UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DEPARTMENT OF INFORMATION ENGINEERING, COMPUTER SCIENCE AND MATHEMATICS

Doctoral Program in Information and Communication Technology

Curriculum Emerging computing models: algorithms, software architectures and intelligent systems

XXXV cycle

Automated composition of (micro)services and decomposition of monoliths

SSD: INF/01

Candidate

Gianluca Filippone

Doctoral Program Supervisor

Prof. Vittorio Cortellessa

Tutor

Prof. Massimo Tivoli

Co-tutor

Prof. Marco Autili

A.Y. 2021/2022

Abstract

Service Oriented Architecture (SOA) has been the leading paradigm for building large-scale systems through the reuse and composition of software services for decades. Choreographies are a powerful and flexible composition approach in which services are composed in a loosely coupled fashion and their interaction is performed without relying on a central entity. By amplifying the loose coupling, independence, and flexibility principles of SOA, Microservice Architecture (MSA) has gained growing interest from companies and researchers. MSA consists of a set of small, independent, and loosely coupled services that communicate through lightweight protocols. Microservices are easy to scale, maintain, and deploy. Attracted by the advantages of MSA, companies are migrating their legacy monolithic systems to microservices.

In this setting, a series of problems arise. First, in choreography-based systems, the realization of the distributed coordination logic required to enforce the correct choreography realization requires automated support. The need for building dynamic and user-centered systems also calls for the realization of choreographies capable to adjust their behavior to the surrounding context and changing user preferences. Second, migrating monolithic systems to microservices is a complex, time-consuming, and error-prone task that needs the support of appropriate tools to assist software designers and programmers, from extracting a proper architecture to implementing novel microservices. Third, in choreographed microservice-based systems, proper support for the scalability of microservices needs to be considered alongside the need for coordination required by choreographies. The two functionalities must coexist while remaining functionally and architecturally independent.

The work presented in this thesis aims at addressing the problems above by proposing (i) a solution for the realization and synthesis of context-aware choreographies, (ii) a fully-automated approach for the migration of monolithic systems into MSA, and (iii) an architectural style for microservice-oriented choreographies decoupling the coordination and load balancing capabilities of the system.

Keywords: Service composition, system decomposition, choreographies, coordination, context-awareness, microservices, scalability, load balancing

Contents

List of Figures **vi**

List of Tables **viii**

Acronyms **x**

1 Introduction **1**

1.1 Problem space 2
1.2 Solution space 4
1.3 Working context 5
1.4 Contribution 6
1.5 Outline of the Thesis 6

2 Background **8**

2.1 Service composition approaches 8
2.1.1 Orchestrations 8
2.1.2 Choreographies 9
2.1.3 Orchestrations vs. Choreographies 10
2.1.4 BPMN2.0 Choreography Diagram 11
2.2 Automated choreography synthesis 14
2.2.1 Choreography realizability enforcement 17
2.3 Microservice Architecture 20
2.3.1 Architectural principles 21
2.3.2 Benefits 22
2.3.3 Drawbacks 23
2.3.4 SOA vs. Microservices 24
2.4 Load balancing approaches 25
2.4.1 Server-side load balancing 26
2.4.2 Client-side load balancing 27
2.4.3 A hybrid approach 28

3 Synthesis of context-aware choreographies **30**

3.1 Dealing with context-awareness 31
3.2 Expressing variability 33
3.3 Reference scenario 35

3.3.1 Arising complexities 38
3.4 Approach description 39
3.4.1 Context model 41
3.4.2 Context-aware CDs 45
3.4.3 Context Manager 48
3.5 Synthesis and refinement processes 52
3.6 Evaluation 56
3.6.1 Experimentation settings 57
3.6.2 Data collection and metrics 58
3.6.3 Experiment results 61
3.7 Discussion 66
3.7.1 Threats to validity 67

4 Monoliths decomposition 70
4.1 Approach description 71
4.1.1 Analysis 72
4.1.2 Decomposition 74
4.1.3 Optimization 77
4.1.4 Refactoring 80
4.2 Evaluation 83
4.2.1 Evaluation metrics 84
4.2.2 Performed experiments 86
4.2.3 Experimental results 87
4.3 Discussion 92
4.3.1 Threats to validity 93
4.3.2 On the applicability of the approach 94

5 Architectural style for scalable choreography-based systems 96
5.1 Case study 97
5.1.1 Arising complexities 99
5.2 Architectural style 100
5.2.1 Architectural layers 100
5.2.2 Architectural style at work 102
5.3 Evaluation 104
5.3.1 Evaluation of design alternatives 105
5.3.2 Experimentation 108
5.4 Discussion 111
5.4.1 Threats to validity 112

6 Related work 114
6.1 Choreography realizability and enforcement 114
6.2 Context-aware systems 115
6.3 Decomposition into microservices 118
6.4 Microservices composition approaches 120
6.5 Architectures for microservices load balancing 121

