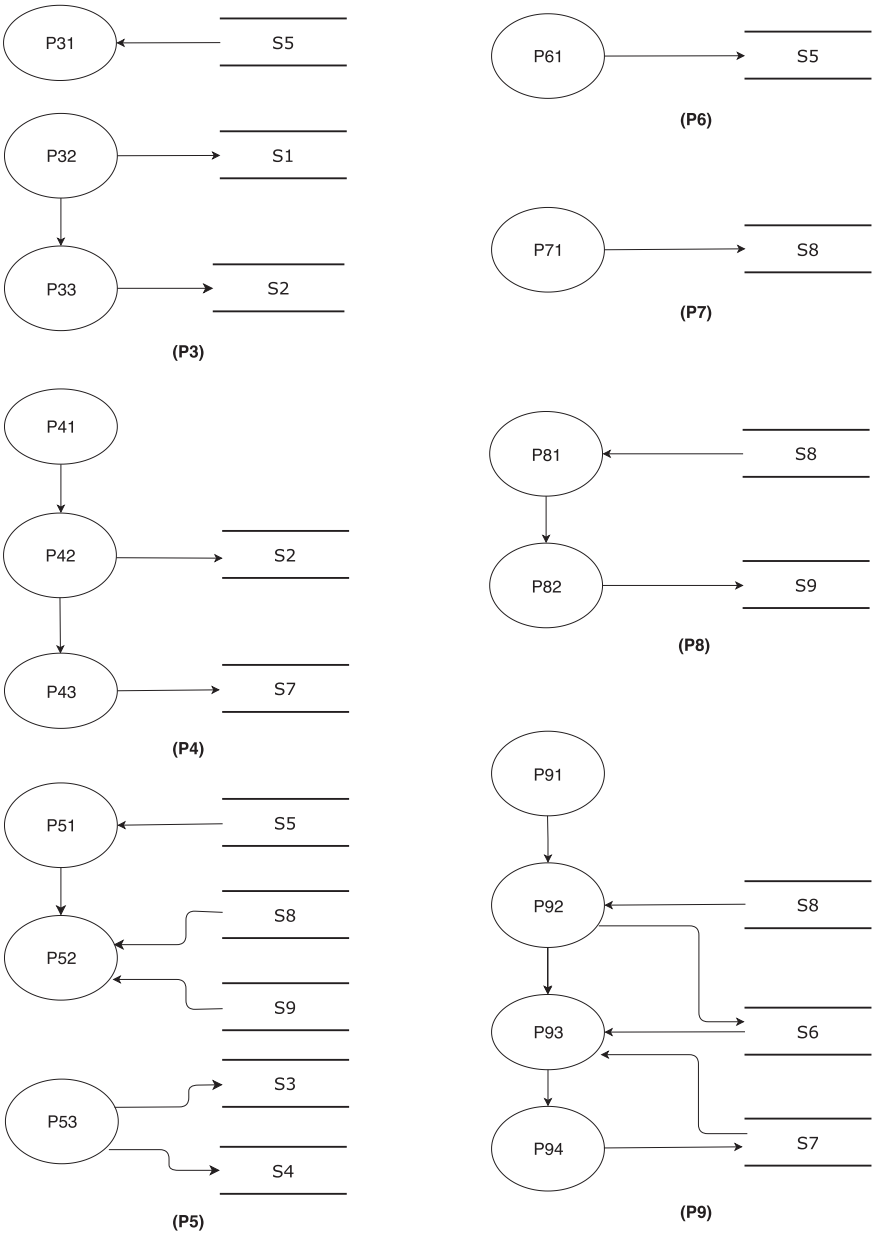

Fig. A.2. The purified level 1 DFDs of the Cargo Tracking System (P3-P9).

References

Abbott, M.L., Fisher, M.T., 2009. The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise, 1st Addison-Wesley Professional, Boston, Massachusetts.

Adler, M., 1988. An algebra for data flow diagram process decomposition. IEEE Trans. Softw. Eng. 14 (2), 169–183.

Alshuqayran, N., Ali, N., Evans, R., 2016. A systematic mapping study in microservice architecture. In: Proceedings of the 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA). IEEE, pp. 44–51.

Arndt, T., Guercio, A., 1992. Decomposition of data flow diagrams. In: Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering. IEEE, pp. 560–566.

Baresi, L., Garriga, M., De Renzis, A., 2017. Microservices identification through interface analysis. In: European Conference on Service-Oriented and Cloud Computing. Springer, pp. 19–33.

Chen, R., Li, S., Li, Z., 2017. From monolith to microservices: A dataflow-driven approach. In: Proceedings of the 2017 24th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 466–475.

