Business Object Centric Microservices Patterns

Adambarage Anuruddha Chathuranga De Alwis[1], Alistair Barros[1], Colin Fidge[1], and Artem Polyvyanyy[2]

[1] Queensland University of Technology, Brisbane, Australia
{adambarage.dealwis,alistair.barros,c.fidge}@qut.edu.au
[2] The University of Melbourne, Parkville, VIC, 3010, Australia
artem.polyvyanyy@unimelb.edu.au

Abstract. A key impediment towards maturing microservice architecture conceptions is the uncertainty about what it means to design fine-grained functionality for microservices. Under a traditional service-oriented architecture (SOA), the unit of functionality for software components concerns individual business domain objects and encapsulated operations, enabling desirable architectural properties such as high cohesion and loose-coupling of its components. However, at present it is not clear how this SOA design strategy should be refined for microservices nor, more generally, how design considerations for different degrees of granularity apply, in a consistent and systematic way, for large SOA systems to smaller microservices. This paper proposes *microservice patterns*, as a contribution to the maturity of microservice architectures, through the refinement of the functional structure of SOAs. The patterns are derived by considering the splitting of business object (BO) operations and salient types of BO relationships, which influence software structure (as captured in UML): object association, exclusive containment, inclusive containment and specialisation (i.e., subtyping). The viability of the patterns for evolving large SOA systems into microservices is demonstrated through automated microservices discovery algorithms, on two open-source enterprise systems used widely in practice, Dolibarr and SugarCRM.

Keywords: Microservices patterns, microservice discovery, system re-engineering

1 Introduction

Microservice architectures are the latest development of distributed systems, evolving service-oriented architectures (SOAs) to enable high-performance, Internet-scale applications. Seen through the software transformations of Netflix^{TM}, Twitter^{TM}, eBay^{TM} and Amazon^{TM} among others, microservices are fine-grained software components which support individualised functionality of businesses, manage local and synchronised databases, and communicate via lightweight protocols.

To date, different ways of understanding microservices have emerged, notably from reported Internet player experiences [1] and more general software practitioner references [2, 3]. The agile development community emphasises microservices as a *unit of continuous development*, which can be separately developed, deployed, composed, and ultimately accrued into a collective software solution, that can provide *high scalability*, *availability* (replication), and execution *integrity*, when compared with monolithic systems. However, these non-functional properties are insufficient for a full understanding of a microservice architecture, which also requires precise insights concerning the functional structure of software in terms of software components, relationships, composability and deployment constraints.

Domain-driven design principles of software [4] have been indispensable for understanding the functional structure of software components in SOA, aligning it with the functionality of business domains. In particular, SOA components are structured to manage distinct business objects (BOs) of domains (e.g., order, customer, payment), containing encapsulated CRUD (create, read, update, and delete) operations of the objects [5]. Designed this way, SOA components can exhibit high intra-functional dependencies, (i.e., high cohesion) and low inter-functional dependencies with other components, (i.e., low coupling), allowing a systematic composition for larger components and applications. Emerging principles and patterns for microservices [6], which likewise appeal to domain-driven design, recommend decomposing BOs for support of fine-grained functionality (e.g., from an order to order entry, order tracking and order delivery) [2]. However, by appealing to business design considerations only, the implications for a precise functional structure required of a microservice architecture are left open.

In this paper we consolidate emerging developments of microservices towards an architecture style that is refined from the functional structure of SOAs (background in Section 2). Specifically, we present *patterns for microservices* (in Section 3), aligning, as done for the SOA paradigm, software components with BOs. To address the requirement for refined decomposition of BOs, the patterns support the splitting of BO operations into microservices (i.e., subsets of business object operations). In addition, our patterns address overall architectural coherence, and how microservices can be related to each other and be potentially co-located in Cloud containers, given BO intra- and inter-relationships. Four important BO relationships that pertain to software component design are considered (as expressed in the widely used Unified Modelling Language) [7]: BO association (relationship), BO exclusive containment (composition), BO inclusive containment (aggregation) and subtyping (specialisation). Taken together, we claim a crucial first step towards maturing the field of microservices through microservice patterns, in a similar vein to other IT fields [8–10].

While presenting the patterns for "greenfield" (new) development of software, we also demonstrate the value of the patterns for "brownfield" (evolving) developments, whereby the patterns are used to identify prospective microservices from the most prominent form of software used in businesses and utilising SOAs, i.e., enterprise systems (algorithms and validation are presented in Section 4 and Section 5). Related works (Section 6) and a conclusion (Section 7) are provided towards the end of the paper.

2 Systems Architecture Context

This section describes a systems architecture context in which microservices operate and is relevant for our definition of the microservice patterns (in Section 3). The context assumes that microservices are aligned with larger SOA-based systems such that they are either refactored parts of the systems or new designs whose processes are connected to backend SOA systems. Enterprise systems are used as exemplar SOA system, given their widespread use, and the examples we use are drawn from this software domain. As depicted in Fig. 1, an enterprise system consists of self-contained modules drawn from different subsystems (e.g., production management), and is deployed on a "backend" platform. Modules consist of a set of functions (e.g., software classes) managing one or more BOs through CRUD operations. For example ‘Software Module$_1$’ consists of several functions with operations manipulating the data related to ‘Shipment’, ‘Element’ and ‘User’ BOs in the database system. In this case, we have illustrated reduced cohesion and increased coupling, as the modules manage multiple BOs, unlike the desirable design of SOA components which should manage a single BO [7]. Through the database, different types of BO relationships are illustrated. These are: subtyping, where ‘At-Store Reclamation’ and ‘Generic Transport’ are specialised from ‘Shipment’; exclusive containment, where ‘Shipment’ is exclusively made up of ‘Element’ and controls its creation/deletion; inclusive containment, where ‘Order’ is inclusively made up of ‘Product’ but does not control its creation/deletion; and basic associations such as a ‘User’ has an ‘Order’.