7 Conclusions and future work **124**

List of Publications **126**

References **127**

List of Figures

2.1 Orchestration 9
2.2 Choreography 10
2.3 Orchestration vs. Choreography 11
2.4 Sample BPMN2 Choreography 12
2.5 Sample choreography with sub-choreography activity 13
2.6 Task with multi-instance participant 13
2.7 Sample architectural description of a choreography with coordination delegates 14
2.8 Coordination delegates interaction pattern 15
2.9 Projection of the sample choreography over Participant A 16
2.10 Coordination of independent sequence of tasks 18
2.11 Independent sequences across independent branches 19
2.12 Synch node metamodel 20
2.13 Centralized server-side load balancer 26
2.14 Decentralized server-side load balancer 27
2.15 Client-side load balancer 28
2.16 Client-side load balancer 29

3.1 BPMN2 metamodel extension for variation points 34
3.2 Sample choreography with variation point 35
3.3 System connections 36
3.4 Reference scenario: Public billposting choreography 37
3.5 Choreoraphy variants for Payment variation point 37
3.6 Approach overview 40
3.7 Context metamodel 42
3.8 Context model for the reference scenario 43
3.9 XML schema of the context-carrying message "*postRequest*" . . . 44
3.10 Context-aware CD interaction pattern for context-carrying messages 46
3.11 Context-aware CD interaction pattern for context-aware participant instantiation 47
3.12 Context-aware CD interaction pattern for variant selection 48
3.13 Class diagram of the context representation in the Context Manager 49
3.14 Context Manager behavior for the context-aware participant instantiaton 50
3.15 Synthesis process overview 52
3.16 Generated system architecture 54

3.17 Refinement process overview 55
3.18 Selection functions implementation 56
3.19 Geographical distribution of the municipalities involved in the choreography 57
3.20 Sequence diagram illustrating timing of the simulated *Mobile App* interactions 58
3.21 Services deployment scheme for the experimentation 59
3.22 Timestamps logged by participant services 60
3.23 Average execution time of choreography tasks 62
3.24 Details of the independent sequence execution 63
3.25 Trend of task execution time, adaptation overhead, and coordination overhead at the increasing number of users 65
3.26 Trend of adaptation overhead and entities selection time at the increasing number of service instances 66

4.1 Overview of the decomposition approach 71
4.2 Information graph obtained for the Spring Petclinic project 73
4.3 Overview of the decomposition phase 76
4.4 Communities obtained from the complete information graph . . . 76
4.5 Communities obtained from the sub-graph of persisted entities . . 77
4.6 Result of the optimization 81
4.7 Example of API controller synthesis 82
4.8 Example of API consumer syntesis 82
4.9 Example of entities duplication 83

5.1 Online ticketing system choreography 98
5.2 Architectural style 101
5.3 Online ticketing system architecture 103
5.4 Deployment setting for the scenario without load balancing 110
5.5 Deployment setting for the scenario with load balancing layers (CDs are equipped with local client-side load balancer) 111
5.6 Experimentation results 112

List of Tables

3.1 Experiment results (data in ms) 64

4.1 Application used in the reported experiments 86
4.2 Result comparison for the decomposition of JPetstore 88
4.3 Result comparison for the decomposition of Spring Petclinic . . . 90
4.4 Comparison of the microservices identified for Cargo Tracking system 91

5.1 Design alternatives evaluation 105

Acronyms

B2B Business-to-Business

caCD Context-aware Coordination Delegate

CD Coordination Delegate

ESB Enterprise Service Bus

MSA Microservice Architecture

PA Public Administration

SCA Static Code Analysis

SLA Service Level Agreement

SOA Service Oriented Architecture

Chapter 1

Introduction

For decades, Service Oriented Architecture (SOA) has imposed as a software design and architectural pattern allowing to build large-scale systems by reusing, composing, and integrating existing services. In SOA, the functionalities of a system are organized as a collection of independent services which are composed together and communicate with each other over the network through a standardized protocol, regardlessy of the technology underlying each service. Those systems show enhanced flexibility since services can be added, removed, or replaced allowing the system to evolve over time without disrupting the overall system [27, 44, 96].

Building upon the principles of SOA, choreographies represent a powerful and flexible service composition approach in which participant services are loosely coupled and their interaction is performed without relying on any central entity [6, 9, 18, 37]. Services communicate in a peer-to-peer style (without the asymmetry found in, e.g., client-server style or orchestration-based approaches), autonomously take decisions, and, out of the blue, engage in the interaction by performing tasks according to their imminent needs and local state. This setting fosters the interoperation among different services provided by different companies and/or institutions in realizing larger value-added systems, in which none of the involved parties is required to take full responsibility for controlling and coordinating the global interaction.

As a further advancement of SOA, in recent years Microservice Architectures (MSAs) gained growing interest from both research and industry for developing and deploying modern distributed applications. MSAs are built through the composition of fine-grained, loosely coupled, and autonomous services that are organized around specific business capabilities, running in autonomous processes and communicating through lightweight protocols [42, 79, 93]. Microservices are

advantageous in terms of scalability, flexibility, and resilience, allowing for more efficient use of resources and infrastructure and better support for continuous integration and delivery [43, 93, 107]. In order to leverage these benefits, companies are adopting microservices architecture to realize their systems, and they are migrating their legacy monolithic architectures towards microservices [114].

