A Multi-Model based Microservices Identiftcation Approach

Mohamed Daoud[1], Asmae El Mezouari[2], Noura Faci[1], Djamal Benslimane[1]

Zakaria Maamar[3] and Aziz El Fazziki[2]

[1] Claude Bernard Lyon 1 University, Lyon, France
[2] Caddi Ayyad University, Marrakesh, Morocco
[3] Zayed University, Dubai, U.A.E

Microservices are hailed for their capabilities to tackle the challenge of breaking monolithic business systems down into small, cohesive, and loosely-coupled services. Indeed, these systems are neither easy to maintain nor to replace undermining organizations' efforts to cope with user's changing needs and governments' complex regulations. Microservices constitute an architectural style for developing a new generation of systems as a suite of services that, although they are separate, engage in collaborative execution and communication sessions. However, microservices success depends, among many other things, on the existence of an approach that would automatically identify the necessary microservices according to organizations' requirements. In this paper, we present such an approach and demonstrate its technical doability in the context of a case study, Bicing, for renting bikes. Some salient features of this approach are business processes as input for the identification needs, three models known as control, data, and semantic to capture dependencies between these processes' activities, and, finally, a collaborative clustering technique that recommends potential microservices. Conducted experiments in the context of Bicing clearly indicate that our approach outperforms similar ones for microservices identification and reinforce the important role of business processes in this identification. The approach constitutes a major milestone towards a better architectural style for future microservices systems.

Keywords: Business process · Control/Data/Semantic dependency · Clustering · Microservice.

1 Introduction

One of the many challenges that today's organizations have to tackle is how to adjust their business functions in response to users' new needs, competitors' new products, authorities' new regulations, to mention just some. Decomposing existing applications is becoming a critical need for modern organizations [5, 35]. Indeed, despite the good will of organizations to remain proactive, the monolithic nature of the information systems (systems for short) associated with their business functions, complicates the work of these organizations. Monolithic means one "sealed" block that encompasses strongly-coupled components that are neither easy to replace nor to maintain. Many stories report on how organizations, e.g., Best Buy, Cloud Elements, and Wix.com, had to "wrestle" with monolithic systems[4].

Over the years, many Information and Communication Technologies (ICT) were put forward to address the ongoing issue of monolithic systems including Service-Oriented Architecture (SOA) [8], Component-Based Software Engineering (CBSE) [23], and Commercial-Of-The-Shelf (COTS) [28]. However, many of these ICT did not live up to their advocates' expectations overlooking factors like resistance to change, legacy practices, and biased advices, and creating more confusion about the best way to move forward. Lucklily new ICT regularly surface exemplified, in this paper, with microservices [18]. The term microservice was coined in 2014 and constitutes an architectural style for developing applications as a suite of small services, having each a separate but collaborative execution and communication process [6]. Some adopters of microservices are Amazon and Netflix. In a recent post by NGINX[5], Netflix shares its successful experience of transitioning "*from a traditional development model with 100 engineers producing a monolithic DVD-rental application to*

[4] tinyurl.com/y6ko2k8o.

[5] tinyurl.com/ojm9zgp.

a microservices architecture with many small teams responsible for the end-to-end development of hundreds of microservices that work together to stream digital entertainment to millions of Netflix customers every day".

It is clearly difficult to easily compare microservices architectures and Service oriented architectures (SOA) as they both achieve similar objectives and adopt closed principles. SOA enables separate applications to communicate with one another by allowing them to expose their services through standardized interfaces. Microservices scope is concerned by the application itself and focuses on the structure within the application. Microservices corresponds to an architectural style for building an application with the objective to separate and decouple components within an application boundary. The separation between identified components is absolute and each component encapsulates its own database. Indeed, SOA and microservices are complementary. An existing application can expose its services through an SOA level. It can also be restructured and migrated to microservices-based architectures. This latter can be then expose its services by using SOA level. All well established techniques of SOA (Interfaces description, service discovery, etc.) can then be used after a microservices-based architecture exposes its services and not during the reorganization of a monolithic application into a microservices-based application.

In fact, this is what we did and demonstrated in [10] where we designed and developed a collaborative clustering-based approach to identify microservices from a set of Business Processes (BPs). To the best of our knowledge, only Amiri [2] identifies microservices using BPs while the rest use UML class diagrams [4], legacy databases [7], log files [12], and source codes [13]. At the core of our collaborative clustering-based approach, that we extend in this paper, are first, activities of BPs that indicate who does what, when, where, and why, second, models that capture structural, data, and semantic dependencies between these activities, and finally, a collaborative clustering technique that combines these models. This technique allowed us to apply separate techniques to the three dependency models prior to consolidating their respective results into a consensual solution.

This paper[6] extends the approach we discussed in [10] from three perspectives. First, we add a new semantic dependency model to the existing structural- and data-dependency models. The semantic model captures the semantic relationships that could exist between a BP's activities using three techniques. The first technique is word-driven and computes the similarity between activities based on their names. The second technique is concept-driven and also computes the similarity between activities using the most similar concepts that would belong to the BP's domain ontology. Finally, the third technique is fragment-driven and, like the previous two techniques, computes the similarity between activities using a set of the most similar, eventually overlapped, fragments that each encompasses concepts belonging to the BP's domain ontology. All these techniques are formalized and then, described with their respective algorithms. On top of the semantic dependency model, we provide more details and algorithms about the collaborative clustering technique that underpins our multi-model based microservices identification approach. Last but not least, we present new experimental results that take into account the new semantic dependency model with its different techniques.

The rest of this paper is organized as follows. Section 2 presents some related works. Section 3 discusses a case study, provides an overview of our approach to automatically identify microservices from BPs, and finally, formalizes the different models associated with this automatic identification. Section 4 presents the collaborative clustering technique that underpins our identification approach and then, discusses the implementation efforts and experimental results. Section 5 concludes the paper and presents future work.

2 Related work

There exist a good number of works that discuss monolithic systems' limitations and how microservices could address these limitations [31]. Such systems are known for incurring significant development, maintenance, and evolution costs [32]. We discuss in this section the works related to microservices identification, which constitutes a critical step when migrating from monolithic applications to microservices-based applications.

In [1], Ahmadvand and Ibrahim propose a microservices decomposition methodology that maps functional requirements onto microservices with taking into account non-functional requirements, mainly security and scalability. The application to decompose is expressed as a set of functional requirements where each corresponds to a functionality of the system. The requirements are complemented by a set of security requirements captured by the use of misuse cases. Security requirements elicitation mechanism is proposed to identify security policies that correspond to each functional requirement.

[6] This work was supported by the French research association (ANRT)[grant number: 2018/0216]

The scalability level (High, Medium, Low) required for each functional requirement is also given. A requirement reconciliation algorithm based on rules is proposed where scalability and security are used as balancing factors for composing a microservice.

In [2], a clustering-based approach is discussed where structural and data dependencies between tasks are extracted from a given set of BPs. All these dependencies are merged into one dependency matrix that is, then, submitted to a classical clustering algorithm to identify candidate microservices. As stated in Section 1, we adopted BPs like in [2], but, considered more dependencies and a mix of clustering algorithms.

In [3], Barbosa et al. propose a process to identify candidate microservices from a set of business rules implemented as a set of stored procedures. There are three main steps to support this identification. The first one discovers the stored procedures related to the requirements and their corresponding system features. An expert is in charge of indicating the requirements. The second step analyzes the discovered stored procedures to identify the candidate microservices. The last step evaluates the source code of all these candidate microservices so, that, unnecessary ones are dropped while refactoring others if deemed necessary. The proposed approach is even relevant for exploring and exploiting stored procedures artifacts and can be extended to include other artifacts (triggers, view, etc.). The approach, unfortunately, remains manual relying heavily on expert intervention which could be time consuming, tedious, and prone to errors.

In [4], Baresi et al. proposed an approach to identify microservices. It uses the semantic similarity of functionalities described in openAPI[7] and a reference vocabulary. The necessary microservices are identified as a cohesive cluster of operations extracted from an UML diagram class while the semantic similarity between functionalities' names is based on the pre-computed database DISCO (DIStributionally related words using CO-occurences)[8].

In [26], Li et al. propose a data flow-driven approach for the microservice-oriented decomposition of existing applications. The input of this approach is a set of Data Flow Diagrams (DFD) where each is generated from a business logic and describes the data flow through business processes. A DFD is represented in terms of processes/operations, data store, data flow, and external entities. The DFDs are transformed into sentences to make them machine-readable. A sentence is in the form of $(I \rightarrow 0)$ meaning that there is a connection between input I and output O. Inputs and outputs could be a process/operation, a data store, or an external entity. A clustering algorithm is then used to cluster sentences that include the same data stores to form a {microservice candidate.

In [16], Gysel et al. suggest Service Cutter framework as a systematic approach to service decomposition. The framework constitutes a process for identifying a set of services and assigning all nanoentities (data, operation, and artifact) to one and only one of these services so, that, service-coupling criteria are adopted. Sixteen of these criteria have been proposed in terms of requirements and have been divided into four categories: cohesiveness, compatibility, constraint, and communication. The input to Service Cutter corresponds to various specifications of software engineering artifacts including use cases, entity-relationship models, security access-groups, and separated security zones. These specifications are represented as a set of System Specification Artifacts (SSAs). These SSAs instances are used to extract from them nanoentities as well as coupling criteria instances. An un-directed and weighted graph is proposed to represent the link between nanoentities (nodes of the graph). A weight link between two nodes indicates to what extent two nanoentities are cohesive and/or coupled. A clustering algorithm is finally used on the produced graph to generate candidate microservice cuts.

In [19], a functionality-oriented microservice extraction by clustering execution traces of programs collected at runtime, is described. These traces are collected by using techniques of program execution monitoring and are used to collect implicit and explicit programs' functional behaviors. The traces also reveal which entities are used for which business logic. The approach clusters source code entities that are related to the same functionalities. Even if the work in [19] is interesting, it suffers from its strong dependence on the quality of the generated execution traces, and consequently on the quality of the test cases.

In [21], Knoche et al. refer to industrial case studies to stress out, in the context of monolithic system shift to microservices-based systems, the importance of separating client-associated internal modules migration from platform migration as several challenges still lie ahead. They, then, present an incremental migration process to gradually decompose an application into microservices by exploiting existing source codes. An important step in the approach is to define from scratch an external and domain-oriented service facade to capture the main functionalities required by the client. All service

[7] www.openapis.org.

[8] www.commonspaces.eu/en/oer/disco-extracting-distributionally-related-words-us.

operations are defined from a targeted domain model. Static analysis technique is also used to identify the entry points accessed by other applications. Another step is dedicated to service operation implementation and the client application migration to service facade. The existing accesses are replaced by the service invocations. Even the approach has no foundations, it has the advantage to be applied and seems to be in production and implementations were developed for some service operations.

In [25], microservices were proposed as potential candidates to modernize a monolithic system that was described using three types of objects: interfaces, business functions, and database tables. These objects are then linked in a dependency graph using calls from interfaces to business functions, calls between business functions themselves, and accesses from business functions to database tables. Potential microservices correspond to the business rules that depend on database tables, and correspond to the facades connected to the database tables.

Table 1 presents a concise summary of the approaches presented above. We use five criteria to categorize them: main inputs, data modeling, identification algorithm, evaluation method that indicates whether the experiments adopted industrial application or case-study, and metrics used to evaluate the performance of the approach. When we examined the approaches presented above, we paid attention to initial inputs that could be business processes, API specifications, UML diagrams, domain ontologies, etc. Our approach considers business processes as a main input. In term of data modeling, other approaches use dependencies between artifacts extracted from the inputs. Such modeling is often based on graphs and relational data (matrices). Our approach considers relational data due to their simple nature. It also seems from a global perspective that clustering algorithm is often used to identify microservices. While the studied approaches adopt the same, we made this algorithm collaborative. Finally, most approaches rely on case studies to implement their solutions. A few works implemented their solution on industrial applications. We considered two case studies, one of them was used to compare the performance of our approach to others. Different metrics are used in our work: Dunn Index, Afferent Coupling, Efferent Coupling, Instability, and Relational Cohesion.

3 Our approach for identifying microservices

This section consists of five parts. The first part presents a case study that refers to Barcelona's bike sharing system known as Bicing. The second part discusses our approach's foundations in terms of dependencies between BPs' activities and collaborative clustering. Finally, the last parts formalize these dependencies.