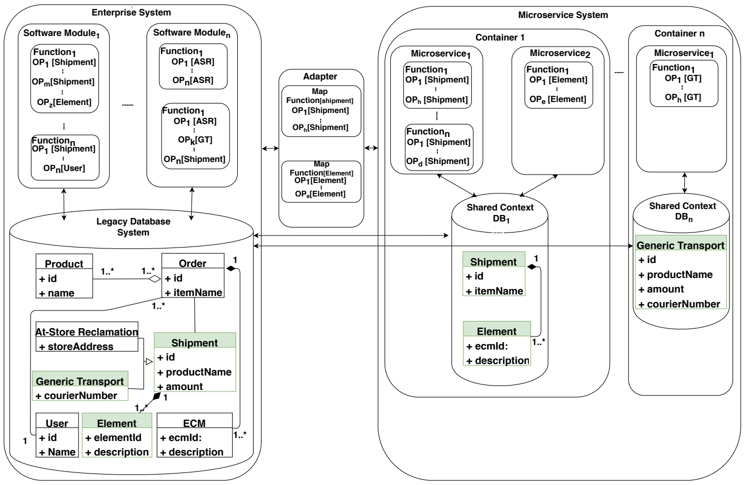

Enterprise System
Software Module1
Function1 OP1 [Shipment] OPm[Shipment] OPz[Element]
Functionn OP1 [Shipment] OPn[User]
Software Modulen
Function1 OP1 [ASR] OPn[ASR]
Function1 OP1 [ASR] OPk[GT] OPn[Shipment]
Legacy Database System
Product + id + name
Order + id + itemName
At-Store Reclamation + storeAddress
Shipment + id + productName + amount
Generic Transport + courierNumber
User + id + Name
Element + elementId + description
ECM + ecmId: + description
Adapter
Map Function[shipment] OP1[Shipment] OPh[Shipment]
Map Function[Element] OP1[Element] OPe[Element]
Microservice System
Container 1
Microservice1
Function1 OP1 [Shipment] OPh [Shipment]
Functionn OP1 [Shipment] OPd [Shipment]
Microservice2
Function1 OP1 [Element] OPe [Element]
Shared Context DB1
Shipment + id + itemName
Element + ecmId: + description
Container n
Microservice1
Function1 OP1 [GT] OPh [GT]
Shared Context DBn
Generic Transport + id + productName + amount + courierNumber

Fig. 1. Enterprise System Integrated with a Microservice system.

Through the microservice system (right-hand side of Fig. 1) aligned with the enterprise system, individual microservices, distributed in Cloud containers, have been introduced (in this case, refactored from the ES). They support subsets of individual BO operations. For example, Microservice$_1$ and Microservice$_2$ are responsible for subsets of ‘Shipment’ and ‘Element’ operations, respectively. In this example, the microservices are co-located on the same container because of the related BO exclusive containment relationship, which results in a dedicated coupling between them. The execution of operations across the enterprise and microservice systems are coordinated through business processes, which means that invocations of BO operations on the microservices trigger operations on enterprise system functions involving the same BOs and potentially related BOs through orchestration or choreography [11]. As per microservice systems, BO data is synchronised across databases managed by microservices and enterprise systems periodically.

Given this systems architecture context, we provide the following formalisation, which is used to define the microservice patterns (in Section 3) and pattern-based microservice discovery from enterprise systems (in Section 4). For this, we reuse the abstract representation of an enterprise system that we introduced previously [12].

Let $\mathbb{I}$ and $\mathbb{O}$ be *input* types and *output* types, respectively. Let $\mathbb{OP}$, $\mathbb{B}$, $\mathbb{T}$ and $\mathbb{A}$ be, respectively, a universe of *operations*, *BOs*, *database tables* and *attributes*. We characterize a *database table* $t \in \mathbb{T}$ by a collection of attributes, i.e., $t \subseteq \mathbb{A}$, while a *business object* $b \in \mathbb{B}$ is defined as a collection of database tables, i.e., $b \subseteq \mathbb{T}$.

An *operation op*, either of an enterprise or microservice system, is given as a triple (I, O, T), where $I \in \mathbb{I}^*$ is a sequence of *input types* the operation expects for input, $O \in \mathbb{O}^*$ is a sequence of *output types* the operation produces as output, and $T \subseteq \mathbb{T}$ is a set of *database tables* the operation accesses, i.e., either reads or augments.[3]

We define an enterprise system, or more precisely its abstract representation, as a finite automaton.

Definition 2.1 (Enterprise system, ES)
An *enterprise system* is a 5-tuple $(Q, \Lambda, \delta, q_0, A)$, where Q is a finite nonempty set of *states*, $\Lambda \subseteq \mathbb{OP}$ is a set of *operations*, such that Q and Λ are disjoint, $\delta : Q \times (\Lambda \cup \{\tau\}) \rightarrow \mathcal{P}(Q)$ is the *transition function*, where τ a is a special *silent operation* such that $\tau \notin Q \cup \Lambda$, $q_0 \in Q$ is the *start state*, and $A \subseteq Q$ is the *set of accept states*.[4] ⌟

We refer to computations of an enterprise system as *processes*, or *process instances* of the system. Finally, a microservice system is defined as follows.

Definition 2.2 (Microservice system, MS)
Let $\mathbb{C}$ and $\mathbb{M}$ be a universe of containers and a universe of MSs, repsectively. A *MS* is a 4-tuple (C, M, σ, μ), where $C \subseteq \mathbb{C}$ is a set of *containers*, $M \subseteq \mathbb{M}$ is a set of *microservices*, $\sigma : C \rightarrow \mathcal{P}(M) \setminus \emptyset$ is a *deployment function* that maps each container $c \in C$ onto a non-empty set of microservices $\sigma(c)$ that are deployed on c, and $\mu : M \rightarrow \mathcal{P}(\mathbb{OP}) \setminus \emptyset$ is a *microservice definition function* that maps each microservice $m \in M$ onto a non-empty set of operations.

3 Microservices Patterns

We present four Microservice (MS) patterns related to intra- and inter-BO relationships that form important design considerations for software components [7]. As detailed above, each pattern identifies a microservice as a collection of operations that manipulate data related to a single BO.

Pattern 3.1 (Microservice Subtyping)
A microservice is used to manage subsets of operations related to a BO subtype. Since there can be multiple BO subtypes of a supertype, a MS can apply to any given subtype, at any level of a subtype hierarchy and a subset of operations related to that subtype.

More formally, let $b_1, b_2 \in \mathbb{B}$ be two BOs such that b_2 is a subtype of b_1, at any level of the subtype hierarchy. A MS m reflects the *microservice subtype pattern* for b_2 and its supertype b_1, iff for each operation $(I, O, T) \in \mu(m)$ it holds that $b_2 \cap T \neq \emptyset$. Support of the other operations that relate to b_2 not covered by m, i.e., $(I, O, T) \notin \mu(m)$ where $b_2 \cap T \neq \emptyset$, is subject to systems architecture considerations, inclusive of the ES and other MSs. We distinguish two variants of the subtype pattern:

[3] A^* denotes the application of the Kleene star operation to set A. [4] Given a set A, by $\mathcal{P}(A)$ we denote the powerset of A.

- *Data population partition:* MS m reflects the *data population partition* variant of the *subtype pattern* iff $b_1 \subseteq \bigcup_{(I,O,T)\in\mu(m)} T$. That is, m involves operations over all the tables b_1 has access to.
- *Data typing partition:* MS m reflects the *data typing partition* variant of the *subtype pattern* iff (i) $b_1 \subseteq \bigcup_{(I,O,T)\in\mu(m)} T$ and (ii) for every BO $b_3 \neq b_2$, such that b_3 is a subtype of b_1 at the same hierarchy level as b_2, it holds that $(b_3 \setminus b_1) \cap (\bigcup_{(I,O,T)\in\mu(m)} T) = \emptyset$. That is, m reflects the data population partition and, in addition, does not involve operations over the tables that are specific to other subtypes of b_1 at the same level of the subtype hierarchy as b_2.