Constantine, L.L., Yourdon, E., 1979. Structured design: fundamentals of a discipline of computer program and systems design, 1st Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Evans, E., 2002. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Evans, E., 2004. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, Boston, USA.

Fernández-Villamor, J.I., Iglesias, C., Garijo, M., 2010. Microservices - Lightweight service descriptions for REST architectural style. In: Proceedings of the 2nd International Conference on Agents and Artificial Intelligence. INSTICC, Institute for Systems and Technologies of Information, Control and Communication, pp. 576–579 .

Fowler, M., Lewis, J., 2014. Microservices. https://martinfowler.com/articles/microservices.html.

Francesco, P.D., Malavolta, I., Lago, P., 2017. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA). IEEE, pp. 21–30.

Fritzsch, J., 2018. From Monolithic Applications to Microservices: Guidance on Refactoring Techniques and Result Evaluation. Master's thesis; Reutlingen University.

Gane, C., 1978. Structured Systems Analysis: Tools and Techniques, Englewood Cliffs. Prentice Hall, New Jersey, USA.
Gysel, M., Kölbener, L., Giersche, W., Zimmermann, O., 2016. Service Cutter: A systematic approach to service decomposition. In: Proceedings of the European Conference on Service-Oriented and Cloud Computing. Springer, pp. 185–200.
Hassan, S., Ali, N., Bahsoon, R., 2017. Microservice ambients: An architectural meta-modelling approach for microservice granularity. In: Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA). IEEE, pp. 1–10.
Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L.E., Pahl, C., Schulte, S., Wettinger, J., 2017. Performance engineering for microservices: research challenges and directions. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. ACM, pp. 223–226.
Ibrahim, R., Siow, Y.Y., 2011. A formal model for data flow diagram rules. ARPN J. Syst. Softw. 1 (2), 60–69.
Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S., 2018. Microservices: The journey so far and challenges ahead. IEEE Softw. 35 (3), 24–35.
Jaramillo, D., Nguyen, D.V., Smart, R., 2016. Leveraging microservices architecture by using Docker technology. In: Proceedings of the SoutheastCon 2016. IEEE, pp. 1–5.
Kecskemeti, G., Marosi, A.C., Kertesz, A., 2016. The ENTICE approach to decompose monolithic services into microservices. In: Proceedings of the 2016 International Conference on High Performance Computing Simulation (HPCS). IEEE, pp. 591–596.
Larman, C., 2012. Applying UML and patterns: an introduction to object oriented analysis and design and interative development. Pearson Education India.
Mardukhi, F., Nematbakhsh, N., Zamanifar, K., Barati, A., 2013. QoS decomposition for service composition using genetic algorithm. Appl. Soft Comput. 13 (7), 3409–3421.
Martin, R.C., 2002. Agile software development: principles, patterns, and practices. Prentice Hall, New Jersey, USA.
Mazlami, G., Cito, J., Leitner, P., 2017. Extraction of microservices from monolithic software architectures. In: Proceedings of the 2017 IEEE International Conference on Web Services (ICWS). IEEE, pp. 524–531.
Messina, A., Rizzo, R., Storniolo, P., Tripiciano, M., Urso, A., 2016. The database-is-the-service pattern for microservice architectures. In: Proceedings of the International Conference on Information Technology in Bio-and Medical Informatics. Springer, pp. 223–233.
Newman, S., 2015. Building microservices, 2nd O'Reilly Media, Sebastopol, California.
Richards, M., 2015. Microservices vs. service-oriented architecture. O'Reilly Media, Sebastopol, California.
Richardson, C., 2015. Introduction to Microservices. https://www.nginx.com/blog/introduction-to-microservices/.
Richardson, C., 2018a. Pattern: Microservice Architecture. http://microservices.io/patterns/microservices.html.
Richardson, C., 2018b. Pattern: Monolithic Architecture. http://microservices.io/patterns/monolithic.html.
Rosenblatt, H.J., 2013. Systems analysis and design. Cengage Learning, Boston, MA, USA.
Soldani, J., Tamburri, D.A., Van Den Heuvel, W.-J., 2018. The pains and gains of microservices: a systematic grey literature review. J. Syst. Softw. 146, 215–232.
Sun, S.X., Zhao, J., 2012. A decomposition-based approach for service composition with global QoS guarantees. Inf. Sci. 199, 138–153.
Tyszberowicz, S., Heinrich, R., Liu, B., Liu, Z., 2018. Identifying microservices using functional decomposition. In: International Symposium on Dependable Software Engineering: Theories, Tools, and Applications. Springer, pp. 50–65.
Vresk, T., Cavrak, I., 2016. Architecture of an interoperable IoT platform based on microservices. In: Proceedings of the International Convention on Information and Communication Technology, Electronics and Microelectronics, pp. 1196–1201.
Zhao, Y., Si, H., Ni, Y., 2009. A service-oriented analysis and design approach based on data flow diagram. In: Proceedings of the 2009 International Conference on Computational Intelligence and Software Engineering. IEEE, pp. 1–5.
Zimmermann, O., 2017. Microservices tenets: agile approach to service development and deployment. Comput. Sci. 32 (3), 301–310.