3.1 Bicing case-study

Bicing is a system for renting bikes in the city of Barcelona with 400 anchor stations and 6000 bikes. Bicing's monolithic architecture is described in [14] along with the managerial and technical challenges, like coordinating large development teams and scaling application programs, that undermine its operations. For the needs of our work, we designed Fig. 1 that corresponds to a high-level BPMN representation of some key activities (a_i), gateways, dependencies, and artefacts associated with Bicing.

Table 1: Synthesis view of microservices identification approaches

Approach	Main Input (starting points)	Data modeling	Identification algorithm	Evaluation	# of metrics	Metric details
[1]	Functional and non functional requirements specifications	Dependency weight between couple of functions	Rule-based algorithm	1 case study: online movie streaming system	0	No metrics
[2]	Business Processes(BPs)	Data and control matrices dependencies	Centralized clustering	1 case study: Plan Approval BPMN process	1	Accuracy
[3]	System requirements and existing stored procedures codes	Stored procedures discovery related to the system requirements	Source code analysis	Real large scale system	0	No metrics but implementation of a Proof of Concept
[4]	OpenAPI specifications of interfaces and a reference vocabulary	Co-occurrence principle based-semantic similarities between functionalities	Centralized clustering	three case studies: (1) online movie streaming system, (2) Transfer money application combined to Kanban Board application, and (3) real-world OpenAPI specifications from APIs. Guru	2	Precision/Recall
[26]	Data Flow Diagram sentences	dependency graph between process, data stores and external entities	Rule-based algorithm	1 case study: cargo tracking	4	Afferent Coupling, Efferent Coupling, Instability of coupling, Relational Cohesion
[16]	Use cases, Entity-Relationship Models, security access groups, separated security zones, etc.	Dependency graph between nanoentities	Centralized clustering	two case studies: cargo tracking and Trading system	1	Microservice candidates rating (Excellent, Expected, Unreasonable)
[19]	Log of execution traces of programs	Links between source codes and business logics	Centralized clustering	four case studies (web applications): e-commerce, discussion board system, blogging system,blog sites	5	Functional Cohesion (Cohesion at Domain level, Cohesion at Message level), Functional Coupling (Interface Number, Operation Number, Interaction Number)
[21]	Source code, domain model	nothing special	Static analysis	Industrial application: customer management application of an insurance company	0	-
[25]	Objects including interfaces, business functions, and database tables	dependency graph of object calls	Rules-based algorithm	industrial application: Bank transactions management	0	-
Our work	Business Processes(BPs)	Data, control, and semantic matrices dependencies	Collaborative clustering	two case studies: Bike rental and cargo tracking	5	Dunn Index, Afferent Coupling, Efferent Coupling, Instability of coupling, Relational Cohesion

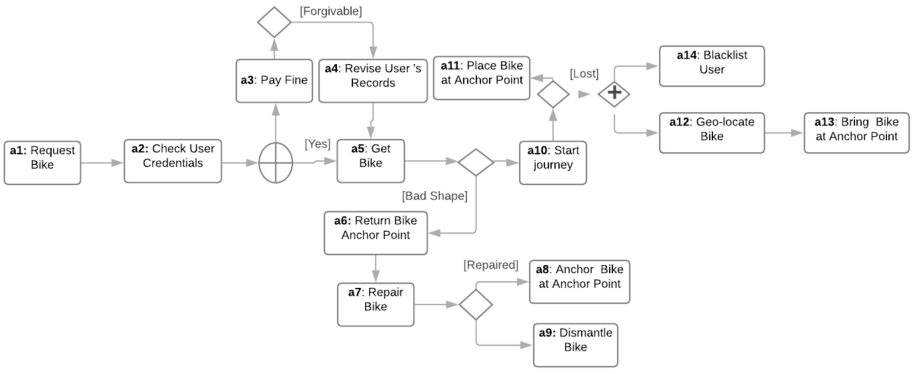


Fig. 1: An illustrative BPMN representation of Bicing

It all starts when a user requests a bike (a_1) at a certain anchor station. After checking the user's credentials (a_2) and any late fee payment (a_3), Bicing updates the user's records (a_4) and then, approves the user's request (a_5). If it turns out that the bike is defective, the user puts it back (a_6) and eventually requests another one. Otherwise, the user starts his journey (a_{10}). Regularly all bikes are serviced (a_7) leading to either putting them back for rent (a_8) or disposing them (a_9). When the user arrives to destination, he returns the bike at a certain anchor station (a_{11}). Otherwise, Bicing blacklists the user (a_{14}) due to bike inappropriate return, and geo-locates the bike (a_{12}) so, that, it is collected by the competent services and then, put back into service for other users (a_{13}).

From a specification perspective, activities ($\{a_i\}$) may require inputs ($\{i_i\}$) and produce outputs ($\{o_i\}$). We relate both inputs and outputs to specific artefacts' attributes. An activity acts upon both attributes and artefacts using *read*(r)/*write*(w) and *update*(u)/*create*(c)/*delete*(d) operations, respectively. Table 2 lists activities, artefacts, attributes of artefacts, and the operations that artefacts/attributes are subject to. For instance, a_5(get bike) applies *write* operation to User_Destination and Rent_Date attributes, which leads to executing *create* operation whose outcome is Rental artefact.

Table 2: Bicing's components

Activity	Artefacts	Attributes of artefacts
a_1	Bike (u)	Anchor_Point (r), Bike_ID (r), Bike_Status (w)
	User (u)	User_ID (r), User_Destination (r)
a_2	User (u)	User_ID (r), User_Credit (r), User_Destination (r)
a_3	User (u)	User_ID (r), User_History (r), User_Validity (r)
a_4	User (u)	User_ID (r), User_History (w)
a_5	Bike (u)	Bike_ID (r), Bike_Status (w)
	User (u)	User_ID (r), User_Status (w)
	Rental (c)	User_Destination (w), Rent_Date (w)
a_6	Bike (u)	Anchor_Point (r), Bike_ID (r), Bike_Status (w)
	User (u)	User_ID (r), User_Status (w)
	Rental (d)	Rent_ID (r)
a_7	Bike (u)	Bike_Status (w)
	Repair (c)	estimated_Repair_Cost (w), agree_Repair (w)
a_8	Bike (u)	Anchor_Point (r), Bike_Status (w)
a_9	Bike (d)	Bike_ID (r)
a_{10}	User (u)	User_ID (r), User_Status (w)
a_{11}	Bike (u)	Anchor_Point (r), Bike_ID (r), Bike_Status (w)
	User (u)	User_ID (r)
	Rental (u)	Rent_ID (r), Rent_Cost (w), User_History (w)
a_{12}	Bike (u)	Bike_ID (r), Bike_Location (w)
a_{13}	Bike (u)	Bike_ID (r), Anchor_Point (r), Bike_Status (w)
a_{14}	User (u)	User_ID (r), User_Status (w), User_History (w)

3.2 Foundations

Compared to the works presented in Section 2, BPs are our main source of identifying microservices. These ones are expected to be fine-grained, strongly cohesive (i.e., degree to which activities in a microservice belong together), and loosely-coupled (i.e., degree to which microservices can be easily replaced). According to Davenport, a BP is a set of logically related activities that are performed to achieve goals [11]. "Logically related" refers to dependencies between

activities such as *control* (with respect to an execution flow), *data* (with respect to an information flow), *semantics* (with respect to a meaningful exchange flow) and *functional* (with respect to a collaboration flow).

- *Control dependency* refers to both the execution order (e.g., finish-to-start and start-to-start) between activities and the logical operators (e.g., XOR and AND) between activities as well. Should two activities be directly connected through a control dependency, then most probably they would form a highly-cohesive microservice to which they will belong. Contrarily, they would most probably be used to form separate microservices to which each will belong.
- *Data dependency* refers to associating activities' outputs/inputs in a way that permits to illustrate data flowing from one activity to another. These inputs/outputs correspond to artefacts' attributes. Data dependency sheds light on both artefacts and artefacts' attributes that could be subject to operations illustrated in Table 2 like *create* (c) and *write* (w), respectively. In addition to input/output association, data dependency could indicate to what extent artefacts and/or artefacts' attributes are either mandatory or optional for a BP execution. We advocate that activities that exchange mandatory artefacts' attributes should be part of the same microservice allowing to avoid delaying this exchange, for example.
- *Semantic dependency* uses activities' names to establish their functional similarity in term of what they do. These names refer to linguistic templates that must include a verb and object/result along with optional parameters like time and location. For instance, a_{11}'s name includes place (verb), bike (object), and anchor point (location). To assess activities' names similarity, we rely on either a reference vocabulary or an ontology. A highest/lowest similarity value would indicate strong/weak coupling between activities making them members of the same/different microservice/microservices.
- *Non-Functional* dependency relies on non-functional requirements such as privacy, security, cost, and scalability that could impact the execution of a business process's activities. Compared to the control-dependency model that focuses on who executes what activities, when, where, and why (5 W's), the non-functional dependency model focuses on the quality of executing activities. Has a certain level of security achieved during execution, and has a certain cost be maintained are examples of questions that the non-functional dependency model could address.

We focus on *control*, *data*, and *semantic dependencies*, only, and leave *functional* dependency for the future. Upon establishing such dependencies, we quantify them using specific metrics. The objective is to evaluate cohesion and coupling among activities so, that, they are gathered in either same or separate microservices. To measure a *control dependency* between two activities (a_i, a_j), we consider a_j's occurrence probability after executing a_i. This probability depends on the execution order and/or logical operators between a_i and a_j. Let us consider the *control dependency* between a_5 and a_{10} that is connected with a_6 through XOR (Fig. 1). After executing a_5, a_{10}'s occurrence probability depends on the decision made at XOR (i.e., either a_6 or a_{10}). We note that any activity's occurrence probability is calculated over time by using the BP's execution log. To measure a *data dependency* between two activities (a_i, a_j), we consider an artefact's and attribute's criticality level that would reflect the importance of information shared between these activities. This level denotes to what extent artefact/attribute unavailability would impact the continuity of business operations. To measure a *semantic dependency* between two activities (a_i,a_j), we measure the distance between their respective semantic annotations using three annotation techniques, *word-driven*, *concept-driven*, and *fragment-driven*. The first relies on a reference vocabulary to annotate activities with terms associated with these activities' names. The second relies on a certain domain ontology to annotate activities with concepts related to their respective names. Finally, the third refines the second by annotating the activities with ontological fragments that would refer to functional domains. More details about formalizing *control*, *data*, and *semantic dependencies* are given in the next sections.

Based on dependencies among activities, we gather activities into microservices by using clustering techniques. In the literature, clustering is either centralized or collaborative [17]. In the former, a single component manages the clustering by utilizing all individuals' features[9] as inputs. In the latter, multiple components, each in charge of one type of features, exchange some details during clustering so, that, appropriate clusters are jointly built. Performance and appropriateness of clustering techniques are thoroughly discussed in the literature [9] and [34]. Many works like [9] and [17] advocate for collaborative clustering to identify microservices. It provides fine-grained and accurate results contrarily to centralized clustering where individuals' features need to be aggregated before initiating any clustering.

[9] In our work, individuals are activities and features are *control*, *data*, and *semantic* dependencies.

Fig. 2 depicts our three-step approach to identify microservices. The approach relies on the aforementioned dependencies and collaborative clustering to group activities into potential fine-grained, highly-cohesive, and loosely-coupled microservices. The steps denoted by collection, dependency analysis, and collaborative clustering proceed as follows. The collection step gathers all necessary details from the BP specification and BP engineer, per dependency type (i.e., *control*, *data*, and *semantics*). Next, the dependency analysis step examines these details per dependency type allowing to define particular measures for storage in dedicated repositories. Finally, the collaborative clustering step applies different clustering techniques (one per dependency type) to the stored measures. More details about how our collaborative clustering algorithm was defined and implemented are given in Section 4.1.