Examples: As depicted in Fig. 1, BO 'Shipment' has two subtypes 'At-Store Reclamation' and 'Generic Transport' that have dedicated attributes *storeAddress* and *courierNumber*, respectively, and common attributes *id*, *productName*, and *amount*. Partitioning of the BOs occurs at the attribute level, to which the MS subtyping variant, *data typing partition*, can be applied (see the MS system in Fig. 1). If, on the other hand, a single attribute in shipment BO is used to indicate whether it is an 'At-Store Reclamation' or a 'Generic Transport', partitioning of the BOs occurs at the data level, to which the *data population partition* variant can be applied.

Problem: Developing a single microservice to manage subtype BOs will result in services which do not follow the single responsibility rule [2] in MS development. Furthermore, this can result in increased request processing time due to the fact that a request related to multiple subtypes are processed through the same service endpoint. Consequently, the unavailability of a single service endpoint may result in the unavailability of all subtype services.

Solution: The pattern addresses selective implementation through MSs of BO subtypes, with BOs abstracted over database tables. Implementations on the database level can vary, ranging from a single table storing the supertype and subtypes of BOs to individual tables per supertype and each subtype, to combinations of both. This in turn can give rise a considerable set of CRUD operations related to the supertype and each subtype of the BO, potentially resulting in software inconsistencies as to which operations are provided as part of the different MSs. Such a problem is compounded when the structure of BOs change, resulting in changes to the different operations. A way of handling is to include a standard set of operations for handling all the supertype and subtypes and allowing for a qualification (e.g. a special parameter) when the operations are called to indicate which subtype is required. Each operation then results in the calling of a data handling MS which contains logic for determining which parameters are required for each different subtype. This way, the BO attribute sets are mapped to different BO subtypes in the same, reusable core MS.

Pattern 3.2 (Microservice Exclusive Containment)
Description: Microservices manage subsets of operations of BOs which have an exclusive containment relationship. One MS relates to the parent (or composite) BO and other MSs manage child BOs, such that the existence (i.e., create, update and delete) of a child BO depends on the existence of the parent BO. MSs should be co-located to run on the same (execution) container, given their tight-coupling resulting from the existence of the corresponding parent and child BOs.

More formally, let $b_1, b_2 \in \mathbb{B}$ be two BOs such that b_2 is exclusively contained in b_1. MSs m_1 and m_2 reflect the *exclusive containment pattern* for b_1 and b_2 iff:

- for each operation $(I, O, T) \in \mu(m_1)$ it holds that $b_1 \cap T \neq \emptyset$,
- for each operation $(I, O, T) \in \mu(m_2)$ it holds that $b_2 \cap T \neq \emptyset$,
- m_1 and m_2 includes object existence operations related to BOs of type b_1 and b_2 respectively,
- BOs of type b_2 are created only after the corresponding BO of type b_1 is created,
- a BO of type b_1 is deleted only after all the corresponding BOs of type b_2 are deleted, and
- updates of a BO of type b_2 always come after that BO is created.

We distinguish two variant of exclusive containment pattern:

- *Mandatory exclusive containment:* The invocation of create and delete operation related to the parent BO b_1 requires that the corresponding child BO's b_2 operations (I, O, T) *must* also be done simultaneously, as part of the same process instance.
- *Optional exclusive containment:* The invocation of create and delete operation related to the parent BO b_1 does not requires that the corresponding child BO's b_2 operations (I, O, T) to be invoked.

Examples: When considering a 'Shipment' BO, as depicted in Fig. 1, the 'Element' BO is contained in the 'Shipment' and the creation of the 'Shipment' BO always requires creating one or more 'Element' BOs. Similarly, deleting the 'Shipment' BO requires deleting the 'Element' BO, resulting in mandatory exclusive containment. As such, the *mandatory exclusive containment* variant can be applied when designing MSs (see the microservice system in Fig. 1). If, on the other hand, creation of a 'Shipment' does not always require having an 'Element' (i.e., 'Shipment' and 'Element' have a zero or more relationship) then *optional exclusive containment* can be applied.

Problem: Developing separate microservices contained in separate microservice containers to manage exclusively contained BOs may result in increased communication overhead between services. In addition, separating the data related to parent BOs and child BOs into separate microservices may result in data inconsistencies and unavailability.

Solution: Given two variants of exclusive containment the development of the MS should be done while checking the last three conditions against all the processes the ES supports as below.

- *Mandatory exclusive containment:* This can be supported in three ways. Firstly, through the MS managing the parent object (i.e., m_1), the parent object b_1 is created and invocations are made on the child MSs (i.e, m_2) to create the child objects b_2 (i.e., concurrently). Secondly, through a process context, the parent object's b_1 creation using the parent MS m_1 and the child objects' b_2 creations using the child MSs m_2, occur in the same process. The process allows the latter to occur at some point following the former (i.e., eventually) but before the end of the process instance. Thirdly, if there are multiple instances of a child object b_2, then the instance after the first can be created through an update operation on the child MS m_2. In the case of object deletion, through the MS managing the parent object (i.e., m_1), invocations are made on the child MSs (i.e., m_2) to delete any child object b_2 as a part of parent's deletion process (i.e., concurrently) and to delete the parent object b_1.

- *Optional exclusive containment:* The same create and delete mechanisms described for mandatory inclusive containment (refer Pattern 3.3) also apply here, with the exception that the creation and deletion of child objects b_2 and therefore corresponding invocations of child MSs m_2 are optional.

Pattern 3.3 (Microservice Inclusive Containment)
Description: Microservices manage subsets of operations of BOs which have an inclusive containment relationship. One MS relates to the parent (or aggregate) BO and other MSs manage child BOs, such that the existence of children does not depend on the parent's existence, i.e., the children exist independently but are used in the context of the parent. No dedicated tight-coupling, as such, exists between the MSs operating on parent and children objects, and therefore there is no requirement to strictly co-locate them for execution.
More formally, let $b_1, b_2 \in \mathbb{B}$ be two BOs such that b_2 is inclusively contained in b_1. MSs m_1 and m_2 reflect the *inclusive containment pattern* for b_1 and b_2 iff:

- for each operation $(I, O, T) \in \mu(m_1)$ it holds that $b_1 \cap T \neq \emptyset$,
- for each operation $(I, O, T) \in \mu(m_2)$ it holds that $b_2 \cap T \neq \emptyset$,
- m_1 and m_2 includes create, read, update and delete operations related to BOs of type b_1 and b_2 respectively,
- m_1 contains an operation for reading BOs of type b_2, and
- each time a BO of type b_1 is created, it reads a BO of type b_2.