1.1 Problem space

When dealing with choreography-based systems that are built by reusing and composing services as third-party or black-box entities, the realization of the distributed coordination logic required to achieve the correct interaction is a complex and error-prone activity [6, 9]. In fact, services involved in the choreography act as active entities that concurrently perform tasks and autonomously take decisions. They may not synchronize as prescribed by the choreography, and the global collaboration may not follow the specification. Composing such services and realizing the required *distributed coordination* calls for suitable automated support that aids the development activities by providing correct-by-construction and ready-to-use solutions to concurrency and realizability issues.

Besides showing the desired behavior emerging from the interaction of the involved parties (i.e., composed services), systems are also required to offer *context-aware* functionalities to allow users to access dynamic and customized functionalities. Systems can leverage their knowledge about the user and/or environment to adapt their behavior according to the surrounding context. For instance, *location-awareness* represents a crucial context dimension that allows users to access the right services, at the right moment, in their current location. For this reason, the tool support for the composition of services through choreographies is also required to provide support for the realization of systems with context-aware capabilities.

"Legacy" SOA systems, and – as a consequence – service choreographies representing their realization, are typically composed of a limited number of large and complex monolithic applications [107]. During their lifecycle, they may become difficult to maintain, evolve, and scale [42, 93]. As said, companies may benefit from the advantages of MSA by migrating their systems (or services – even if only some of those that compose a SOA system) into microservices. However, the migration of monolithic systems to microservices is a difficult, complex, and time-consuming task requiring much effort and specific skills [114]. If performed manually, the quality of the decomposed system depends on the experi-

ence and knowledge of experts [68]. Hence, it requires tools and techniques that can ease and drive the decomposition process. During the last years, several researchers have been proposing migration solutions through tools or frameworks, e.g., [33, 49, 59, 61, 64, 70, 82, 117, 128]. Most of the proposed approaches are challenged by the difficult task of establishing the proper granularity of microservices or supporting the so-called "single responsibility principle", i.e., a microservice must conform to and stay within a bounded context [43], which may contain one or more aggregates [93]. More importantly, many of the proposed approaches require different types of inputs such as use cases, domain models, activity diagrams, business models [49], or even manual inputs [82, 117] provided by system engineers having specific knowledge of the system to be decomposed. The lack of models or specific system knowledge may limit the applicability of these refactoring approaches. Moreover, once the new system architecture is extracted, developers must define and implement the APIs that allow communication between microservices. Even if the new architectural style enables DevOps and continuous delivery, the definition and implementation of interfaces remain a time-consuming activity that needs tool support. Given the above, the migration of monolithic systems into microservices requires tools that can assist developers and engineers during all the phases of the migration process (i.e., system analysis, decomposition, microservice implementation), automatizing the whole process without requiring models or users' system knowledge.

As for "classical" service-oriented systems, also microservice-based systems can be conveniently realized through decentralized composition approaches, like choreographies, that permit furtherly enhance loose coupling, independence, and flexibility. Such systems can benefit from the small size and isolation of microservices to achieve enhanced scalability. In fact, microservices enable the replication of those that are more exposed to growing loads in terms of the number of requests and may represent a bottleneck for system performances. In this way, the workload of each microservice can be distributed among a set of instances concurrently running in different servers/hosts/containers [43] through *load balancers* that actively route each incoming request toward one of the running instances of the target microservice. In choreographed microservice-based systems, both load balancing and coordination issues have to be taken into account. Here, load-balancing capabilities must coexist with the distributed coordination needs of the system. However, they should remain both functionally and architecturally independent in order to cope with the loose-coupling, autonomy, and flexibility principles of SOA and, being its evolution, MSA. This concern may arise both in

those scenarios where microservices are obtained after having decomposed coarse-grained services from a "legacy" choreography-based system, and both in systems that are built from scratch by composing new microservices as a choreography. In literature, several approaches deal with scalability [14, 39, 50, 121, 124], but none of them specifically address the scalability of choreography-based service-oriented systems together with, yet by fully decoupling, coordination issues.

1.2 Solution space

In previous works [6, 9], choreography realization is enforced through the automated synthesis of additional software entities, called *Coordination Delegates (CDs)* that, when interposed between the participant services, control their interactions by running a distributed coordination algorithm and exchanging synchronization messages. This thesis extends this approach by accounting for and enabling the development of context-aware choreographies, allowing the specification and the execution of the adaptable behavior of the choreography according to the context. An enhanced version of CDs, called *Context-aware Coordination Delegates (caCDs)*, is introduced to dynamically adapt the choreography behavior according to the context conditions. A *Context Manager* is introduced as a new software entity that provides, at runtime, all the functionalities needed for context sensing and evaluation. An ad-hoc built context metamodel allows developers to describe all the context characteristics that have to be taken into account for adaptation.