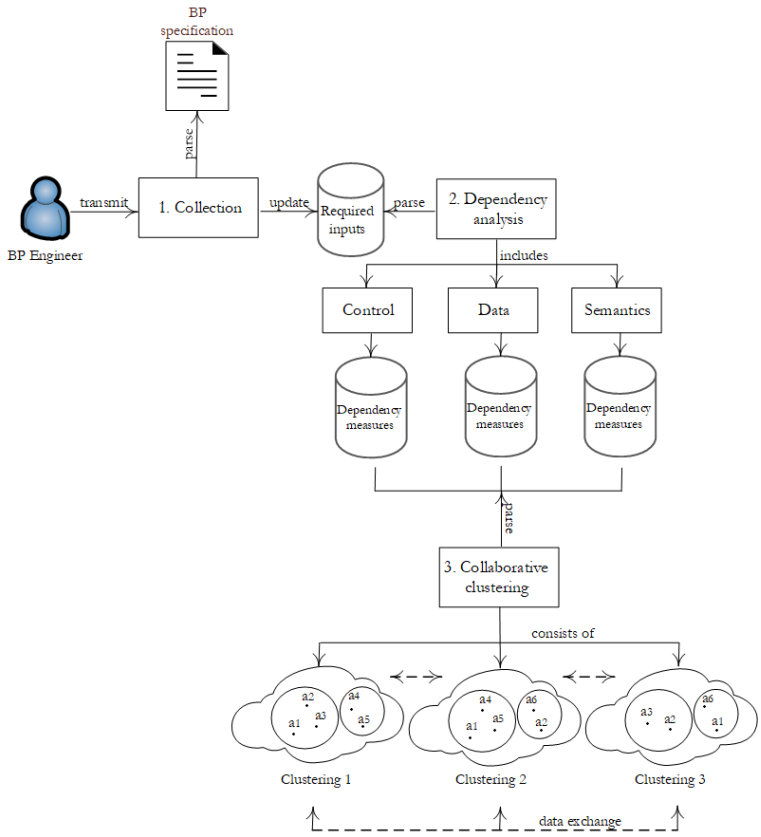


Fig. 2: General representation of our microservices identification approach

3.3 Control dependency analysis

In Section 3.2, we mention that there are two types of *control dependencies* namely, direct and indirect. The former refers to a certain execution order (ExecOrder) between a_i and a_j that **can** be connected to other activities $\{a_k\}$ through a certain Operator. An execution order between two activities could be exemplified with either *finish-to-start* (our focus and denoted by SEQ)), *finish-to-finish*, *start-to-start*, or *start-to-finish*. The latter refer to an execution path (ExecPath) with a certain execution order between a_i and a_j that involves other activities $\{a_k\}$ connected through two or more operators.

Direct. Let $\mathrm{CD}(a_i, a_j[\mathsf{Operator}\ \{a_k\}])_{\mathsf{ExecOrder}}$ be a direct *control dependency*.
We start with a simple *control dependency* that stands for $CD(a_i, a_j)_{\mathsf{SEQ}}$ (i.e., $\{a_k\} = \emptyset$)). SEQ between a_i and a_j means that a_j starts only after a_i has successfully completed (i.e., a_j's execution remains uncertain). $CD(a_i, a_j)_{\mathsf{SEQ}}$,

thus, denotes a_j's occurrence probability (p) after a_i's completion as per Equation 1:

$$CD(a_i, a_j)_{\mathsf{SEQ}} = p \tag{1}$$

where $p \in]0, 1]$.

We now examine the *control dependency* $CD(a_i, a_j \ \mathsf{Operator}\ \{a_k\})_{\mathsf{SEQ}}$ (i.e., $\{a_k\} \neq \emptyset$)). According to Operator's semantics, we assume that some r activities in $\{a_k\} \cup a_j$ will be selected for execution. Equation 2 defines the number of sets containing r activities that will be selected for execution as a combination $C(n, r)$ where n corresponds to $card(\{a_k\} \cup a_j)$.

$$C(n, r) = \frac{n\,!}{r\,! \times (n - r)\,!} \tag{2}$$

Depending on the semantics of Operator whether AND, XOR, or OR, $CD(a_i, a_j \ \mathsf{Operator}\ \{a_k\})_{\mathsf{SEQ}}$ is calculated as follows:

1. $CD(a_i, a_j \ \mathsf{AND}\ \{a_k\})_{\mathsf{SEQ}}$. This dependency means that a_j will start only after a_i has successfully completed regardless of $\{a_k\}$. Formally, Equation 3 computes $CD(a_i, a_j \ \mathsf{AND}\ \{a_k\})_{\mathsf{SEQ}}$ as follows:

$$CD(a_i, a_j \ \mathsf{AND}\ \{a_k\})_{\mathsf{SEQ}} = C(n, n) * CD(a_i, a_j)_{\mathsf{SEQ}} \tag{3}$$

   where $p \in]0, 1]$ & $C(n, n) = 1$, as per Equation 2.
2. $CD(a_i, a_j \ \mathsf{XOR}\ \{a_k\})_{\mathsf{SEQ}}$. This dependency means that one activity from $\{a_k\} \cup a_j$ will be selected after a_i has successfully completed. Formally, Equation 4 computes $CD(a_i, a_j \ \mathsf{XOR}\ \{a_k\})_{\mathsf{SEQ}}$ as follows:

$$CD(a_i, a_j \ \mathsf{XOR}\ \{a_k\})_{\mathsf{SEQ}} = \frac{1}{C(n, 1)} * CD(a_i, a_j)_{\mathsf{SEQ}} \tag{4}$$

   where $C(n, 1)$ is the number of possibilities to select one activity from $\{a_k\} \cup a_j$. As per Equation 2, $C(n, 1)$ is equal to n.
3. $CD(a_i, a_j \ \mathsf{OR}\ \{a_k\})_{\mathsf{SEQ}}$. This dependency means that a set of r activities from $2^{\{a_k\} \cup a_j}$ (i.e., all possible multiple choices) will be selected after a_i has successfully completed. For the sake of simplicity, we assume that any activity in $\{a_k\} \cup a_j$ has the same occurrence probability over $2^{\{a_k\} \cup a_j}$, that is equal to $\frac{r}{n}$ where r varies from 1 to n. Formally, Equation 5 computes $CD(a_i, a_j \ \mathsf{OR}\ \{a_k\})_{\mathsf{SEQ}}$ as follows.

$$CD(a_i, a_j \ \mathsf{OR}\ \{a_k\})_{\mathsf{SEQ}} = \frac{\sum_{r=1,n}(\frac{r}{n} \times C(n, r))}{\sum_{r=1,n} C(n, r)} * CD(a_i, a_j)_{\mathsf{SEQ}} \tag{5}$$

   where
   - $\sum_{r=1,n}(\frac{r}{n} \times C(n, r))$ represents the number of a_j's occurrences among possible combinations of activities[10].
   - $\sum_{r=1,n} C(n, r)$ corresponds to the total number of possible combinations of activities[11].

Indirect. Let $CD(a_i, a_j)_{\mathsf{ExecPath}}$ be an indirect *control dependency* between a_i and a_j. We start with a simple *control dependency* that stands for $CD(a_i, a_j)_{\mathsf{Path}_{i,j}}$ where $\mathsf{Path}_{i,j}$ refers to a single set of other peers ($\{a_k\}$) connected with operators. Here, $CD(a_i, a_j)_{\mathsf{Path}_{i,j}}$ denotes the probability of **conjunctive** events where each event refers to an a_k's occurrence in $\mathsf{Path}_{i,j}$ as per Equation 6:

$$CD(a_i, a_j)_{\mathsf{Path}_{i,j}} = \prod_{a_l, a_m \in \mathsf{Path}_{i,j}} CD(a_l, a_m \ \mathsf{Operator}\ \{a_{k_m}\})_{\mathsf{SEQ}} \tag{6}$$

[10] Let n be 3, $\sum_{r=1,n}(\frac{r}{n} \times C(n, r))$ has the following value: $(\frac{1}{3} \times 3 + \frac{2}{3} \times 3 + \frac{3}{3} \times 1{=}4)$.

[11] Let n be 3, $\sum_{r=1,n} C(n, r)$ has the following value: $(3{+}\ 3{+}\ 1){=}7$.

We now examine the *control dependency* $CD(a_i, a_j)_{\mathsf{Paths}_{i,j}}$ where multiple **possible** execution paths ($\{\mathsf{Path}^q_{i,j}\}$) exist between a_i and a_j. Here, $CD(a_i, a_j)_{\mathsf{Paths}_{i,j}}$ denotes an aggregation of all simple *control dependencies*, each associated with as per Equation 7:

$$CD(a_i, a_j)_{\mathsf{Paths}_{i,j}} = \mathsf{Agg}(CD(a_i, a_j)_{\{\mathsf{Path}^q_{i,j}\}_{q=1,\ldots}}) \tag{7}$$

where Agg refers to some common aggregate function like maximum used for our experiments.

Table 3 depicts an excerpt of *control dependencies* in Bicing.

Table 3: Control dependencies with the occurrence probability (p) set to 0.5

Activity	a_1	a_2	a_3	a_4	a_5
a_1	-	1/2	5/6	11/12	17/12
a_2	1/2	-	1/3	7/12	11 /12
a_3	5/6	1/3	-	1/4	3/4
a_4	11/12	7/12	1/4	-	1/2
a_5	7/12	11/12	3/4	1/2	-

3.4 Data dependency analysis

In Section 3.2, we considered the *criticality* of an artefact and attribute as a means for measuring *data dependency* between two activities. In [30], Paulsen et al. provide a comprehensive criticality analysis for the benefit of organizations that need to identify and prioritize assets (e.g., artefacts and processes) that are vital for achieving their goals. Taping into this analysis, we distinguish two types of criticality: *functional* (F) that refers to an artefact's/attribute's unavailability which could hinder the BP's proper execution, and *non-functional* (NF) that refers to an artefact's/attribute's corruption which could undermine the BP's Quality-of-Service (QoS). To assist BP engineers classify artefacts/attributes as (to some degree) either critical or non-critical, we identify F- and NF-based critical attribute's/artefact's properties. First, we consider *strategic*, *tactical*, and *operational* information levels as F criticality-related properties. For instance, a *strategic* attribute should be more critical than *tactical* and *operational* attributes. Second, we consider *privacy*, *confidentiality*, *integrity*, and *availability* security levels as NF criticality-related properties. For instance, some BP's operations can be less concerned about *confidentiality* but more concerned about first *availability* and then, *integrity*. Table 4 illustrates these properties with examples for Bicing. For instance, User_Credit attribute considered as both decision-making and financial data would have *operational* and *confidentiality* as F and NF criticality-related properties, respectively.

Table 4: Examples illustrating F and NF criticality-related properties

Criticality	Properties	Example
Functional	Operational	Decision-making data
	Tactical	Concurrency data
	Strategic	Customer experience data
Non-functional	Privacy	Protected personal data
	Integrity	Identity data
	Confidentiality	Financial data

To define an artefact (ar)/attribute (at)'s Degree of Criticality ($DC(ar|at)$), the BP engineer needs to work out the priority among F and NF criticality-related properties by using three clusters, Low (L), Medium (M), and High (H). Then,

she maps L, M, and H clusters onto [0, k[, [k, k'[, and [k', k''], respectively, after setting the values of k, k', and k". Finally, the BP engineer sets $DC(ar|at)$ based on the cluster's range to which an ar|at's property belongs. Since artefacts and attributes can be associated with F and/or NF, for instance User_Credit, we refine $DC(ar|at)$ into $DC^{\mathsf{F}}(ar|at)$ and $DC^{\mathsf{NF}}(ar|at)$.

To calculate $DC(ar|at)$, we define two strategies. The first strategy computes $DC(ar|at)$ as a weighted sum of $DC^{\mathsf{F}}(ar|at)$ and $DC^{\mathsf{NF}}(ar|at)$ (Equation 8).

$$DC(ar|at) = w_1 \times DC^{\mathsf{F}}(ar|at) + w_2 \times DC^{\mathsf{NF}}(ar|at), \quad w_1 + w_2 = 1 \tag{8}$$

where w_1 and w_2 are weights (i.e., importance) associated with $DC^F(ar|at) \mid DC^{NF}(ar|at)$, respectively, and set by the BP engineer. Table 5 depicts artefacts/attributes associated with criticality degrees for Bicing. For instance, Bike_Status has a higher DC than Anchor_Point.