Variants:

- *Mandatory inclusive containment:* The invocation of create and delete operations related to BO b_1 of parent MS m_1 requires that the corresponding child BOs b_2 *must* also exist and be used in the parent's aggregation, despite their independent existence. As such parent MS m_1 will call child MS m_2 for information retrieval.
- *Optional inclusive containment:* The invocation of create and delete operations related to BO b_1 of parent MS m_1 does not require invocation of operations related to the child MS m_2 corresponding to child BOs b_2.

Examples: The 'Order' BO as depicted in Fig. 1 has an inclusive containment relationship with the 'Product' BO, because an order should contain a list of products which are ordered, which results in a mandatory inclusive containment relationship. As such the *mandatory inclusive containment* variant can be applied to design MSs. If, on the other hand, creating an 'Order' is not always require creating a 'Product' (i.e, 'Order' and 'Product' have a zero or more relationship), then the *optional inclusive containment* variant can be applied to design MSs.

Problem: Developing a single microservice to manage inclusively contained BOs will result in bigger services which does not follow the single responsibility rule in MS development and can result in difficulty of maintenance due to the fact that it is responsible for the operations related to multiple BOs. Furthermore, this can result in service unavailability of multiple BOs if the hosted microservice fails. When considering the scalability aspect, this will result in reduced scalability due to the fact that it requires more resources such as memory and CPU to provide the services.

Solution: Given two variants of inclusive containment the development of the MSs should be done while checking the last three conditions against all the processes the ES supports as below.

- *Mandatory inclusive containment:* When the parent MS m_1's create operations are invoked, read operations are invoked on the child MS m_2 and corresponding object references are held in the parent MS m_1. These references can be created concurrently (as part of the parent's create operation) or sometime before the end of the process instance (as part of a process context which includes the creation of parent object b_1 and child object b_2 references). If there are multiple instances of a specific child object b_2, its object reference is used to update the list of instances in the parent aggregate using reads on the child MS m_2. When the parent MS m_1's delete operations are invoked, object references to the child BOs b_2 are thereby deleted.
- *Optional inclusive containment:* The same create and delete mechanisms described for mandatory inclusive containment also apply here, with the exception that the creation and deletion of child object b_2 references the parent b_1 and therefore corresponding invocations of child MS m_2 existence operations are optional.

Pattern 3.4 (Microservice Association)
Description: Two microservices manage operations of BOs which have an association relationship. One MS relates to one BO and the other MS manages another BO, such that the existence of both BOs are independent of each other. As there is no dedicated tight-coupling, there is no requirement to strictly co-locate them.
More formally, let $b_1, b_2 \in \mathbb{B}$ be two BOs in an association relationship. MSs m_1 and m_2 reflect the *association pattern* for b_1 and b_2 iff:

- for each operation $(I, O, T) \in \mu(m_1)$ it holds that $b_1 \cap T \neq \emptyset$,
- for each operation $(I, O, T) \in \mu(m_2)$ it holds that $b_2 \cap T \neq \emptyset$,
- m_1 includes operations over BOs of type b_1,
- m_2 includes operations over BOs of type b_2, and
- m_1 contains operations which reference m_2 for reading BOs of type b_2.

Examples: 'Order' and 'User' BOs can have an association relationship as depicted in Fig. 1. In this situation one 'User' can have one or more 'Orders' related to it. The create, read, update, and delete operations on the values related to each BO can occur independently of each other even though they are related. However, some create or update operations related to 'Order' might require the 'User' information which leads to a read execution of the BO 'User'.

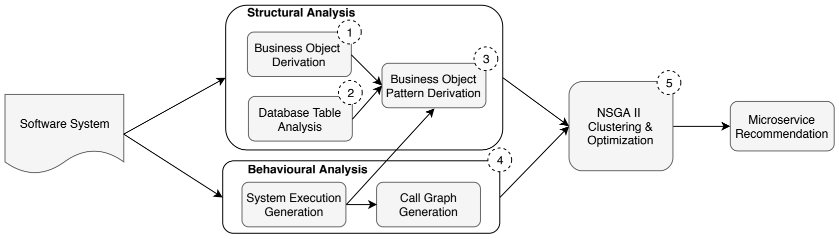
Problem: Developing a single microservice to manage associated BOs will result in bigger services which result in difficulty of maintenance due to the fact that it is responsible for the operations related to multiple BOs. Furthermore, this can result in service unavailability of multiple BOs if the hosted microservice fails. When considering the scalability aspect, this will result in reduce scalability due to the fact that it requires more resources such as memory and CPU to provide the services.

Solution: Develop separate MSs m_1 and m_2 to manage associate BOs b_1 and b_2, such that the CRUD operation of one MS may invoke CRUD operations related to the other MS.

4 Utilisation of MS Patterns in ESs Re-engineering

In this section, we demonstrate the added value of using our microservice patterns in the context of re-engineering parts of enterprise systems as microservices (i.e., “brownfield” development) (refer to Fig. 2).

In the first step, we identify the BOs in the given enterprise system by evaluating the SQL queries and relationships between database tables according to the method described by Nooijen *et al.* [13]. In the second step, we analyse the database tables and their cardinalities in order to derive the mandatory and optional containment relationships. In the third step, we derive the BO relationships based on the BOs derived from step one, cardinalities derived from step two and system execution logs, as detailed in Section 4.1. In the fourth step, we generate the call graphs based on the execution logs which were generated in the previous step. Finally, in the fifth step, all the structural and behavioural details generated are provided to an optimization algorithm (Non-dominated Sorting Genetic Algorithm (NSGA II)) which suggests the best splitting of subset of operations related to BOs, by considering the four major factors namely, operations’ relationships with BOs, BO relationships, execution frequencies of operations and their execution patterns.



Structural Analysis
1
Business Object Derivation
2
Database Table Analysis
3
Business Object Pattern Derivation
Software System
Behavioural Analysis
4
System Execution Generation
Call Graph Generation
5
NSGA II Clustering & Optimization
Microservice Recommendation

Fig. 2. Overview of the microservice discovery approach.

4.1 Business Object Pattern Derivation

As depicted in Fig. 2, to derive microservice recommendations, BO pattern derivation should be achieved in the third step. This is done using two algorithms. The first algorithm derives the subtyping patterns as detailed in our previous work [12]. The second algorithm (i.e., Algorithm 1) is detailed in this section. It derives inclusive containment, exclusive containment and association patterns using eight steps.