Concerning the challenges in the migration of monolithic systems into microservices, this thesis presents a bottom-up decomposition approach that fully automates the decomposition process and identifies microservices having functionalities combined together for each application's bounded context, without requiring manual inputs, system models, specific skills, or knowledge. The approach first performs Static Code Analysis (SCA) of the monolithic system to produce a graph representation of the system with a method-level resolution, in which arcs are weighted in order to represent the relevance of each of the relationships between nodes. Then, graph nodes are clustered in communities to obtain the granularity and the functionality scope of the services, while a combinatorial optimization problem is solved to obtain the final set of microservices having the minimum coupling. Given the resulting architecture, a synthesis algorithm automatically generates the code of the microservices by suitably moving methods and classes of the monolith into the correct microservice, further identifying the

novel APIs that have to be exposed and implementing their controllers.

In order to deal with both scalability and coordination in microservice-based systems, this thesis proposes a layered architectural style that allows realizing scalable microservice-oriented choreographies. The architectural style is composed of two main layers: a coordination and a load-balancing layer. The coordination layer is needed to ensure that microservices behave as prescribed by the choreography specification and avoid undesired interactions. It leverages the notions of CDs as mentioned above to achieve coordination capabilities. The load-balancing layer distributes the workload by properly routing the incoming requests among the available microservice instances. This allows optimizing the resource usage, avoids a single microservice instance being overloaded by incoming requests, avoids bottlenecks, and hence maximizes the system performances. The layered nature of the architectural style implies that the interactions are balanced after being coordinated, hence avoiding the routing of undesired interactions to microservices instances. The clear separation of the two layers allows them to be agnostic to each other, hence totally decoupling their functionalities.

1.3 Working context

With the objective of bringing the adoption of choreographies to the development practices adopted by IT companies, during the last decade, the activities of our research group focused on developing automatic approaches to support the realization of service choreographies. Moreover, by following the industrial and research evolution of SOA towards the adoption of microservices, the activities of the research group moved to the study and realization of automated tools in support of the migration of monolithic systems into microservices. In this direction, this thesis has been supported by the EU CHOReVOLUTION[1], INCIPICT[2], ConnectPA, and the SISMA projects.

The HORIZON 2020 *CHOReVOLUTION* project proposes a development process for the automatic realization of choreography-based systems. It provides the approach for coordinating service choreographies that is extended in this work to support the realization of context-aware choreographies.

The *INCIPICT* (*Innovating City Planning through Information & Communication Technologies*) project aims at realizing an experimental optical network to build a Metropolitan Area Network (MAN) for the city of L'Aquila, located in

[1]https://cordis.europa.eu/project/id/644178
[2]https://incipict.univaq.it

the region of Abruzzo in central Italy. This network provides the essential substrate for the delivery of different services, enabling the realization of dynamic large-scale applications for L'Aquila city.

The POR FESR 2014-2020 *ConnectPA* Italian project collects the needs and innovation requests of the Italian Public Administration (PA). The project goal is to develop a choreography-based approach for the provisioning of IT solutions for public administrations by implementing a platform capable of interconnecting, composing, and coordinating geographically distributed services in order to create dynamic and interoperable e-Government services for smart cities.

The MUR PRIN *SISMA* (*Solutions for Engineering Microservice Architectures*) project contributes to the enhancement of the microservices architectural style. It addresses their architectural design and the problem of migrating existing applications towards microservice architectures, the continuous deployment, and the runtime management of microservices and the resources needed for their efficient execution.

1.4 Contribution

The contribution of this thesis is advancing the state-of-the-art by:

- Proposing a novel solution to the realization of context-aware choreographies through automated synthesis;
- Proposing an automated approach to the decomposition of monolithic systems into microservices;
- Presenting a layered architecture for the composition of microservices that considers the system's coordination and load balancing needs decoupling their roles.

1.5 Outline of the Thesis

This thesis is structured as follows:

Chapter 2 presents the background notions of the work, concerning (i) the automated synthesis and enforcement of service choreographies, (ii) microservice architectures, and (iii) load-balancing approaches;

Chapter 3 presents the approach for the realization of context-aware choreographies. It leverages a reference scenario related to the ConnectPA project as a working example and for the evaluation of the approach;

Chapter 4 presents the approach for the automated decomposition of monoliths into microservices. The approach is evaluated among some well-known publicly available monolithic systems and results are compared with other related *sota* approaches;

Chapter 5 presents the layered architecture for the coordination and load-balancing of microservice-based systems. A case study is presented and the architectural style is shown at work on it. The layered architecture is evaluated against its properties and the impact of the load-balancing capabilities on the system is assessed through the case study;

Chapter 6 discusses the related works of the presented approaches;

Chapter 7 discusses the conclusions of the thesis and the future research directions.

Chapter 2

Background

This chapter provides background notions on the approaches for the service composition (orchestrations and choreographies) highlighting their peculiarities and relationships. It also presents the automated approaches for the automated synthesis of service choreographies which is leveraged and extended for the realization of context-aware choreographies. Then, it provides basic notions about MSA as well as approaches for load balancing. This information serves as a foundation for the subsequent chapters of the thesis.

2.1 Service composition approaches

2.1.1 Orchestrations

Orchestrations are a centralized approach for the composition of software services into a larger application in which the interactions are controlled by a single coordinator service. Here, the control is always represented and realized from one party's perspective.