Table 5: Artefact/Attribute criticality for Bicing

Artefact	Attributes	DC^F
Bicycle	Anchor_Point	M (k'$_1$)
	Bike_Status	H (k"$_1$)
User	User_Status	H (k"$_2$)
	User_Destination	L (k$_2$)
	User_History	H (k"$_2$)
Rental	Rent_ID	H (k"$_3$)
	Rent_Cost	M (k'$_3$)
Repair	agree_Repair	M (k'$_4$)

Artefact	Attributes	DC^{NF}
Bicycle	Bike_ID	H (k"$_1$)
User	User_ID	H (k"$_2$)
	User_Validity	M (k'$_2$)
Repair	estimated_Repair_Cost	H (k"$_4$)

Once all $DC(ar|at)$ are established, we now specify *data dependencies* ($\mathcal{DD}^1$) between a_i and a_j (Equation 9).

$$\mathcal{DD}^1(a_i, a_j) = \sum_{ar_{i,j}|at_{i,j} \in DATA_{i,j}} pair(ar_{i,j}|at_{i,j}) \times DC(ar_{i,j}|at_{i,j}) \tag{9}$$

where

- $ar_{i,j}|at_{i,j}$ represents the artefact/attribute exchanged between a_i and a_j,
- $DATA_{i,j}$ indicates the set of $ar_{i,j}|at_{i,j}$, and
- $pair(ar_{i,j}|at_{i,j})$ denotes the value associated with the operation pair (e.g., r/w, w/w, and c/r) between a_i and a_j, proposed by Amiri [2].

Table 6 depicts an excerpt of *data dependencies* for Bicing using Equation 8.

Table 6: Excerpt of data dependencies

Activity	a_1	a_2	a_3	a_4	a_5
a_1	-	1/2	5/6	11/12	17/12
a_2	1/2	-	1/3	7/12	11 /12
a_3	5/6	1/3	-	1/4	3/4
a_4	11/12	7/12	1/4	-	1/2
a_5	7/12	11/12	3/4	1/2	-

The second strategy considers $DC(ar|at)$ as a tuple $< DC^{\mathsf{F}}(ar|at), DC^{\mathsf{NF}}(ar|at) >$. We, thus, specify *data dependencies* ($\mathcal{DD}^2$) between a_i and a_j (Equation 10):

$$\mathcal{DD}^2(a_i, a_j) = \sum_{ar_{i,j}|at_{i,j} \in DATA_{i,j}} \mathcal{F}(pair(ar_{i,j}|at_{i,j}), DC^{\mathsf{F}}(ar_{i,j}|at_{i,j}), DC^{\mathsf{NF}}(ar_{i,j}|at_{i,j})) \quad (10)$$

where $\mathcal{F}$ returns the *data dependency* value specified by the BP engineer for $< pair(ar_{i,j}|at_{i,j}), DC^{\mathsf{F}}(ar_{i,j}|at_{i,j}), DC^{\mathsf{NF}}(ar_{i,j}|at_{i,j}) >$.

3.5 Semantic dependency analysis

Semantic dependencies between activities define whether or not there are similarities between what activities are supposed to perform like ensuring the payment of fines and and blacklisting users. Such similarities could be extracted using activities' names defined with respect to a domain ontology. We advocate that activities' names presenting semantic similarities and/or referring to the same ontology concept(s) could be inter-related and the, would likely be part of the same microservices.

Earlier, we mentioned three annotation techniques that we use to measure *semantic dependencies* between two activities. Let $\mathcal{X}$ be an annotation technique and $SD^{\mathcal{X}}(a_i, a_j)$ be a *semantic dependency* between a_i and a_j using $\mathcal{X}$. Prior to formalizing $SD^{\mathcal{X}}$, we describe these techniques.

Word-driven technique ($\mathcal{W}$). Albeit being simple, this technique measures to what extent 2 words are similar. In our case, word would refer to an activity's name like "check user credentials" in our Bicing case-study. One of the advantages of *Word-driven* technique is that it does not refer to a domain ontology but relies on the word co-occurrence[12] principle. Co-occurrence refers to the above-chance frequency of occurrence of two terms and is used as an indicator to measure the semantic proximity of two terms. Algorithm 1 outlines how an activity a_i is annotated using a set of the n most similar words ($\mathcal{W}_{a_i}$). To develop $\mathcal{W}_{a_i}$, we adopt No et al.'s solution, $DISCO$ [22], that assumes that words with similar meaning (co-)occur in similar bag-of-words contexts. By adopting $DISCO$, the set of a_i's n most distributionally similar words ($\{w_k\}$) along with their respective similarity degrees ($sd_{i,k}$) (i.e., $\mathcal{W}_{a_i}$) are returned. Besides $DISCO$, any other semantic textual similarity technique can be adopted in our approach. Thus, a_i will be annotated with $\{< w_k, sd_{i,k} >\}$.

Algorithm 1: *Word-driven* annotation technique

```
Input: a_i, n
Output: W_{a_i}
1 begin
2     W_{a_i} ← DISCO(a_i, n)
3     return W_{a_i}
```

Concept-driven technique ($\mathcal{C}$). It uses a domain ontology to annotate activities' names with closed concepts. The semantic dependency between activities is then highly dependent on the similarities that could exist between the annotated concepts. Algorithm 2 outlines how an activity a_i is annotated using a set of the most similar **concepts** ($\mathcal{C}_{a_i}$) that would belong to the BP's domain ontology ($\mathcal{O}_{BP}$). Fig. 3 depicts our in-house $\mathcal{O}_{BP}$ for bike rental. For each concept c_j, Algorithm 5 first calls for No et al.'s semantic similarity measure between a_i and c_j namely, $DISCO_2$[13], and then stores $DISCO_2$'s outcome into $D[i, j]$. To develop $\mathcal{C}_{a_i}$, the max_s function will keep the concepts (c_k) with the highest similarity values stored in $D[i]$ with respect to a certain precision σ. For instance, if the highest similarity value is 0.5 and σ is set to 0.1, then max_s will include all the concepts with a similarity value between 0.4 and 0.5, as well. Thus, a_i will be annotated with $\{< c_k, D[i, k] >\}$. Table 7 depicts similarity values associated with Bicing.

[12] en.wikipedia.org/wiki/Co-occurrence.

[13] $DISCO_2$ computes semantic similarity (i.e., relation between concepts) while $DISCO$ computes distributional similarity (i.e., relation between words).

Algorithm 2: *Concept-driven* annotation technique

Input: a_i, $\mathcal{O}_{BP}$, σ
Output: $\mathcal{C}_{a_i}$
begin
  foreach $c_j \in \mathcal{O}_{BP}$ **do**
    $\mathcal{D}[i,j] \leftarrow DISCO_2(a_i, c_j)$
  $\mathcal{C}_{a_i} \leftarrow max_s(\{\mathcal{D}[i,j]\}, \sigma)$
  return $\mathcal{C}_{a_i}$

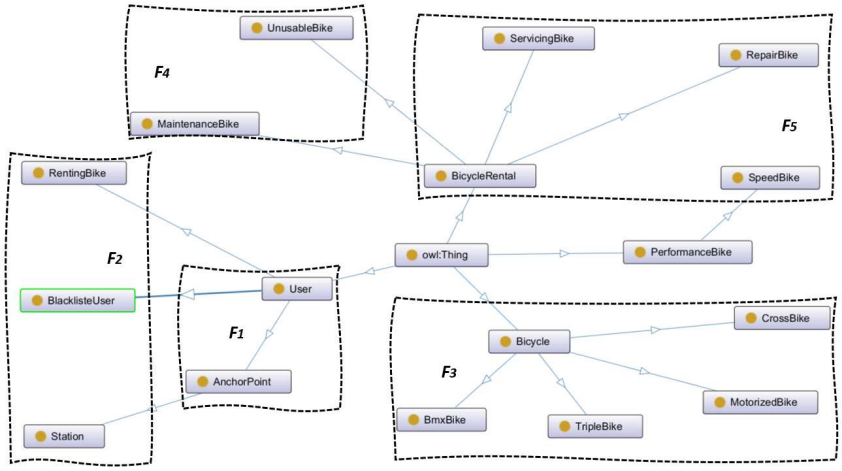


Fig. 3: Domain ontology for bike rental and its fragments F_i

Table 7: Excerpt of similarity values between activities and concepts

Activity \ Concept	*Bicycle*	*BicycleRental*	*User*	*AchorPoint*	*Station*
a_1	0.01818	0.02632	0.18076	0.04190	0.03621
a_2	0.01924	0.04481	0.99997	0.05231	0.03667
a_3	0.00421	0.01784	0.04266	0.12018	0.14423
a_5	0.00718	0.00476	0.00962	0.05937	0.02105
a_9	0.00968	0.01392	0.15772	0.0614	0.03038
a_{10}	0.12016	0.06755	0.04724	0.04081	0.10965

Fragment-driven technique ($\mathcal{F}$). Instead of using the most closed concept, this technique annotates an activity's name using the most closed set of concepts called fragments. In this case, semantic dependencies between two activities is highly dependent on the similarities that could exist between two fragments. Algorithm 3 outlines how an activity a_i is annotated using a set of the most similar (eventually overlapped) **fragments** ($\mathcal{F}_{a_i}$) that each encompasses concepts belonging to $\mathcal{O}_{BP}$. To develop $\mathcal{F}_{a_i}$, Algorithm 3 first computes a_i's membership degrees ($\mathcal{M}[i]$) to each fragment $\in \mathcal{O}_{BP}$ based on the set of common concepts (c_j) between this fragment and $\mathcal{C}_{a_i}$ along with $D[i,j]$ obtained in Algorithm 2, Line 2. Then, the max_m function considers the fragments (F_k) with the highest membership degrees ($\mathcal{M}[i]$) with respect to a certain precision σ. Thus, a_i will be annotated with $\{< F_k, \mathcal{M}[i,k] >\}$. Table 8 depicts an excerpt of similarity values associated with Bicing.

Algorithm 3: *Fragment-driven* annotation technique

Input: $\mathcal{C}_{a_i}$, $\mathcal{O}_{BP}$, σ
Output: $\mathcal{F}_{a_i}$

```
1 begin
2     foreach F_k ∈ O_BP do
3         M[i,k] ← Σ_{c_j ∈ F_k ∩ C_{a_i}} D[i,j]
4     F_{a_i} ← max_m(M[i], σ) return F_{a_i}
```

Table 8: Excerpt of similarity values between activities and fragments

Activity \ Fragment	F_1	F_2	F_3	F_4	F_5
a_1	0.98182	0.97368	0.9581	0.98182	0.97368
a_2	0.98076	0.95519	0.94769	0.98076	0.96333
a_3	0.99579	0.98216	0.95734	0.99579	0.98216
a_5	0.99524	0.99524	0.99038	0.99282	0.99524
a_9	0.99032	0.98608	0.9386	0.99032	0.98608
a_{10}	0.93245	0.95276	0.95919	0.95919	0.93245

Formally, Equation 11 computes the Semantic Dependency (SeD) between a_i and a_j.

$$SeD(a_i, a_j) = 1 - d_{\mathcal{X}}(\mathcal{X}_{a_i}, \mathcal{X}_{a_j}) \tag{11}$$

where $\mathcal{X}_{a_i}$ corresponds to the annotation technique's outcome (either $\mathcal{W}_{a_i}$, $\mathcal{C}_{a_i}$, or $\mathcal{F}_{a_i}$) and $d_{\mathcal{X}}$ represents the distance between $\mathcal{X}_{a_i}$ and $\mathcal{X}_{a_j}$. We hereafter define $d_{\mathcal{X}}$ per technique.

- *Word-driven* distance. The rationale of $d_{\mathcal{W}}$ is to compare all the words in $\mathcal{W}_{a_i}$ to those in $\mathcal{W}_{a_j}$. More $\mathcal{W}_{a_i}$ and $\mathcal{W}_{a_j}$ contain similar *private* words (i.e., that exclusively belong to either $\mathcal{W}_{a_i}$ or $\mathcal{W}_{a_j}$) with higher similarity with a_i and a_j, respectively, more a_i and a_j will be far away from each other. Formally, Equation 12 computes $d_{\mathcal{W}}$.

$$d_{\mathcal{W}}(\mathcal{W}_{a_i}, \mathcal{W}_{a_j}) = \sum_{w_k \in \mathcal{W}^p_{a_i}} sd_{i,k} + \sum_{w_k \in \mathcal{W}^p_{a_j}} sd_{j,k} \tag{12}$$

where $\mathcal{W}^p_{a_i|a_j}$ represents $\mathcal{W}_{a_i|a_j}$'s set of *private* words (i.e., $\mathcal{W}_{a_i|a_j} - \mathcal{W}_{a_i} \cap \mathcal{W}_{a_j}$).