Shanshan Li is a Ph.D candidate at the Software Institute of Nanjing University, China. She received her M.E. by Research from the China University of Petroleum (East China), Qiangdao, China in 2016. She is the author or co-author of several international journal and conference publications. Her research interests include empirical software engineering, software architecture, DevOps and microservices.

He Zhang is a professor of software engineering in the Software Institute at Nanjing University, China. He joined academia after seven years within industry, developing software systems in the areas of aerospace and complex data management. Dr. Zhang received his Ph.D in computer science from the University of New South Wales (UNSW), and has published 100+ peer-reviewed research papers in the high quality international journals, conferences, and workshops. He undertakes research in software engineering, in particular software process (modeling, simulation, analytics and improvement), software quality, empirical and evidence-based software engineering, and service-oriented computing. Dr. Zhang is a member of the IEEE Computer Society and the ACM (SIGSOFT), and serves on the Steering Committees, Program Committees, and Organizing Committees of a number of premier international conferences in software engineering community.

Zijia Jia is a Master by Research Student at the Software Institute of Nanjing University, China. He received his Bachelor Degree in Software Engineering from the Northeastern University, Shenyang, China in 2018. His research interests include software architecture, DevOps and microservices.

Zheng Li is an assistant professor at the Department of Computer Science at University of Concepcin, Chile. He received his Ph.D. degree and M.E. by Research from the Australian National University (ANU) and the University of New South Wales (UNSW) respectively. During the same time, he was a graduate researcher with the Software Systems Research Group (SSRG) at National ICT Australia (NICTA). Before moving abroad, he had around four-year industrial experience in China after receiving his M.Sc. from the Beijing University of Chemical Technology and the B.Eng. from the Zhengzhou University. His research interests include Cloud computing, performance engineering, empirical software engineering, software cost/effort estimation, and Web service composition.

Cheng Zhang is an Associate Professor in the School of Computer Science and Technology, Anhui University, China. And also he is the vice chair of Software Engineering Department in the school. He is the member of the Technical Committee on Software Engineering, China Computer Federation. He received Ph.D. degree from Durham University, UK in 2011. Then he joined Anhui University under the universitys talent introduction scheme in 2012. Chengs main research interests include the study of empirical software engineering, the use of evidence-based software engineering techniques, microservices and cloud workflow. His research has been published on the journal of IEEE Transactions on Software Engineering. He has received research funding from the National Natural Science Foundation of China (NSFC), Ministry of Education of China and Anhui Provincial Natural Science Foundation.

Jiaqi Li is a Master by Engineering Student at the Software Institute of Nanjing University, China. He received his Bachelor Degree in Software Engineering from the Hefei University of Technology, Hefei, China in 2018. His research interests include software architecture and microservices.

Qiuya Gao is a Master by Engineering Student at the Software Institute of Nanjing University, China. She received her Bachelor Degree in Software Engineering from the Nanjing University Jinling College, Nanjing, China in 2016. Her research interests include software architecture and microservices.

Jidong Ge is an Associate Professor at Software Institute, Nanjing University, China. He received his Ph.D. degree in Computer Science from Nanjing University in 2007. His current research interests include cloud computing, workflow scheduling, software engineering, workflow modeling, process mining. His research results have been published in more than 60 papers in international journals and conference proceedings including IEEE TSC, JASE, COMNET, JPDC FGCS, JSS, Inf. Sci., JNCA, JSEP, ESA, ICSE, IWQoS etc.

Zhihao Shan (Mark Shan), is a member of Tencent Technical Committee and Tencent Open Source Office. He has nearly 15 years' experience in research, development, operation and management of large-scale Internet systems. He has worked in Baidu, Sina and Tencent. His responsibilities in Tencent include managing the operating system, application security, global load balancing platform, messaging middleware, public login, process system, operating platform, operational planning, resources and budget management, etc. Recently, he is committed to the construction of open source ecosystems and has promoted TARS, a ten-year supporting system of Tencent for microservices' development, to be open source in Linux Foundation. As the manager of TARS, Shan is working on the development of TARS open source technology and the operation of its community. He has contributed to developing TARS into a multi-language supporting framework for microservices' development with high performance, e.g., C++, Java, Go, Node.js, PHP, Python, and .Net Core. He is also an expert of CloudNative, Microservices and DevOps standards in China, and is involved in the development and application of microservices and DevOps standards in the industry.