As described in Section 2, BOs are often stored across several database tables. Thus, in the first step, the *BOS* function is performed by Algorithm 1, which derives the set of all BOs *B* of the system through the analysis of the database table relationships and their data similarities, as described by Nooijen *et al.* [13]. In the second step, the algorithm extracts all the operations *OPS* related to the system through static analysis. This mainly involves the invocation of function *STATOPEX* which uses Abstract Syntax Tree (AST) parsing to process the source code *SC* of the system to extract all the CRUD operations related to the system. Generally, in *join sql operations*, database tables are referred to by different names, other than their original names in the database (e.g., ‘cus’ is used to refer to the ‘customer’ database table). As such, the details (*TNM*) related to such tables (i.e., table names and alias names) are provided to the function for processing. Function *STATOPEX* extracts all the operations related to the given system and uses a mapping function to map table names to alias names (*TNM*) when there are join operations. In the third step of the algorithm, function *DYNOPEX* is executed to extract the operations from the execution logs (line 3). In the fourth step, function *DYNTM* computes the execution time of each dynamic operation as per the log. This allows to classify the operations which execute together, by analysing the execution sequences and execution times in the logs. According to Andrews *et al.* [14], such classification helps to derive relationships between operations. Apart from operation extraction, both functions *STATOPEX* and *DYNOPEX* identify and store the database tables related to each operation. In the fifth step, Algorithm 1 classifies the static operations extracted in step three into association create (OP_a^c) and association delete (OP_a^d) operations (lines 6–13). Generally, in a database, there can be foreign key relationships among the tables that relate to the same BO. Such relationships are not important for the containment derivation. As such, in this step, the *TBLS* function extracts the tables T_k related to each association operation op_k. Next, after confirming that the tables in T_k do not relate to the same BO, the algorithm adds the association operation to the respective set, either OP_a^c or OP_a^d. Using a similar method, in the sixth step (line 14), the dynamic operations are classified by the function *DCLS* into association create (OP_{da}^c), association delete (OP_{da}^d), create (OP_d^c), update (OP_d^u), read (OP_d^r), and delete (OP_d^d) operations.

In the seventh step, Algorithm 1 identifies the exclusively contained, inclusively contained and associated BOs (lines 15–22). Exclusively contained BOs (γ) are identified by verifying that two BOs are related by create (OP_{da}^c) and delete (OP_{da}^d) association operations, or related by update (OP_{da}^u) and delete (OP_{da}^d) association operations, or related by create (OP_d^c) and delete (OP_d^d) operations, or related by update (OP_d^u) and delete (OP_d^d) operations while not having any other operations in OP_a^c and OP_a^d that govern the relationship between b_i and b_j (lines 16–17). Similarly, inclusively contained BOs (β) are derived if there are two BOs b_i, b_j in which b_j is created or read before updating the values in b_i related to it while not having any delete association operations in OP_a^d and OP_{da}^d between them (lines 18–19). Note that in inclusive and exclusive containment derivation, the algorithm uses the time sequences related to the dynamic operations to identify the execution patterns of the operations. For example, if there is a create operation of a BO which is immediately followed by a create operation of another BO, there is a possibility of exclusive containment given that the other criteria match. The BOs which are not in the inclusive or exclusive containment relationships are categorised as associated BOs (δ) (lines 20–21).

In the final step of the algorithm, function *MANDOROP* analyses the constructed inclusive and exclusive relationships and verifies which BOs have mandatory or optional relationship (line 23). Function *MANDOROP* requires information about foreign keys and primary keys of database tables (*TDATA*), because different developers use different patterns to define foreign key and primary key relationships. Based on the given keys, mandatory relationship is identified if for every record in BO b_i there is at least one record in BO b_j. However, if there are no records related to items of b_i in b_j, it is considered as an optional relationship.

Algorithm 1: Computing BO relationship patterns

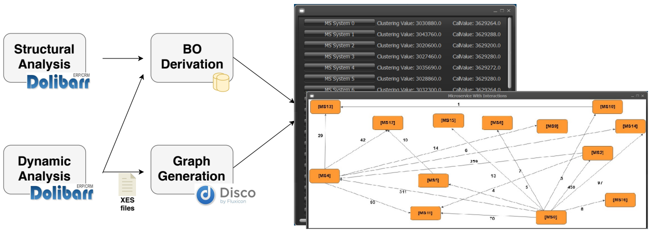
```
Input: Source Code $SC$, Execution Logs $EL$, Database Schema $DB$, Alias
       names $TNM$, Database Table Data $TDATA$ of an ES $s$.
Output: $\gamma, \beta, \delta$ are binary relations over BOs that capture exclusive
        containment, inclusive containment and association, respectively, $\alpha$
        captures mandatory and optional types.
1  $B = \{b_1, \ldots, b_n\} := BOS(SC, DB)$;                                  // Identify BOs
2  $OPS = \langle op_1, \ldots, op_m \rangle := STATOPEX(SC, TNM)$;            // static analysis
3  $OPD = \langle opd_1, \ldots, opd_n \rangle := DYNOPEX(EL, TNM)$;           // dynamic analysis
4  $OPT = \langle opt_1, \ldots, opt_n \rangle := DYNTM(EL, TNM)$;             // dynamic operation time
   analysis
5  $OP^c_a := OP^d_a := OP^c_{da} := OP^d_{da} := OP^c_d := OP^u_d := OP^r_d := OP^d_d := \emptyset$;
6  for each $k \in [1 .. m]$ do
7      if $op_k$ is an association operation then
8          $T_k := TBLS(op_k)$;
9          if $\nexists b \in B . T_k \subseteq b \wedge op_k$ is a create operation then
10             $OP^c_a := OP^c_a \cup \{op_k\}$;             // Identify an association create operation
11         else if $\nexists b \in B . T_k \subseteq b \wedge op_k$ is a delete operation then
12             $OP^d_a := OP^d_a \cup \{op_k\}$;             // Identify an association delete operation
13 end
14 $\langle OP^c_d, OP^u_d, OP^r_d, OP^d_d, OP^c_{da}, OP^d_{da} \rangle := DCLS(OPD)$;
15 for each $b_i, b_j \in B$ where $i \neq j$ do
16     if $((\exists op \in OP^c_{da} . op \in OP^d_{da}) \vee (\exists op \in OP^u_{da} . op \in OP^d_{da}) \vee$
        $(\exists op^i_d \in OP^c_d . \exists op^j_d \in OP^d_d . op^i_d = op^j_d) \vee$
        $(\exists op^i_d \in OP^u_d . \exists op^j_d \in OP^d_d . op^i_d = op^j_d)) \wedge (\nexists op \in OP^c_a \wedge op \in OP^d_a)$ then
17         Record in $\gamma$ that $b_i, b_j$ are exclusively contained;
18     else if $((\exists op^i_d \in OP^c_d \vee \exists op^i_d \in OP^r_d) \wedge \exists op^j_d \in OP^u_d \cup OP^c_d . op^i_d < op^j_d) \wedge$
        $(\nexists op \in OP^d_a \wedge \nexists op \in OP^d_{da})$ then
19         Record in $\beta$ that $b_i, b_j$ are inclusively contained;
20     else
21         Record in $\delta$ that $b_i, b_j$ are associated;
22 end
23 $\alpha := MANDOROP(TDATA, \gamma, \beta)$;
24 return $\gamma, \beta, \delta, \alpha$;
```

After obtaining the BO relationships using Algorithm 1 we provide that information with the call graphs generated from system executions (see step 4 in Fig. 2) to our genetic optimization algorithm, which is detailed in our previous work [15]. This algorithm clusters the executed operations in the graphs based on their relationships to the BOs while minimizing the communication overhead between different clusters (i.e., the number of calls between operations of different clusters would be minimal). These clusters are then provided to the users as recommendations to create microservices to optimize the system's performance.