Equation 13 computes the normalized $d_{\mathcal{W}}$.

$$d^{norm}_{\mathcal{W}}(\mathcal{W}_{a_i}, \mathcal{W}_{a_j}) = \frac{d_{\mathcal{W}}(\mathcal{W}_{a_i}, \mathcal{W}_{a_j})}{|\mathcal{W}^p_{a_i}| + |\mathcal{W}^p_{a_j}|} \tag{13}$$

Table 9 depicts an excerpt of semantic dependencies for Bicing using Equation 13.

Table 9: Excerpt of semantic dependencies using *word-driven* technique

Activity / Activity	a_1	a_2	a_3	a_4	a_5	a_6
a_1	1	0.180	0.034	0.015	0.010	0.041
a_2	0.180	1	0.042	0.012	0.009	0.020
a_3	0.034	0.042	1	0.0019	0.266	0.120
a_4	0.015	0.012	0.0019	1	0.0043	0.0021
a_5	0.010	0.009	0.266	0.0043	1	0.0593
a_6	0.041	0.020	0.120	0.0021	0.0593	1

- *Concept-driven* distance. The rationale of $d_{\mathcal{C}}$ is to compare all the concepts in $\mathcal{C}_{a_i}$ with those in $\mathcal{C}_{a_j}$. More $\mathcal{C}_{a_i}$ and $\mathcal{C}_{a_j}$ contain closest concepts with higher similarity with a_i and a_j, respectively, more a_i and a_j will be close from each other. Formally, Equation 14 computes $d_{\mathcal{C}}$.

$$d_{\mathcal{C}}(\mathcal{C}_{a_i}, \mathcal{C}_{a_j}) = \sum_{c_k \in \mathcal{C}_{a_i}} (1 - \mathcal{D}[i,k]) * \sum_{c_l \in \mathcal{C}_{a_j}} (1 - \mathcal{D}[j,l]) * WU(c_k, c_l) \tag{14}$$

where $WU(c_k, c_l)$ represents the distance between the two concepts c_k and c_l[36].

Equation 15 computes the normalized $d_{\mathcal{C}}$.

$$d^{norm}_{\mathcal{C}}(\mathcal{C}_{a_i}, \mathcal{C}_{a_j}) = \frac{d_{\mathcal{C}}(\mathcal{C}_{a_i}, \mathcal{C}_{a_j})}{|\mathcal{C}_{a_i}| * |\mathcal{C}_{a_j}|} \tag{15}$$

Table 10 depicts an excerpt of semantic dependencies for Bicing using Equation 15.

Table 10: Excerpt of semantic dependencies between activities using *concept-driven* technique

Activity / Activity	a_1	a_2	a_3	a_4	a_5
a_1	-	0,488573	0,488843	0,390858	0,320864
a_2	0,488573	-	0,488315	0,320517	0,320517
a_3	0,488843	0,488315	-	0,495525	0,325429
a_4	0,390858	0,320517	0,495525	-	0,275969
a_5	0,320864	0,320517	0,325429	0,275969	-

- *Fragment-driven* distance. The rationale of $d_{\mathcal{F}}$ is to compare all the fragments in $\mathcal{F}_{a_i}$ with those in $\mathcal{F}_{a_j}$. More $\mathcal{F}_{a_i}$ and $\mathcal{F}_{a_j}$ contain *distinctive* concepts with higher membership degrees for a_i and a_j, respectively, more a_i and a_j will be far away from each other. Formally, Equation 16 computes $d_{\mathcal{C}}$.

$$d_{\mathcal{F}}(\mathcal{F}_{a_i}, \mathcal{F}_{a_j}) = \prod_{\mathcal{F}_k \in \mathcal{O}_{BP}} (1 - |\mathcal{M}[i,k] - \mathcal{M}[j,k]|) \tag{16}$$

Let $\mathcal{P}_2(BP)$ be the set of all distinct activity pairwises built from BP (i.e., $a_k \neq a_l$). Formally, Equation 17 computes the normalized $d_{\mathcal{F}}$ as follows.

$$d^{norm}_{\mathcal{F}}(\mathcal{F}_{a_i}, \mathcal{F}_{a_j}) = \frac{d_{\mathcal{F}}(\mathcal{F}_{a_i}, \mathcal{F}_{a_j})}{d^{max}_{\mathcal{F}}} \tag{17}$$

where

- $d^{max}_{\mathcal{F}} = max(\{d_{\mathcal{F}}(\mathcal{F}_{a_k}, \mathcal{F}_{a_l})\}_{<a_k, a_l> \in \mathcal{P}_2(BP)})$

Table 11 depicts an excerpt of semantic dependencies for Bicing using Equation 17.

Table 11: Excerpt of semantic dependencies using *fragment-driven* technique

Activity / Activity	a_1	a_2	a_3	a_5	a_9	a_{10}
a_1	-	0,97696	0,95914	0,90827	0,94119	0,90829
a_2	0,97696	-	0,92116	0,87033	0,92959	0,90731
a_3	0,95914	0,92116	-	0,94248	0,96703	0,83428
a_5	0,90827	0,87033	0,94248	-	0,92845	0,82091
a_9	0,94119	0,92959	0,96703	0,92845	-	0,82586
a_{10}	0,90829	0,90731	0,83428	0,82091	0,82586	-

4 Collaborative clustering development and experimentation

In this section we discuss the technical details of our microservices identification approach with focus on the collaborative clustering technique and the experiments that were carried out.

4.1 Collaborative clustering

Clustering is one of the well known Machine Learning (ML) techniques. Given a set of objects, it consists of classifying each object into a specific group called cluster. In our work, we consider each BP's activity (a_i) a distinct object. Activities of the same cluster are expected to be as homogeneous as possible to ensure the cohesion property of a group. Contrarily, activities belonging to different groups are expected to be as distinct as possible to ensure the loose coupling of a group. Each group of activities could be a potential microservice.

Our *c*ollaborative clustering algorithm (*c*HAC) extends the classical Hierarchical agglomerative algorithm (HAC) [29]. *c*HAC is performed by N homogeneous clustering nodes $(CN_1, CN_2, \ldots, CN_n)$ executing the same program. However these nodes differ in terms of inputs. Each CN node handles one and only one dependency matrix. The chosen number k of clusters at each CN is not necessary the same; it can be different from one CN to another one.

Our *c*HAC algorithm fosters collaboration between CN since each CN has its own dependency matrix along with "keeping an eye" on what other CNs are doing by sharing some dependencies scores about activities, if deemed necessary. Thus, prior to each new HAC clustering iteration, a CN uses both a Local Score Matrix (LSM) that stores dependency scores between couple of activities and a Shared Score Matrix (SSM) that stores a global dependency score between each couple of activities. Our *c*HAC algorithm goes over three phases:

- Initialization phase (Algorithm 4). All CN nodes are launched with their respective number of clusters k, and their respective local dependency score matrices. An empty shared dependency matrix is also created to store the shared dependency score of activities. Each activity constitutes a cluster. A shared variable is also introduced to synchronize the iteration of the nodes. A new iteration is launched at a given node if-and-only-if the other nodes have already finished their current iterations.

- Collaborative Iteration phase. A classical HAC clustering is extended to make it collaborative as per Algorithm 5. The nearest pair of clusters C_u and C_v is computed by using both LSM and SSM based on calculating the distance using the formula:

$$SM_p\left(C_u, C_v\right) = \sum_{(i,j)=(1,1)}^{|C_u|*|C_v|} SM_{p-1}\left(a_i, a_j\right) / \left(|C_u| * |C_v|\right)$$

where the score matrix SM_p designates either LSM or SSM matrix at the p^{th} iteration.
Clusters C_u and C_v are merged if and only if:[14]

$$[distance(C_u, C_v)]_p^{LSM} >= [distance(C_u, C_v)]_{p-1}^{SSM}$$

[14] $[distance(C_u, C_v)]_p^{SC}$ computes any distance between two clusters C_u and C_v by using the score Matrix SM of the iteration p.

Algorithm 4: *c*HAC - collaborative clustering initialization

Input: N - number of clustering node CN, K_i $i = 1..N$ - targeted number of clusters of each node CN_i, $DM_i[M, M]$ $i = 1..N$ - DM_i Dependency Matrix of CN_i, $SDSM[M, M]$ - Shared dependency score matrix;
Output: final cluster result fc;

```
1 begin
2     SIC ← 0 ;                                                    // shared counter between CN nodes
3     cSET[i] ← cHACn(k_i, DM_i, @SDSM, @SIC), i = 1..M ;          // parralel execution of CN nodes
4     ; fc ← chooseFinalClusterResult(cSET);
5     return fc;
```

Algorithm 5: *c*HACn - collaborative clustering of each node CN_i

Input: K - targeted number of clusters, N: number of clustering node CN, DM: Dependency matrix, $@SDSM$: Shared dependency score matrix;
Output: $\mathbb{C}$ - clustering set

```
1  begin
2      C_u, C_v, C_p - cluster variables
3      ℂ ← {{a_1}, {a_2}, ..., {a_M}} ;                             // each activity is a cluster
4      while |ℂ| > k do
5          (C_u, C_v) ← NearestPeerCluster(ℂ);
6          if [distance(C_u; C_v)]^DM ⩾ [distance(C_u; C_v)]^SDSM then
7              C_p ← fusion(C_u, C_v)  ℂ ← ℂ − (C_u ∪ C_v)  ℂ ← ℂ ∪ C_p;
8              updateDM();
9              [SDSM(t_i, t_j)]_p = Max([LDSM(t_i, t_j)]_p[SDSM(t_i, t_j)]_{p−1});
10             @SIC ← @SIC + 1  mod N ;   // The node indicate to other nodes that its current clustering iteration is done
11
12         WAIT (@SIC = 0) ;  // the new iteration will be started only when all oher CN nodes have finished their current clustering iteration
13         updateSDSM();
14     return ℂ ;
```

Then, the node updates the LSM by calculating the new scores of activities using the formula:

$$SM_p\,(a_i, a_j) = \sum_{j=1}^{|C_v|} SM_p\_1\,(a_i, a_j)\,/|C_v|$$

To foster similarities between couples of activities (a_i, a_j), the shared score matrix is also updated as follows: $[SSM(a_i, a_j)]_p = Max([LDSM(a_i, a_j)]_p, [SSM(a_i, a_j)]_{p-1})$

- Selection phase. Once the different clustering results are produced by the different CNs, the distance metrics are applied to them (Algorithm 4, line 5) to choose the best one that fosters both cohesion and loose-coupling of groups.

It is important to note that our *c*HAC algorithm can work either in a uniform collaboration strategy where each CN collaborates with other CNss, or in diverse collaboration strategy where each CN node has its own collaborators. For the latter case, different shared matrices are needed, one by CN node.

The *c*HAC algorithm we propose in this work is different from the distributed HAC (dHAC) [20]. Indeed, dHAC is described in terms of two phases. In the first phase, the entire collection of objects is divided into n disjoint segments and distributed over n HAC processes. Each HAC process is dedicated to one and only one segment to generate one separate clustering result. In the second phase, the individuals generated clusters are merged into one final cluster result. Instead of merging individuals clustering results, our *c*HAC algorithm allows nodes to continuously collaborate to generate their clustering results. The collaboration between the different HAC nodes is carried out between two successive iterations by sharing intermediate clustering results.

4.2 Potential microservices identification options from a set of business processes

One of the salient features of our approach and its collaborative clustering algorithm is, that, it can be applied to many BPs when identifying potential microservices. Let us assume a set of n BPs $\{BP_1, BP_2, \cdots, BP_n\}$ and a set of m dependency models $\{M_1, M_2, \cdots, M_m\}$ related to these BPs. Our approach generates n dependency matrices from each model, which means $n * m$ dependency matrices. Afterwards, these matrices would be used according to one of the following options:

- No merge option. It keeps each matrix independent from the rest and then, applies the collaborative clustering algorithm to $m * n$ nodes where each node is related to $m * n$ independent matrices.

- Merge by model option. It merges all matrices that originate from the same model applied to the different BPs and then, applies the collaborative clustering algorithm to m nodes where each node is related to a merged matrix.