Figure 2.1 sketches the idea underlying this approach. All the services involved in an orchestration are *provider* services (i.e., they only offer functionalities through interfaces), and a centralized entity (the *orchestrator*), interacts and coordinates these services being the only *consumer* of the functionalities provided by the services. When invoked by the orchestrator, provider services play the "passive" role of executing black-box operations and providing the related responses. Both the application and coordination logic are all centralized in the orchestrator, and all composition/coordination issues can simply be solved from within the orchestrator.

Orchestrations are good in those situations in which a single party needs to

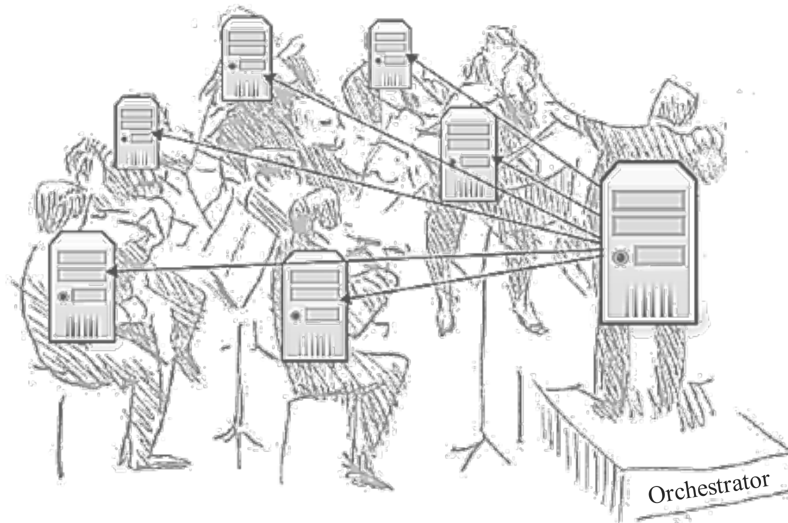


Figure 2.1: Orchestration

take the responsibility to centrally coordinate the interaction among different services; they are not suitable to cross-enterprise employments where multiple parties must collaborate, but none of them want to, or cannot, take the full responsibility for performing centralized coordination. This is where service choreographies step in.

2.1.2 Choreographies

Choreographies are a decentralized composition approach in which participant services communicate as peers without any central coordinator. The interactions of services are described from a global perspective since the common goal of the application is realized by the globally controlled collaboration of services, which can autonomously perform tasks according to the global state of the choreography [101]. Figure 2.2 sketches the idea underlying the notion of service choreography, which can be summarised as follows: *dancers dance following a global scenario without a single point of control.*

Service choreographies differ significantly from service orchestrations, where one stakeholder (i.e., the orchestrator) centrally determines how to reach the goal through cooperation with other services. Choreography does not rely on a central coordinator since each involved service knows exactly when to execute its operations and with whom to interact. Choreographies are a collaborative effort focusing on the exchange of messages among several participants to reach a common global goal.

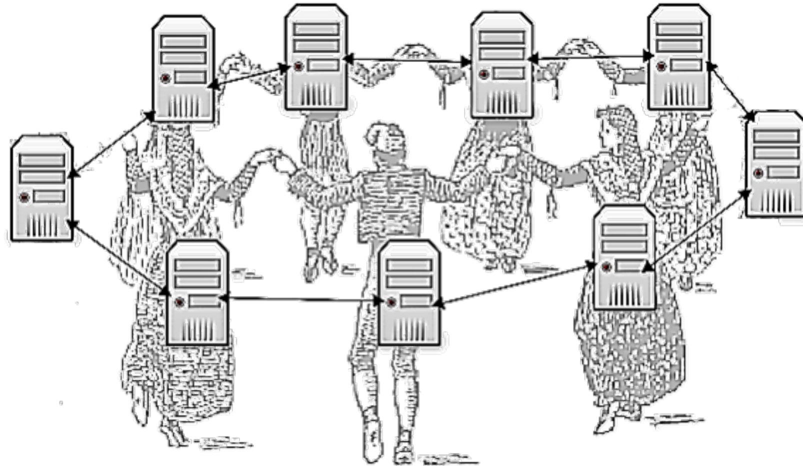


Figure 2.2: Choreography

2.1.3 Orchestrations vs. Choreographies

Both orchestrations and choreographies are used to create business processes by composing services. Although these approaches differ significantly from the coordination point of view, i.e., centralized vs. decentralized, they overlap somewhat [101]. Figure 2.3 illustrates their relationships and how they can coexist in a large cross-enterprise application. Orchestrations refer to specific executable business processes from one party's perspective. Such business processes can interact with both internal and external services. This is rather different in choreography, which is highly collaborative and concerns the message exchanges among multiple parties (i.e., external message exchanges that occur between services) rather than a specific business process that a single party executes. As already said, choreographies are particularly useful in those situations in which multiple parties have to collaborate, but none of them wants to take the responsibility of running a centralized orchestration, e.g., in Business-to-Business (B2B) transactions. In B2B applications, which are by definition cross-enterprise, it is difficult to specify the implementation of specific participants, and there is no central authority for the overall flow. Conversely, inside organizations, orchestrations may be advantageous due to their simpler specification and the simpler identification of a central authority.