5 Implementation and Validation

To demonstrate the applicablity of the patterns for re-engineering of legacy software systems, we developed a prototype[5] which uses the patterns to discover prospective microservices from two enterprise systems: Dolibarr[6] and SugarCRM[7]. The validity of the discovered microservices was evaluated by comparing their performance against the enterprise systems, considering five system characteristics, namely, scalability, availability, execution efficiency, cohesion and coupling.

A detailed description of the experiments conducted with Dolibarr is presented here. (The experimental details related to another case study using SugarCRM can be found in our technical report [20]). The Dolibarr system contains 10,735 files and around 3,000 attributes divided between 250 tables. We performed the static analysis of the Dolibarr source code and identified the BOs related to the system. Meanwhile, Selinium[8] scripts were used to perform dynamic analysis and generation of execution sequences. For Dolibarr we performed executions related to 'product purchase and sales', covering all the major functionalities. The execution logs were captured by customizing the log generation functionality of the system. The logs were then converted to XES file format and analyzed using the process mining tool Disco[9], as depicted in Fig. 3, to generate the call graph. The call graph generated for Dolibarr contained 301 unique nodes. Each node in the call graph represents a unique operation performed on database tables by the system and edges between the nodes represent the number of calls between the nodes. The execution graphs and the identified BOs were fed into our prototype to discover microservices and their interactions, as depicted in Fig. 3.



```
MS System 0    Clustering Value: 3030880.0    CallValue: 3629264.0
MS System 1    Clustering Value: 3043760.0    CallValue: 3629288.0
MS System 2    Clustering Value: 3020600.0    CallValue: 3629200.0
MS System 3    Clustering Value: 3027460.0    CallValue: 3629280.0
MS System 4    Clustering Value: 3035690.0    CallValue: 3629272.0
MS System 5    Clustering Value: 3028860.0    CallValue: 3629280.0
MS System 6    Clustering Value: 3032300.0    CallValue: 3629264.0
```

Fig. 3. Microservice discovery steps with related tools.

Discovered MSs: Based on the provided data, the prototype managed to identify 41 different business objects related to Dolibarr, for example 'User', 'Shipment', 'Element', 'ECM', 'Order', 'Third-party', 'Account', and 'Product'. Furthermore, system execution analysis led to the identification of *mandatory exclusive containment relationship* between 'Order' and 'ECM', and 'Element' and 'Shipment' and *mandatory inclusive containment relationship* between 'Order' and 'Thirdparty', 'Shipment' and 'Thirdparty', and 'Order' and 'Product' BOs. Also, the prototype identified two *subtypes* related to shipments named 'At-Store Reclamation' and 'Generic Transporter'. The relationship between 'User' and 'Order' was identified as an *association* relationship. Based

[5] https://github.com/AnuruddhaDeAlwis/BORelationshipDerivation.git
[6] https://www.dolibarr.org/ [7] https://www.sugarcrm.com/ [8] https://www.seleniumhq.org/
[9] https://fluxicon.com/disco/

on the containment and association relationship patterns and the execution sequences the prototype derived eleven microservice recommendations for Dolibarr.

Validation Process: The validation of the microservice recommendations was conducted in two steps. First, we evaluated the improvement of cohesion and coupling of different modules when clustering the classes based on recommendations provided by the prototype. This was achieved through measuring the Lack of Cohesion (LOC) and Structural Coupling (StrC) of the clusters, as described by Candela *et al.* [18]. We calculated the corresponding values for the enterprise system by clustering the classes into folders while preserving the original package structure, and then calculated the same values for the systtem with microservices. We have only considered six microservices in these calculations out of 11 recommendations. The obtained values are reported in Tables 1 and 2.

Then, we evaluated the system's performance improvement. In order to validate performance, we hosted the original Dolibarr system on a AWS cloud. [10] We used 2 EC2 instances which individually contained one virtual CPU and memory of 1GB. The data related to the system was hosted in a MySQL relational database on AWS, which had one virtual CPU and storage of 20GB. This implementation can be seen as the 'Enterprise System' depicted in Fig. 4. This system was then tested against 100 and 200 executions generated by 4 machines simultaneously, simulating customer requests, while recording their total execution time, average CPU consumption, and average network bandwidth consumption. For Dolibarr, we simulated the functionality related to *product purchase and sales*, using Selenuim[11] scripts which executed the system similar to a real environment. The values we obtained for Dolibarr enterprise system are detailed in Table 3.

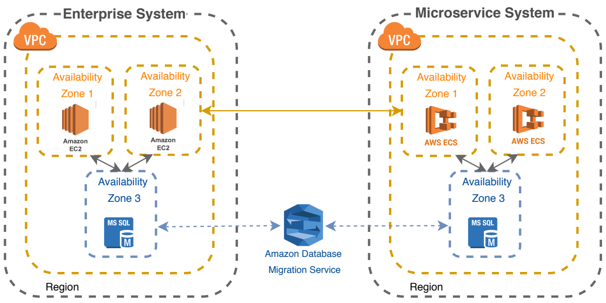


Fig. 4. System implementation in AWS.

Next, we introduced the microservices we have identified to the system to evaluate the effectiveness of the patterns. First, we evaluated the effectiveness of the 'Subtyping' pattern. We conducted two experiments. The first experiment was conducted by introducing a single microservice which contains both 'At-Store Reclamation' and 'Generic Transporter' BOs and a subset of the operations related to them. The second experiment was conducted by introducing individual microservices for 'At-Store Reclamation' and

[10] https://aws.amazon.com/ [11] https://www.seleniumhq.org/

‘Generic Transporter’ BOs and the subsets of the operations related to them. Similarly two experiments were conducted regarding inclusive containment of ‘Order’ and

Table 1. Dolibarr ES vs MS System (Lack of Cohesion).

System Type	thirdparty	order	ecm	shipment	product	user
Original ES	62	17	4	7	217	17
MSs	42	17	4	7	81	17

Table 2. Dolibarr ES vs MS System (Structural Coupling).