- Merge by BP option. It merges all matrices that originate from the different dependency models applied to one BP and then, applies the collaborative clustering algorithm to m nodes where each node is related to a merged matrix.

Figure 4 illustrates these three main options for the case of n BPs and our three activity dependency models. Even if no merge option is more appropriate for the collaborative clustering algorithm as it makes the collaboration more larger, the user can, for any reason, choose the other options or define new ones such as merging matrices in a random way.

Once microservices are identified, their implementation could be either proprietary or based on external resources. The choice could be based on some requirements including security and privacy.

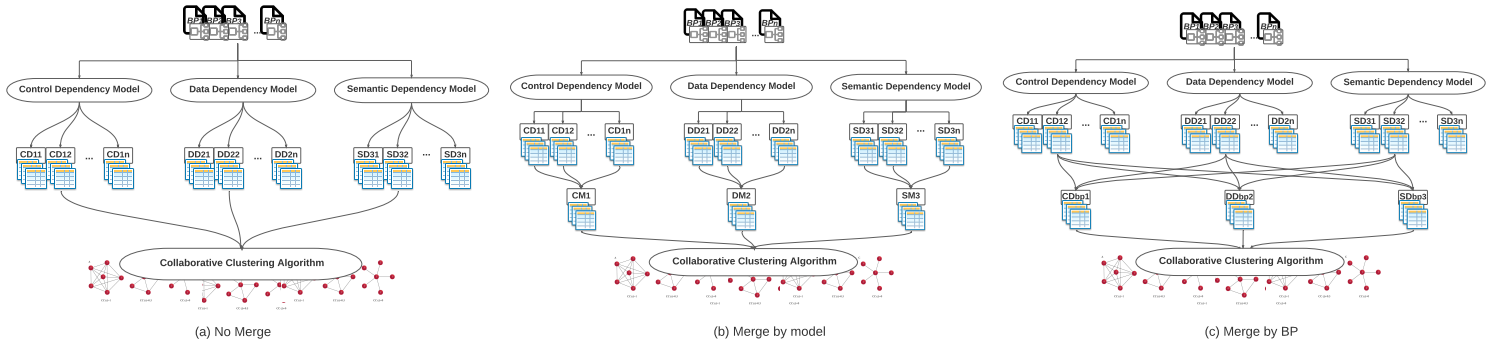


Fig. 4: Options for combining dependency matrices

4.3 System implementation

A system demonstrating the technical doability of our approach for identifying microservices has been implemented. Fig. 5 shows the system's architecture. Once a BP is modeled, its process model is converted into XML using Camunda plugin. The system also includes two core modules, *dependency analysis* and *microservice extraction*. The first parses the XML-based BP to extract all execution paths between any couple of activities $< a_i, a_j >$ and their respective artefact/attribute exchange, as well, and takes the BP's OWL domain ontology developed under Protégé 5.5[15], as input. Note that artefacts/attributes will be associated with criticality values as per Section 3.4. To compute dependency values, the *dependency-analysis* module implements all the equations and algorithms reported in Sections 3.3, 3.4, and 3.5. All these values are stored as dependency graphs in XML. The second module, *microservice extraction*, implements our *c*HAC algorithm (Section 4.1). It first imports all the XML files produced by the first module using Java DOM Parser[16], a Java API for Document Object Models. Then, it runs the *c*HAC with respect to the selected technique whether *word-driven*, *concept-driven*, or *fragment-driven* for the *semantic* perspective to build different clusters (i.e., microservices). Finally, this module maps the obtained clusters onto a given XSD format.

[15] `protege.stanford.edu`.

[16] `xerces.apache.org/xerces2-j/javadocs/xerces2/org/apache/xerces/parsers/DOMParser.html`.

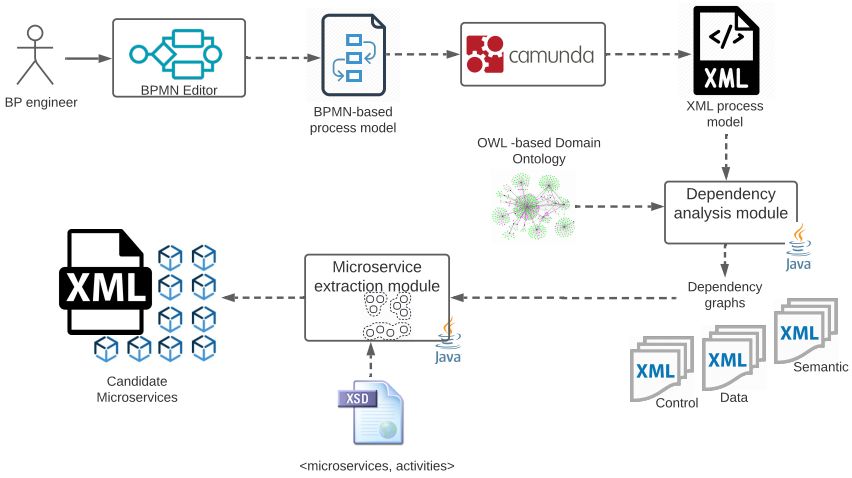
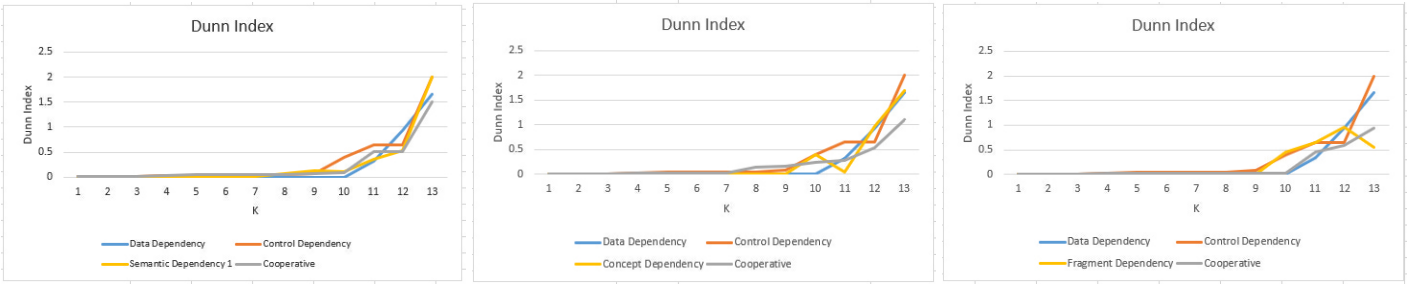


Fig. 5: Architecture of the implemented system

4.4 Experiments

First experiment. We evaluated the *c*HAC algorithm using both the quality metric, Dunn index, and the convergence time, Time index. Note that Dunn index permits to identify the clusters that are compact (i.e., minor variance between activities belonging to the same cluster) and separate (i.e., large distance between clusters). Thus, a higher Dunn index indicates better clustering. We applied the *c*HAC algorithm to dependencies produced by the *dependency-analysis* module from the BP's Bicing. Initially, fourteen activities related to Bicing were used and then, more random activities were added to the case study to capture the complexity of real BPs. In this first experiment, we proceed with three nodes as per Section 4.1 namely, *control*, *data*, and *semantic*, another node, named *cooperative*, that aggregates all dependencies. We noted that the *semantic* node can be refined into either *word-driven*, *concept-driven*, or *fragment-driven*. In Fig. 6, we clearly observe that the Dunn index in the case of the *control* node is ten almost always better than the Dunn index the other two nodes (i.e., *data* and *semantic* (regardless of the annotation technique used)). This means that for a given BP, the control node gives richer and more informative analysis than the rest. This confirms that aggregating different dependencies (i.e., cooperative node) degrades the quality of the final clustering and the collaboration between nodes is the most appropriate option for achieving better clustering results. Regarding the time index (Fig. 7), the *control* node still gives the best results.

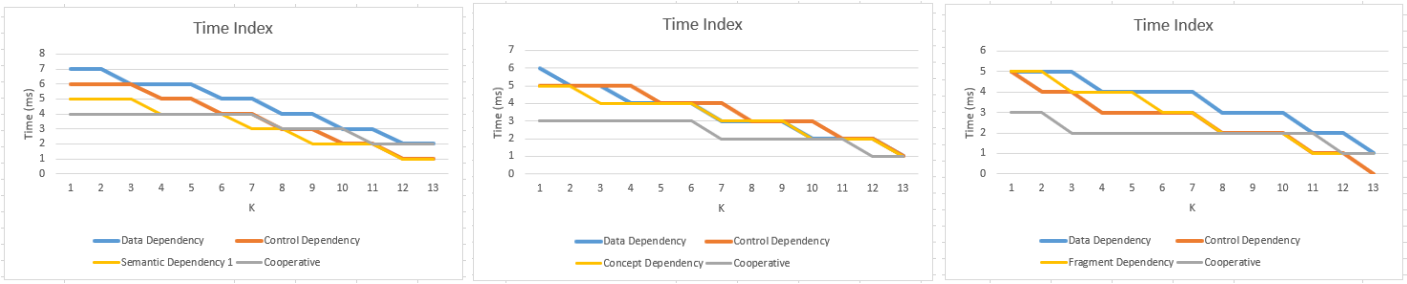


(a) Control, data, word-driven, and cooperative

(b) Control, data, concept-driven, and cooperative

(c) Control, data, fragment-driven, and cooperative

Fig. 6: Dunn Index - Bicing case



(a) Control, data, word-driven, and cooperative (b) Control, data, concept-driven, and cooperative (c) Control, data, fragment-driven, and cooperative

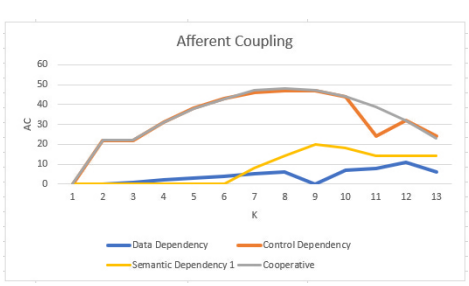
Fig. 7: Time Index - Bicing case

Note that Dunn index sheds light on the quality of clustering regardless of whether the obtained clusters (i.e., microservices) are meaningful to the BP designer. To this end, we proceed with additional metrics for further analysis. As per Section 3.2, coupling and cohesion are important aspects to consider when assessing the quality of microservices. To this end, we rely on the work of [15] that quantifies both aspects using four metrics, Afferent Coupling (AC), Efferent Coupling (EC), Instability (I), and Relational Cohesion (RC) (Table 12), with focus on AC and RC. Since these metrics refer to classes and class packages, we mapped them onto concepts relevant and applicable to our work. Therefore, class/package becomes activity/cluster, service becomes refers to microservice, and internal relations among classes in the same package becomes either connector type, data transfer, or semantic similarity between activities in the same cluster. Fig. 8 depicts AC's average values over clusters obtained at iteration K for the *control*, *data*, and *word|concept|fragment-driven* nodes. We observe that the control node still outperforms the cooperative one, while it provides stronger coupling than data and semantic nodes. Regarding RC (Fig. 9), the *control* node outperforms significantly the three other nodes.

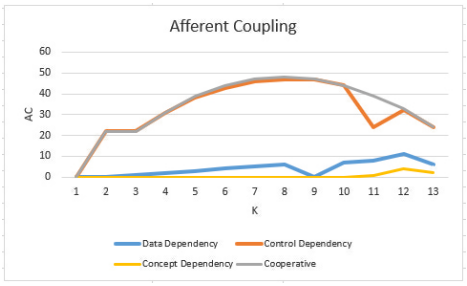
Table 12: Performance metrics extracted from [15]

Aspect	Metrics	Definition
Coupling ([27])	Afferent Coupling (AC)	Measures 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 (EC)	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)	Measures a package's (service's) resilience to change and is calculated as $\frac{EC}{EC+AC}$. $I = 0$ indicates a completely stable package (service) whereas $I = 1$ a completely unstable package (service)
Cohesion ([24])	Relational Cohesion (RC)	Measures 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).