Although conceptually different, at the implementation level, the two approaches may overlap to some extent. A choreography might be realized as a set

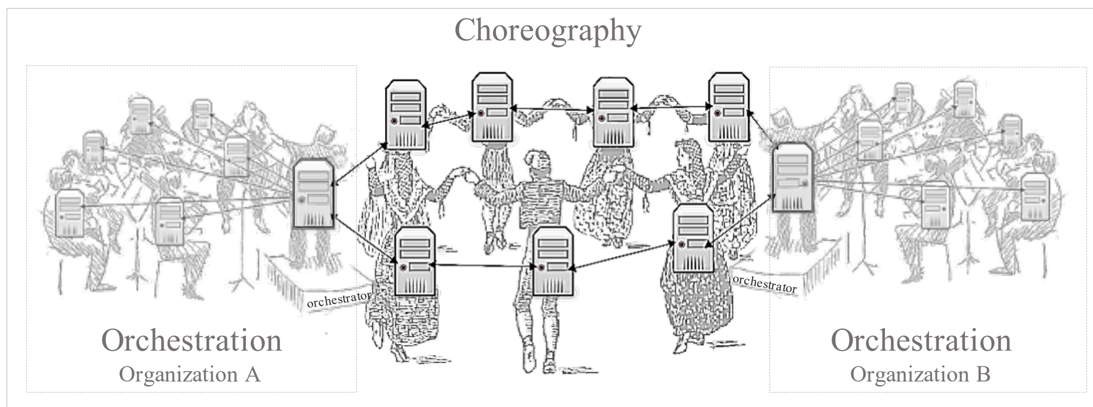


Figure 2.3: Orchestration vs. Choreography

of distributed software entities implemented as BPEL or BPMN processes. However, the technology used to realize them does not matter. Rather, the real difference resides in their nature, purpose, and how they are specified. Orchestration-oriented approaches consider a specification that is centralized and represents a form of centralized coordination whose goal is to (re-)expose the resulting composed system in a way that is amenable to further hierarchical composition. Instead, choreography-oriented approaches account for a specification that represents a form of distributed coordination of different participants collaborating to achieve a global common goal. Thus, the purpose is to exploit this specification to enable the prescribed collaboration in a fully-distributed way and let it emerge while the participants interact. These two specifications can be exploited together, as shown in Figure 2.3, by reusing orchestration-based composed systems as participants of a wider choreography-based system.

2.1.4 BPMN2.0 Choreography Diagram

The OMG's BPMN 2 standard[1] offers *Choreography Diagrams*, a practical notation for specifying choreographies that, following the pioneering BPMN process and collaboration diagrams, is amenable to be automatically treated and transformed into actual code. BPMN2 choreography diagrams focus on specifying the message exchanges among the participants from a global point of view. A participant role models the expected behavior (i.e., the expected interaction protocol) that a service should be able to perform to play the considered role.

In BPMN2, choreography is modeled as a set of tasks that represent the interactions among the participant services. A task is an atomic activity that

[1] https://www.omg.org/spec/BPMN/2.0.2/

describes the message exchanges (request and optionally response) between two participants. Each task is performed by an initiating participant, which sends a message to the receiving participant and, optionally, receives a response message when the task is complete.

Figure 2.4 shows a sample choreography in BPMN2 notation. Graphically, a task is represented by a rounded-corner box, and it is labeled with the role names of the participant services and the name of the task. The initiating participant role name is contained in the upper white box; the receiving participant role name is contained in the gray box at the bottom. Request and response message names are attached to the initiating and receiving participant boxes, respectively. By referring to the figure, for the task `T1`, the initiating participant *Participant A* sends the message `RequestMessage` to *Participant B*, which replies with the message `ResponseMessage`. Tasks are connected by an arrow, which represents a sequence flow. The BPMN2 specification states that the initiating participant of a choreography task must have been involved (as initiating or receiving participant) in the previous choreography task.