System Type	thirdparty	order	ecm	shipment	product	user
Original ES	23	27	7	20	34	18
MSs	22	27	7	20	26	18

Table 3. Legacy Enterprise System Result for Dolibarr.

System Type	No of Executions	No of Packets Sent	Ex. Time (s)	Avg CPU Util (EC2)	Avg CPU Util (DB)	Avg Network (Kb/s)
Original ES	100	136452	8220	9.535	2.37	47.31
Original ES	200	163270	17400	8.805	2.13	45.31

Table 4. Dolibarr System. (1) All Subtypes in one MS. (2) Subtypes in separate MSs.

System Type	No of Executions	No of Packets Sent	Ex. Time (s)	Avg CPU Util (EC2)	Avg CPU Util (DB)	Avg Network (Kb/s)
ES & Single MS (1)	100	139860	8400	5.288	1.71	30.585
ES & Single MS (1)	200	162099	16740	4.498	1.665	19.55
ES & Multi MS (2)	100	124188	7860	8.543	1.597	28.73
ES & Multi MS (2)	200	168990	15720	8.345	1.64	26.57

Table 5. Dolibarr System. (1) Exclusive BOs in one MS. (2) Exclusive BOs in separate MSs.

System Type	No of Executions	No of Packets Sent	Ex. Time (s)	Avg CPU Util (EC2)	Avg CPU Util (DB)	Avg Network (Kb/s)
ES & Single MS (1)	100	110789	7980	5.045	1.67	26.75
ES & Single MS (1)	200	181828	15720	5.228	1.55	21.71
ES & Multi MS (2)	100	117912	8160	6.127	1.87	25.77
ES & Multi MS (2)	200	165273	16020	5.613	2.11	18.48

Table 6. Dolibarr System. (1) Inclusive BOs in one MS. (2) Inclusive BOs in separate MSs.

System Type	No of Executions	No of Packets Sent	Ex. Time (s)	Avg CPU Util (EC2)	Avg CPU Util (DB)	Avg Network (Kb/s)
ES & Single MS (1)	100	95499	7860	5.04	1.75	25.795
ES & Single MS (1)	200	135182	15780	5.05	1.75	21.695
ES & Multi MS (2)	100	105996	7920	3.702	1.907	25.81
ES & Multi MS (2)	200	136371	15720	2.633	1.653	14.58

Table 7. Dolibarr System. (1) Association BOs in one MS. (2) Association BOs in separate MSs.

System Type	No of Executions	No of Packets Sent	Ex. Time (s)	Avg CPU Util (EC2)	Avg CPU Util (DB)	Avg Network (Kb/s)
ES & Single MS (1)	100	109052	8220	5.123	1.71	26.655
ES & Single MS (1)	200	124343	18060	3.1	1.415	13.275
ES & Multi MS (2)	100	227010	8280	4.318	1.987	39.67
ES & Multi MS (2)	200	252591	16140	3.017	1.837	21.14

Table 8. Legacy vs MS System characteristics comparison for Subtyping.

Campaign Type	Scalability [CPU EC2]	Scalability [CPU DB]	Scalability [Network DB]	Availability [100]	Availability [200]	Efficiency [100]	Efficiency [200]
Original ES	3.458	3.366	3.586	99.27	99.31	1.000	1.000
ES & Single MS	2.915	3.336	2.19	99.29	99.29	0.979	1.039
ES & Multi MS	2.871	3.019	2.72	99.24	99.24	1.046	1.107

Table 9. Legacy vs MS System characteristics comparison for Exclusive Containment.

Campaign Type	Scalability [CPU EC2]	Scalability [CPU DB]	Scalability [Network DB]	Availability [100]	Availability [200]	Efficiency [100]	Efficiency [200]
Original ES	3.458	3.366	3.586	99.27	99.31	1.000	1.000
ES & Single MS	2.45	2.195	1.919	99.25	99.24	1.03	1.107
ES & Multi MS	2.511	3.103	1.972	99.27	99.26	1.007	1.086

Table 10. Legacy vs MS System characteristics comparison for Inclusive Containment.

Campaign Type	Scalability [CPU EC2]	Scalability [CPU DB]	Scalability [Network DB]	Availability [100]	Availability [200]	Efficiency [100]	Efficiency [200]
Original ES	3.458	3.366	3.586	99.27	99.31	1.000	1.000
ES & Single MS	2.85	2.847	2.394	99.24	99.24	1.045	1.103
ES & Multi MS	2.17	2.655	1.729	99.24	99.24	1.037	1.107

Table 11. Legacy vs MS System characteristics comparison for Association.

Campaign Type	Scalability [CPU EC2]	Scalability [CPU DB]	Scalability [Network DB]	Availability [100]	Availability [200]	Efficiency [100]	Efficiency [200]
Original ES	3.458	3.366	3.586	99.27	99.31	1.000	1.000
ES & Single MS	2.56	3.503	2.108	99.28	99.34	1	0.964
ES & Multi MS	2.381	3.157	1.819	99.28	99.26	0.993	1.078

‘Thirdparty’ BOs and association of ‘Product’ and ‘User’, for which, firstly, a single microservice was created to manage both BOs and, secondly, separate microservices were created to manage each BO. The results obtained for these experiments are summarised in Tables 4, 6 and 7. Each microservice was hosted on an AWS elastic container service (ECS), which has two virtual CPUs and a total memory of 1GB, as depicted on the right side of Fig. 4. The data related to the BOs of each microservice was stored in one MySQL relational database instance with one virtual CPU and total storage of 20GB. Next, the executions were performed on ES again. Since microservices are extended parts of the enterprise system, in these executions, the enterprise system used API calls to pass the data to the microservices and the microservices processed and sent back the data to the enterprise system. The data in the microservice databases and enterprise system database was synchronized using the Amazon database migration service. Again, we recorded the total execution time, average CPU consumption, and average network bandwidth consumption for the entire system (i.e., enterprise system integrated with microservices). Since exclusive containment suggests co-locating contained BOs in a single MS, we evaluated the effectiveness of this using two experiments. First, we implemented one microservice to manage 'Order' and 'ECM' BOs with a subset of their operations in a single container and obtained the execution results. Then, we created a separate microservices to manage 'Order' and 'ECM' and conducted the experiment again. The results obtained for this experiment are summarised in Table 5.

Based on the attained values, we calculated the scalability, availability, and execution efficiency of the different combinations, the results are summarized in Tables 8–11. Scalability was calculated according to the resources and their usage over time, as described by Tsai *et al.* [16]. In order to determine availability, first we calculated the time taken to process 100 packets when a particular microservice is not available. Then, we measured the difference between the total uptime and total downtime, as described by Bauer *et al.* [17]. Efficiency gain was calculated by dividing the total time taken by the legacy system to process all requests by the total time taken by the corresponding microservice system.