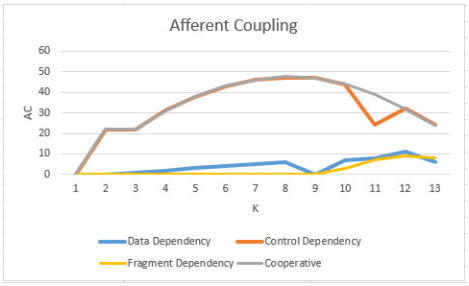
Specifically, we challenge the *word-driven* and *concept-driven* annotation technique for microservices identification (Table 13). Both techniques yield into five microservices. However, the *word-driven* technique outperformed the *concept-driven* one in many aspects. First, the obtained microservices in the *word-driven* are more strongly-cohesive and more or less loosely-coupled compared to the second technique (i.e., higher AC and a bit lower RC). Moreover, the microservices in the *word-driven* are more meaningful (or consistent) than those obtained in the *concept-driven*. Note that due to this inconsistency, we do not name the obtained microservices. For instance, $ms\#2$ encompasses loosely-dependent activities related to bike return and bike geo-localization, respectively. This can be explained by the lack of completeness of the ontology used.



(a) Control, data, word-driven, and cooperative

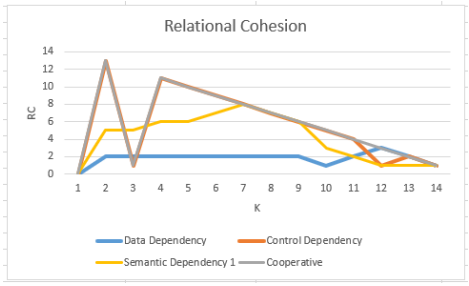


(b) Control, data, concept-driven, and cooperative

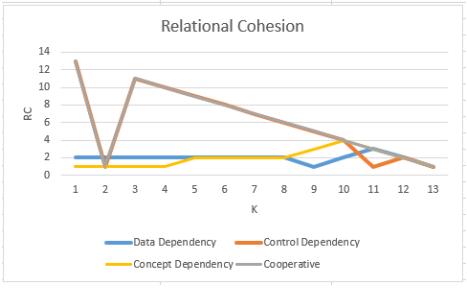


(c) Control, data, fragment-driven, and cooperative

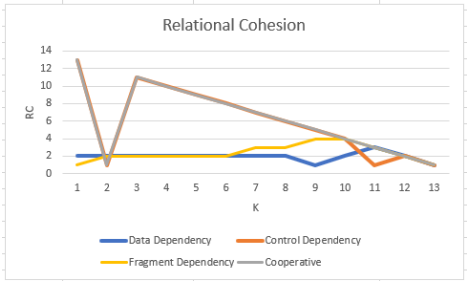
Fig. 8: Afferent coupling over K - Bicing case



(a) Control, data, word-driven, and cooperative



(b) Control, data, concept-driven, and cooperative



(c) Control, data, fragment-driven, and cooperative

Fig. 9: Relational cohesion over K - Bicing case

Table 13: Metrics of candidate microservices using our approach - Bicing case

	Microservice	Activities \ Metric	*AC*	*EC*	*I*	*RC*	
word-driven	*RequestHandling*	a_1, a_2	13	0	0	2	
	Revision&Validation	a_3, a_4, a_5	18	1	0.05	5	
	BikeAbandon	a_{12}, a_{13}, a_{14}	11	5	0.31	6	
	BikeReturn	a_{10}, a_{11}	12	5	0.29	3	
	BikeReparation	a_6, a_7, a_8, a_9	23	3	0.12	5	
			15.4	2.8	0.154	4	*Avg*
concept-driven	*ms*#1	a_1, a_2, a_3, a_4, a_5	25	0	0	14	
	ms#2	a_{13}, a_{14}	14	1	0.07	2	
	ms#3	a_{11}, a_{12}	14	1	0.07	2	
	ms#4	a_9, a_{10}	14	1	0.07	2	
	ms#5	a_8, a_7, a_6	21	1	0.05	3	
			17.6	0.8	0.052	5	*Avg*

Second experiment. The objective here is to compare our approach to those presented in [4], [16], and [26] using the case study of cargo tracking of Gysel et al. [16]. For the needs of the second experiment, we designed a BPMN-based process model for cargo tracking (Fig. 10). In this process model, it starts when the shipping company ships a customer's containers (a_1') by land and sea. After checking the customer's credentials (a_2') and doability of the shipping options (a_3'), the company either sends an invoice to the customer for payment (a_5') or notifies her of the rejection (a_4'). Upon payment confirmation (a_6'), additional activities are performed. First, a_7' arranges the routing of containers by road and sea while a_8' tracks the warehouses' incoming and outgoing flows. Second, a_9' loads the containers onto the vessel while a_{10}' tracks the clearance of the containers during potential stopovers. Upon cargo arrival to destination, a_{11}' unloads the containers from the vessel onto trucks. Finally, a_{12}' reports irregularities that, eventually, are issued by customs. Should there be any irregularities, the containers' sender would be subject to fines ending the process (a_{13}'). to the sender with fine payment ending the process (a_{13}'). Contrarily, a_{14}' arranges the routing of cargo by road while a_{15}' tracks containers between warehouses before delivery.

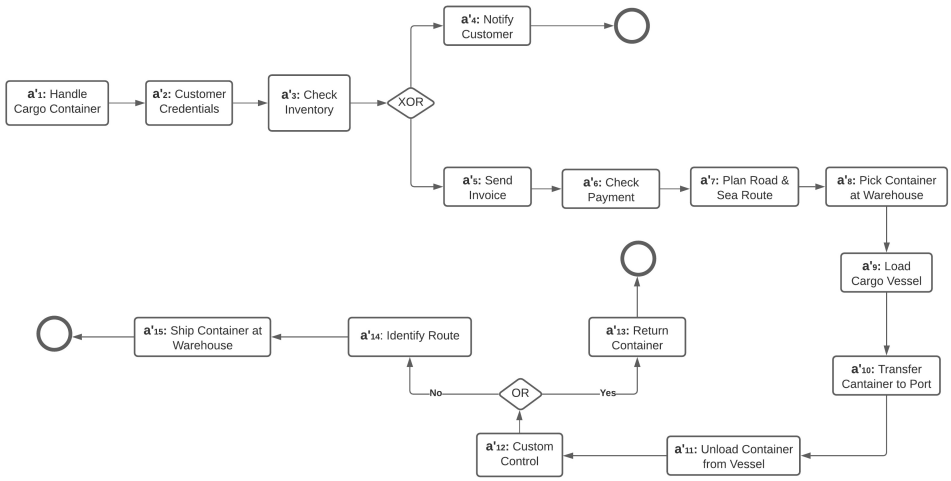


Fig. 10: BPMN-based process model for cargo tracking

In conjunction with specifying the cargo tracking process-model, we identified artefacts between activities as per Section 3.4 and adapted the Maritime Cargo Ontology (MCO) ([33], Appendix A) to define the semantic distance between activities as per Section 3.5. We submitted these three inputs to the *dependency-analysis* module that produces dependency values between activities with respect to the *control*, *data*, and *semantic* perspectives (Appendix B - Tables 17-21). After applying the collaborative clustering algorithm to the *control*, *data*, and *word-driven* dependencies, we obtained five candidate microservices namely, *Preparation*, *Handling*, *Planning&Tracking*, *Delivery*, and *Return*.

For the needs of benchmarking, we considered the values of the metrics presented in [15] and applied them to the candidate microservices (Table 14). For readability purposes, we merge the three sets of values, each associated with the corresponding approach, into Table 15.

Table 14: Metrics of candidate microservices using our approach - Cargo Tracking case

Metric / Microservice	*AC*	*EC*	*I*	*RC*	
Preparation	23	0	0	14	
Handling	18	3	0.14	11	
Planning&Tracking	16	2	0.11	23	
Delivery	12	8	0.4	5	
Return	7	5	0.4	-	
	15.2	3.6	0.21	13.25	*Avg*

Table 15: Metrics of candidate microservices per approach

	Metric / Microservice	*AC*	*EC*	*I*	*RC*	
[16]	*Location*	14	1	0.1	21.5	
	Tracking	11	3	0.2	16.7	
	Voyage&Planning	15	4	0.2	16.7	
		13.3	2.7	0.2	14.2	*Avg*
[4]	*Planning*	16	2	0.1	20.3	
	Product	13	4	0.2	1.8	
	Tracking	15	5	0.3	14.9	
	Trip	8	2	0.2	0	
		13	3.3	0.2	9.3	*Avg*
[26]	*Cargo*	13	4	0.2	1.8	
	Planning	10	3	0.2	11.5	
	Location	15	1	0.1	21.5	
	Tracking	16	5	0.2	14.1	
		13.5	3.3	0.2	12.2	*Avg*

Table 14 shows that our approach yields into five candidate microservices that are finer-grained compared to the four microservices in Baresi et al.'s and Li et al.'s approaches and the three microservices in Gysel et al.'s approach. Moreover, we observe that our approach's microservices are more strongly-cohesive compared to Baresi et al.'s and Li et al.'s approaches and less loosely-coupled compared to Gysel et al.'s approach. This demonstrates that our approach provides better results when considering both *AC* and *RC* compared to the three other approaches. In addition, there is no real distinction for *I* on average across all the approaches. As for *RC* metric, our approach performs better than Li et al.'s and Baresi et al.'s approaches, while less appealing than Gysel et al.'s approach. This can be explained as follows. Baresi et al.'s approach has some limitations due to their assumption that interfaces are well-defined and described with meaningful names whereas Li et al.'s approach constrains the analysts to make sure that they have a consistent viewpoint between the control and data perspectives due to lack of semantics associated with whether processes or data. Since our approach heavily relies on the domain ontology at different levels (concept and fragment), ensuring the ontology quality (e.g., completeness and accuracy) is critical for the decomposition results. To deal with the lack of ontology quality, additional factors like data usage context should be incorporated into our analysis for a better microservices identification.

5 Conclusion

Microservices identification remains an important obstacle that is undermining ICT practitioners' efforts when migrating existing applications to a better architectural style. Existing academic and industrial approaches exploit many sources like databases, log files, UML diagrams, and domain ontologies to identify microservices. Contrarily, we adopted business processes as a primary source offering a comprehensive view of what happens in organizations in terms of who does what, when, where, and why. Thanks to this comprehensive view, we designed and demonstrated a multi-model approach for microservices identification. These models are referred to as structural, data, and semantics, and capture dependencies between a BP's activities. We also proposed a clustering algorithm to collaboratively combine all extracted dependencies for the needs of identifying microservices. Our approach fosters decoupling and cohesion of future microservices and also considers to some extent the non-functional requirements via the data dependencies model. The results are promising showing how our collaborative clustering algorithm outperforms in term of precision the cooperative clustering algorithm. In term of future work, we would like to examine the appropriateness of other activity dependency models for microservices identification. For instance, it could be interesting to examine to what extent security requirements could impact the microservices identification. Also, it could be interesting to allow users to annotate their BPs and exploit such annotations with advanced machine learning techniques including Natural Language Processing to generate new activities dependencies. A third direction could be the exploration of BP's activities code to extract useful details that could be complement to the first models we already defined.