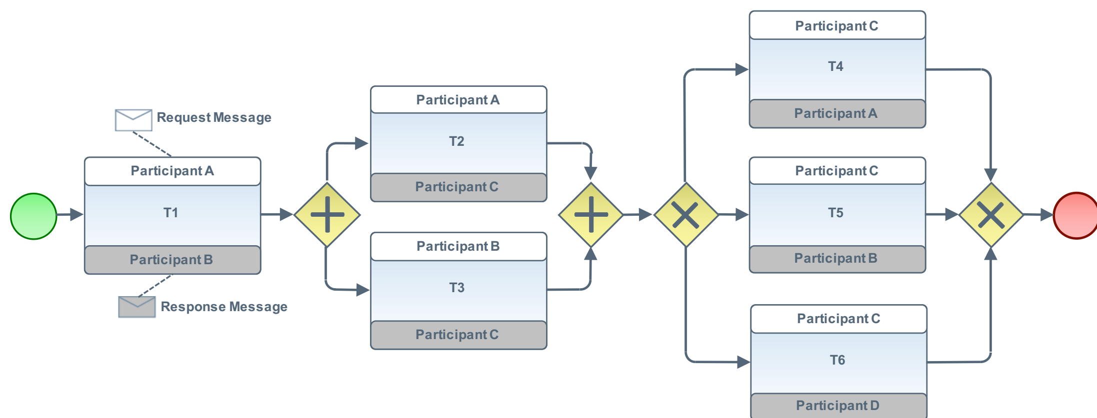


Figure 2.4: Sample BPMN2 Choreography

The sequence flow can be controlled through *gateways*, which allow splitting a flow into two or more parallel or alternative flows. *Parallel gateways* are represented graphically by rhombuses marked with the "+" symbol. They are used to fork a flow in two or more parallel flows (diverging parallel gateway) and/or synchronize and join two or more parallel flows (converging parallel gateway). In the diverging case, the gateway creates parallel paths of the choreography that are executed concurrently and that all the involved participants are aware of. In the converging case, this gateway waits for all incoming flows to be completed before triggering the flow through its outgoing arrow. Concerning the constraints imposed by the BPMN2 standard specification, the initiating participant(s) of all

the tasks immediately after the gateway must be involved in all the tasks that immediately precede such gateway.

Exclusive gateways are represented by rhombuses marked with the "×" symbol. They are used to create alternative paths (diverging exclusive gateway) according to the result of the evaluation of a conditional expression and/or merge alternative paths (converging exclusive gateway). According to the BPMN2 official specification, the initiating participants of the choreography tasks immediately following the gateway must have sent or received the message that provided the data upon which the conditional decision is made. Furthermore, like for the parallel gateway, the initiating participants of the tasks following the gateway must be involved in the task(s) immediately preceding the gateway.

Portions of choreography can be modularly specified through compound activities, named *sub-choreographies*, which define in their detail a flow of other tasks. As shown in Figure 2.5, a sub-choreography is represented by a rounded-corner box having a square with a "+" symbol inside. The labels of the participants that initiate tasks in the flow are contained in white boxes, while the other participant into gray boxes. In the expanded form, sub-choreographies explicitly show the tasks in the body of the box instead of its name.

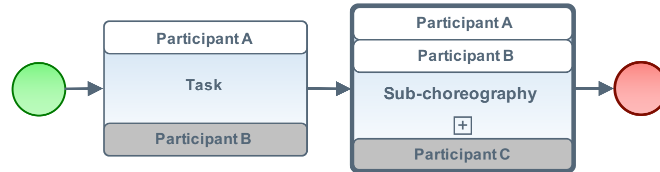


Figure 2.5: Sample choreography with sub-choreography activity

There are cases when there may be more than one possible service to be involved in a choreography task. For example, there may be more than one shipping service to be invoked according to, e.g., the selected destination. This situation can be modeled in BPMN2 through a *multi-insance participant*. Graphically, as shown in Figure 2.6, a multi-instance participant is represented by putting a multi-instance marker (three vertical lines) in the participant band for that participant.

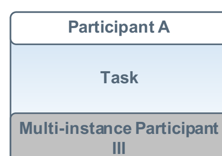


Figure 2.6: Task with multi-instance participant

BPMN2 Choreography diagrams also allow expressing looping tasks, events, and several gateways besides the ones already discussed. However, we avoid presenting them in this thesis for the sake of simplicity, since their knowledge is not required to fully understand the work.

2.2 Automated choreography synthesis

The approach for the realization and coordination of service choreographies considered in this work, developed in the scope of the CHOReVOLUTION project, leverages a synthesis tool that automatically generates the distributed coordination logic of the choreography [5, 6, 9, 10]. The tool provides a *synthesis processor*, which takes as input the BPMN2 choreography diagram and the set of services selected as participants to be composed into the choreography. As an output of the *synthesis process*, a set of additional software entities called *Coordination Delegates (CDs)* are automatically generated and properly interposed among the participant services. They proxify the service interaction in order to realize the specified choreography by running a distributed coordination algorithm [6]. CDs allow developers to decouple the coordination logic – which is external to the involved services since it resides in the CDs implementation – from the business logic that resides in the implementation of the services. In this way, CDs are able to perform external coordination while the services are completely agnostic to possible coordination and concurrency issues.

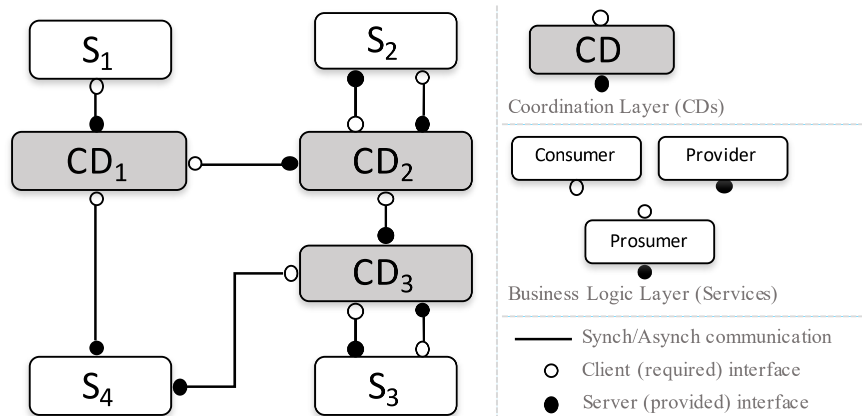


Figure 2.7: Sample architectural description of a choreography with coordination delegates