Experimental Results: As described by Tsai *et al.* [16], the lower the number the better the scalability. Thus, it is evident from Tables 8, 10 and 11 that developing separate microservices to manage BOs which have subtype, association and inclusive containment relationships achieve better scalability than creating single microservices to manage such BOs. When comparing availability, there is not much gain. However, better execution efficiency for subtypes, association and inclusively contained microservices were achieved when they are developed as separate microservices. When comparing scalability, availability and execution efficiency for exclusively contained BOs, it is evident that they achieved better results when the microservices were co-located in the same container (see results for single microservice and multiple microservices in Table 9). These results affirmed that the patterns that our work suggested for subtyping, association, inclusive containment and exclusive containment relationships work well when developing microservices.

The lower the lack of cohesion and structural coupling numbers the better the cohesion and coupling of the system [18]. As such, it is evident from Tables 1 and 2 that the microservices derived from the Dolibarr system achieved better cohesion and coupling values than the legacy system. Hence, the obtained results have affirmed that the microservices extracted based on the patterns described in Section 3 led to microservices which could provide the same services to users while preserving overall system behaviour and achieving higher scalability, availability, efficiency, and cohesion, and lower coupling.

6 Related Work

Architectural and design conceptions of microservices have emerged through practitioner reports [1] and the textbooks by Newman [2] and Fowler [3], and have contributed to ongoing developments of microservice architectures. To date, general properties which are widely accepted for microservices as distributed and fine-grained software components are high scalability, high availability, high integrity, strong cohesion, loose coupling, and eventual consistency [2]. Furthermore, microservices have been characterised through agile units of software development, which can be separately developed, deployed, and composed.

Concerning the essential structure of a microservice architecture, the domain-driven design principle (DDD) [4] has been popularized through the widely cited references by Newman [2] and Fowler [3]. According to DDD, each service-based software component encapsulates all operations of objects, which in SOA based enterprise systems corresponds to business objects (BOs). Accordingly, patterns have been proposed for microservices by Chris Richardson such as 'database per service' and 'saga' [19]. However, to date, there are no clear definitions that help to refine SOA components which manage distinct BOs of domains, containing encapsulated CRUD operations of the objects [5], into microservices which recommend decomposing BOs for support of fine-granular functionality. Our work presented a refinement of the design strategy for SOA components such that the system operations are split based on four fundamental object relationships, namely, subtyping, association, inclusive containment, and exclusive containment. Considering such BO relationships and system execution patterns, we have conducted research here and in previous work [12, 15] and demonstrated that BO relationships could be used in the microservice derivation process to achieve prominent outcomes, such as high scalability, availability, and processing efficiency.

7 Conclusion

This paper presented four microservice patterns, namely object association, exclusive containment, inclusive containment and subtyping for 'greenfield' (new) development of software while demonstrating the value of the patterns for 'brownfield' (evolving) developments by identifying prospective microservices using prototypes we developed for two enterprise systems, Dolibarr (Section 5) and SugarCRM [20]. The conducted experiments confirmed that our patterns provide good suggestions for better microservice development while obtaining desirable characteristics such as high cohesion, low coupling, high scalability, high availability, and processing efficiency. The patterns presented herein could be used for further extended and refined pattern development, for example by taking into account the operation subtypes (i.e., create, read, update and delete). Furthermore, considerations for software modularization based on class clustering and method clustering should be considered as future work for microservice derivation.

References

1. Adopting Microservices at Netflix: Lessons for Architectural Design, https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices. Last accessed 5 May 2019

2. Newman, S.: Building microservices: designing fine-grained systems. O'Reilly. (2015)
3. Microservices a definition of this new architectural term, https://martinfowler.com/articles/microservices.html. Last accessed 5 May 2019
4. Evans, E., 2004. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional.
5. Erl, T., 2008. SOA Design Patterns (paperback). Pearson Education.
6. Taibi, D., Lenarduzzi, V. and Pahl, C., 2018, March. Architectural Patterns for Microservices: A Systematic Mapping Study. In CLOSER (pp. 221-232).
7. Barros, A., Duddy, K., Lawley, M., Milosevic, Z., Raymond, K. and Wood, A., 2000, October. Processes, roles, and events: UML concepts for enterprise architecture. In International Conference on the Unified Modeling Language (pp. 62-77). Springer, Berlin, Heidelberg.
8. Gamma, E., 1995. Design patterns: elements of reusable object-oriented software. Pearson Education India.
9. van der Aalst, W.M., 2009. Workflow patterns. Encyclopedia of Database Systems, pp.3557-3558.
10. Barros, A., Dumas, M. and Ter Hofstede, A.H., 2005, September. Service interaction patterns. In International Conference on Business Process Management (pp. 302-318). Springer, Berlin, Heidelberg.
11. Decker, G., Barros, A., Kraft, F.M. and Lohmann, N., 2008, December. Non-desynchronizable service choreographies. In International Conference on Service-Oriented Computing (pp. 331-346). Springer, Berlin, Heidelberg.
12. De Alwis, A.A.C., Barros, A., Polyvyanyy, A. and Fidge, C.: Function-splitting heuristics for discovery of microservices in enterprise systems. In: International Conference on Service-Oriented Computing, pp. 37–53. Springer, Cham. (LNCS, volume 11236)
13. Nooijen, E.H.J, van Dongen, B. F. and Fahland, D.: Automatic discovery of data-centric and artifact-centric processes. In : In International Conference on Business Process Management, pp. 316–327. Springer (2012)
14. Andrews, R., Suriadi, S., Ouyang, C. and Poppe, E : Towards Event Log Querying for Data Quality. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", pp. 116-134. Springer, Cham. (LNCS, volume 11229)
15. De Alwis, A.A.C., Barros, A., Fidge, C. and Polyvyanyy, A., 2018, October. Discovering Microservices in Enterprise Systems Using a Business Object Containment Heuristic. In OTM Confederated International Conferences" On the Move to Meaningful Internet Systems" (pp. 60-79). Springer, Cham. (LNCS, volume 11230)
16. Tsai, W.T., Huang, Y. and Shao, Q.: Testing the scalability of SaaS applications. In: Service-Oriented Computing and Applications (SOCA), IEEE International Conference, pp. 1–4. (2011)
17. Bauer, E. and Adams, R.: Reliability and availability of cloud computing, 1st edn. John Wiley & Sons (2012)
18. Candela, I., Bavota, G., Russo, B. and Oliveto, R.: Using cohesion and coupling for software remodularization: Is it enough?. In: ACM Transactions on Software Engineering and Methodology (TOSEM), pp. 24. (2016)
19. A pattern language for microservices, https://microservices.io/patterns/. Last accessed 5 Aug 2019
20. Technical report for SugarCRM, https://drive.google.com/file/d/1u0Ai0XNOx-3yGoz0Keycxj093_NyDrJl/view?usp=sharing