Bibliography

[1] Ahmadvand, M., Ibrahim, A.: Requirements reconciliation for scalable and secure microservice (de)composition. In: IEEE International Requirements Engineering Conference (RE). pp. 68–73. IEEE Computer Society (2016)

[2] Amiri, M.J.: Object-aware identification of microservices. In: 2018 IEEE International Conference on Services Computing (SCC). pp. 253–256. IEEE (2018)

[3] Barbosa, M.H.G., Maia, P.H.M.: Towards identifying microservice candidates from business rules implemented in stored procedures. In: 2020 IEEE International Conference on Software Architecture Companion, ICSA Companion 2020, Salvador, Brazil, March 16-20, 2020. pp. 41–48. IEEE (2020). https://doi.org/10.1109/ICSA-C50368.2020.00015, https://doi.org/10.1109/ICSA-C50368.2020.00015

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

[5] Beni, E.H., Lagaisse, B., Joosen, W.: Infracomposer: Policy-driven adaptive and reflective middleware for the cloud-ification of simulation & optimization workflows. Journal of Systems Architecture **95**, 36–46 (2019)

[6] Butzin, B., Golatowski, F., Timmermann, D.: Microservices approach for the internet of things. In: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA). pp. 1–6. IEEE (2016)

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

[8] Chung, J.Y., Chao, K.M.: A view on service-oriented architecture (2007)

[9] Cornuéjols, A., Wemmert, C., Gançarski, P., Bennani, Y.: Collaborative clustering: Why, when, what and how. Information Fusion **39**, 81–95 (2018). https://doi.org/10.1016/j.inffus.2017.04.008, https://doi.org/10.1016/j.inffus.2017.04.008

[10] Daoud, M., El Mezouari, A., Faci, N., Benslimane, D., Maamar, Z., El Fazziki, A.: Automatic microservices identification from a set of business processes. In: International Conference on Smart Applications and Data Analysis. pp. 299–315. Springer (2020)

[11] Davenport, T., Short, J.: The new industrial engineering: Information technology and business process redesign. Sloan Management Review (1990)

[12] Djogic, E., Ribic, S., Donko, D.: Monolithic to microservices redesign of event driven integration platform. In: 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2018, Opatija, Croatia, May 21-25, 2018. pp. 1411–1414 (2018). https://doi.org/10.23919/MIPRO.2018.8400254, https://doi.org/10.23919/MIPRO.2018.8400254

[13] Escobar, D., Cárdenas, D., Amarillo, R., Castro, E., Garcés, K., Parra, C., Casallas, R.: Towards the understanding and evolution of monolithic applications as microservices. In: 2016 XLII Latin American Computing Conference (CLEI). pp. 1–11. IEEE (2016)

[14] Estañol, M.: Artefact-centric Business Process Models in UML: Specification and Reasoning. Ph.D. thesis, Universitat Politècnica de Catalunya (2016)

[15] Fritzsch, J., Bogner, J., Zimmermann, A., Wagner, S.: From monolith to microservices: a classification of refactoring approaches. In: International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment. pp. 128–141. Springer (2018)

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

[17] Hammouda, K., Kamel, M.: Collaborative document clustering. In: SIAM International Conference on Data Mining (2006)

[18] Hassan, S., Bahsoon, R.: Microservices and their design trade-offs: A self-adaptive roadmap. In: 2016 IEEE International Conference on Services Computing (SCC). pp. 813–818. IEEE (2016)

[19] Jin, W., Liu, T., Zheng, Q., Cui, D., Cai, Y.: Functionality-oriented microservice extraction based on execution trace clustering. In: 2018 IEEE International Conference on Web Services (ICWS). pp. 211–218. IEEE (2018)

[20] Ke, W., Gong, X.: Collaborative hierarchical clustering in the browser for scatter/gather on the web. Proceedings of the American Society for Information Science and Technology **49**(1), 1–8 (2012)

[21] Knoche, H., Hasselbring, W.: Using microservices for legacy software modernization. IEEE Softw. **35**(3), 44–49 (2018). https://doi.org/10.1109/MS.2018.2141035, https://doi.org/10.1109/MS.2018.2141035
[22] Kolb, P.: Disco: A multilingual database of distributionally similar words. Proceedings of KONVENS-2008, Berlin **156** (2008)
[23] Kouroshfar, E., Shahir, H.Y., Ramsin, R.: Process patterns for component-based software development. In: International Symposium on Component-Based Software Engineering. pp. 54–68. Springer (2009)
[24] Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall PTR (2004)
[25] Levcovitz, A., Terra, R., Valente, M.T.: Towards a technique for extracting microservices from monolithic enterprise systems. arXiv preprint arXiv:1605.03175 (2016)
[26] Li, S., Zhang, H., Jia, Z., Li, Z., Zhang, C., Li, J., Gao, Q., Ge, J., Shan, Z.: A dataflow-driven approach to identifying microservices from monolithic applications. Journal of Systems and Software **157** (2019)
[27] Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR (2003)
[28] Mendez-Bonilla, O., Franch, X., Quer, C.: Requirements patterns for cots systems. In: Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008). pp. 232–234. IEEE (2008)
[29] Murtagh, F., Legendre, P.: Ward's hierarchical clustering method: clustering criterion and agglomerative algorithm. arXiv preprint arXiv:1111.6285 (2011)
[30] Paulsen, C., J.M., B., Bartol, Winkler, N.: Criticality analysis process model. Tech. rep. (2018)
[31] Schroer, C., Kruse, F., Marx Gómez, J.: A qualitative literature review on microservices identification approaches. In: Service-oriented Computing: 14th Symposium and Summer School on Service-oriented Computing, Summersoc. Springer (2020)
[32] Singh, V., Peddoju, S.K.: Container-based microservice architecture for cloud applications. In: 2017 International Conference on Computing, Communication and Automation (ICCCA). pp. 847–852. IEEE (2017)
[33] Villa, P., Camossi, E.: A description logic approach to discover suspicious itineraries from maritime container trajectories. In: Claramunt, C., Levashkin, S., Bertolotto, M. (eds.) GeoSpatial Semantics - 4th International Conference, GeoS 2011, Brest, France, May 12-13, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6631, pp. 182–199. Springer (2011)
[34] Wilkin, G., H., X.: A practical comparison of two k-means clustering algorithms. BMC bioinformatics **9** (2008)
[35] Wiśniewski, R., Karatkevich, A., Wojnakowski, M., et al.: Decomposition of distributed edge systems based on the petri nets and linear algebra technique. Journal of Systems Architecture **96**, 20–31 (2019)
[36] Wu, Z., Palmer, M.: Verb semantics and lexical selection. arXiv preprint cmp-lg/9406033 (1994)

A Cargo tracking additional description

In this appendix, we list all the elements related to the dependency analysis performed on cargo tracking. Meanwhile, Fig. 11 depicts an excerpt of the domain ontology for cargo tracking. For instance, in Table 16, a_9'(transfer container to port) applies *write* operation to Event_ID and Destination attributes, which leads to executing *create* operation whose outcome is Event artefact.

B Applying the 3 dependency models to Cargo tracking

Tables 17 and 18 depict an excerpt of the control and data dependencies for cargo tracking using Equations 7 and 8, respectively.

Tables 19, 20, and 21 depict an excerpt of semantic dependencies for Cargo tracking using Equations 13, 15, and 17, respectively.

Table 16: Cargo tracking's components

Activity	Artefacts	Attributes of artefacts
a_1'	Container (u)	Container_ID (r), Container_Destination (r), Container_Status (w)
	Customer (u)	Customer_ID (r), Customer_Status (w)
a_2'	Customer (u)	Customer_ID (r), Customer_Credit (r) , Customer_Status (w)
a_3'	Storage (u)	Storage_ID (r), Check_Storage_Capacity (w)
	Container (u)	Container_ID (r), Container_Destination (r), Container_Status (w)
a_4'	Event (c)	Event_ID (w), Event_Content (w)
	Customer (u)	Customer_ID (r)
a_5'	Invoice (u)	Invoice_Validity (w)
	Customer (u)	Customer_ID (r), Customer_Notification (w)
a_6'	Payment (u)	Check_Payment_ID (r)
a_7'	Container (u)	Container_ID (r), Container_Destination (r), Container_Status (r)
a_8'	Port (u)	Port_ID (r), Port_Destination (r)
	Container (c)	Container_ID (r), Container_Status (w)
a_9'	Vessel (u)	Vessel_ID (r)
	Container (c)	Container_ID(r), Container_Destination (r)
a_{10}'	Container (u)	Container_ID (r)
	Port (u)	Port_ID (r), Location (r)
	Event (c)	Event_ID (w), Location (w)
a_{11}'	Vessel (u)	Vessel_ID (r), Vessel_Destination (r)
a_{12}'	Control (u)	Control_ID (r), Update_Control (w)
	Customer (u)	Customer_ID (r)
a_{13}'	Event (c)	Event_ID (w), Event_Type (w)
a_{14}'	Trip (u)	Trip_ID (r), Customer_Destination(r)
	Storage (u)	Storage_ID (r), Storage_UseRate (w), Storage_Destination (r)
a_{15}'	Trip (u)	Trip_ID (r), Customer_ID (r)
a_{15}'	Event (c)	Event_ID (w), Container (w), Event_Type (w)
	Truck (u)	Truck_ID (r)

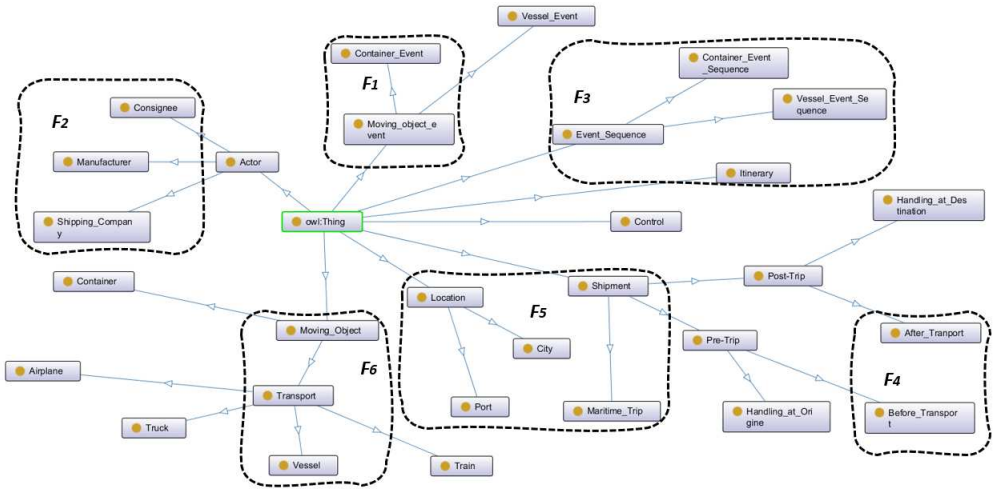


Fig. 11: Excerpt of Cargo tracking's domain ontology adapted from [33] [2011]

Table 17: Control dependencies with the occurrence probability (p) set to 0.5

Activity / Activity	a_1'	a_2'	a_3'	a_4'	a_5'	a_6'
a_1'	-	0.5	0.25	0.062	0.062	0.0312
a_2'	0.5	-	0.5	0.125	0.125	0.0625
a_3'	0.25	0.5	-	0.25	0.25	0.125
a_4'	0.062	0.125	0.25	-	0.25	0.125
a_5'	0.062	0.125	0.25	0.25	-	0.5
a_6'	0.0312	0.0625	0.125	0.125	0.5	-

Table 18: Excerpt of *data dependencies*

Activity / Activity	a_1'	a_2'	a_3'	a_4'	a_5'	a_6'
a_1'	0	0.1875	0.34375	0.03125	0.03125	0.21875
a_2'	0.1875	0	0	0.1875	0.4375	0
a_3'	0.3437	0	0	0	0	0.21875
a_4'	0.3437	0.1875	0	0	0.1875	0
a_5'	0.3425	0.4375	0	0.1875	0	0
a_6'	0.21875	0	0.21875	0	0	0

Table 19: Semantic dependencies between activities using *word-driven* technique

Activity / Activity	a_1'	a_2'	a_3'	a_4'	a_5'	a_6'
a_1'	1	0.0214	0.0186	0.0127	0.0217	0.0135
a_2'	0.0214	1	0.042	0.012	0.009	0.020
a_3'	0.0186	0.042	1	0.0019	0.266	0.120
a_4'	0.0127	0.012	0.0019	1	0.0043	0.0021
a_5'	0.0217	0.009	0.266	0.0043	1	0.0593
a_6'	0.0135	0.020	0.120	0.0021	0.0593	1

Table 20: Semantic dependencies between activities using *concept-driven* technique

Activity / Activity	a_1'	a_2'	a_3'	a_4'	a_5'	a_6'
a_1'	-	0.394576	0.397204	0.493221	0.493818	0.322241
a_2'	0.394576	-	0.326376	0.392990	0.393466	0.272318
a_3'	0.397204	0.326376	-	0.395607	0.396086	0.274131
a_5'	0.493818	0.393466	0.396086	0.491833	-	0.321335
a_6'	0.322241	0.272318	0.274131	0.320940	0.321335	-

Table 21: Semantic dependencies between activities using *fragment-driven* technique

Activity / Activity	a_1'	a_2'	a_3'	a_4'	a_5'	a_6'
a_1'	-	0.735274	0.968140	0.735274	0.970280	0.789005
a_2'	0.735274	-	0.724042	1,008878	0.753490	0.842985
a_3'	0.968140	0.724042	-	0.724042	0.953346	0.780517
a_4'	0.735274	1,008878	0.724042	-	0.753490	0.842985
a_5'	0.970280	0.753490	0.953346	0.753490	-	0.820096
a_6'	0.789005	0.842985	0.780517	0.842985	0.820096	-