Figure 2.7 shows, through an informal notation, the architecture instance of a system that realizes a sample choreography. The involved services (S_1, S_2, S_3 and

S_4 in Figure 2.7) realize the functionalities and the business logic of the system by interacting with each other and exchanging business messages. Their communication is proxified by CDs (CD_1, CD_2 and CD_3 in Figure 2.7) which coordinate the interaction between services, ensuring that (i) the messages are exchanged according to the control flow prescribed by the choreography specification and that (ii) no undesired interactions are performed (e.g., for a given service, a certain message has to be received before that another message is sent). In general, for each service that requires an interface from another service (i.e., a *consumer* or *prosumer* service), a CD is deployed. Each CD coordinates many choreography executions through a correlation mechanism that is realized by adding a *choreography ID* in the messages sent by the consumer and prosumer services. The choreography ID is unique for each execution of the choreography and it is shared among all the CDs, consumer, and prosumer services. It is required to let CDs check the current state of a given choreography execution and enforce the correct sequence of interactions.

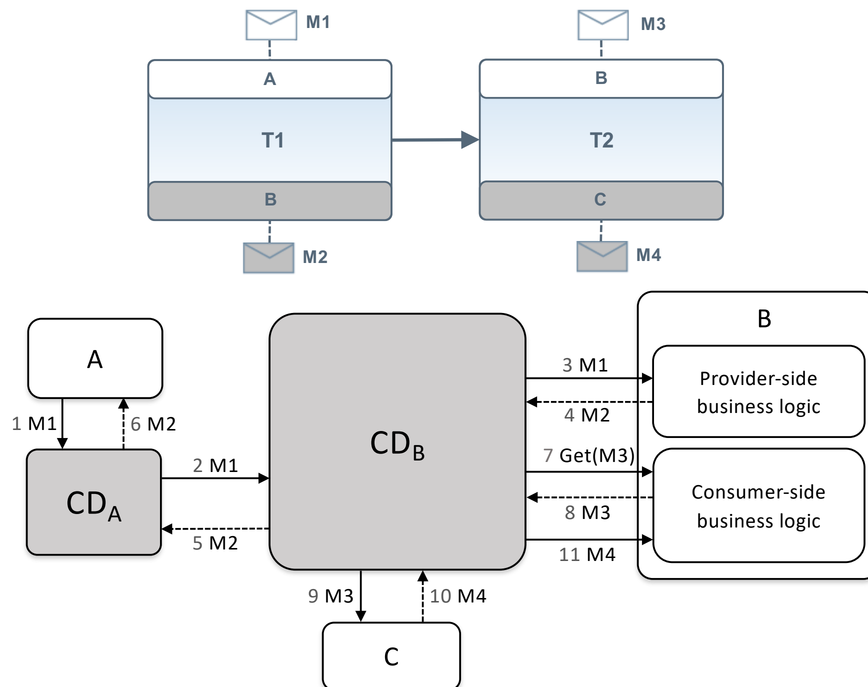


Figure 2.8: Coordination delegates interaction pattern

Figure 2.8 shows the interaction pattern of CDs in the most general case of a sequence of two tasks involving a consumer (A), a provider (C), and a prosumer service (B). The prosumer logic is split into *provider-side* and *consumer-side* business logic. *Provider-side* business logic offers the business functionalities of the service when acting as a provider (i.e., when it is a receiving participant in a

task), and needs to be implemented from scratch if no other existing services are reused. *Consumer-side* business logic offers the *message retrieval logic* and *message construction logic*: the former allows the CD to intercept and store messages exchanged by the service; the latter is the logic needed for building messages sent by the service as a consumer (i.e., when it is the initiating participant in a task). As it is shown in the figure, the CD generated for the consumer A (CD_A) receives the message sent by the service A (`M1`) as first, hence forwarding it to the CD generated for the prosumer B, which in turn forwards it to the provider side of B (interactions 1 to 3). Then, the related response messages are sent back, hence ending the execution of task `T1` (interactions 4 to 6). In order to execute `T2`, CD_B asks the consumer-side of B for the message to be sent (`M3` in the interactions 7 and 8), forwards it to C, intercepts the response and sends it back (interactions 9 to 11).

The automated synthesis of CDs is performed through a model-to-code transformation that takes as input the participant models of the choreography. A participant model describes the interaction protocol of a participant (i.e., a service): it represents the expected behavior of a participant, as the sequence of actions that it performs when interacting with other services and/or the environment. The participant model is a BPMN2 diagram, which is generated through a model-to-model transformation (*Choreography Projection*) carried out from BPMN2 the choreography specification [5, 9]. It is a partial view of the choreography which considers only the flows and tasks in which a participant is involved.

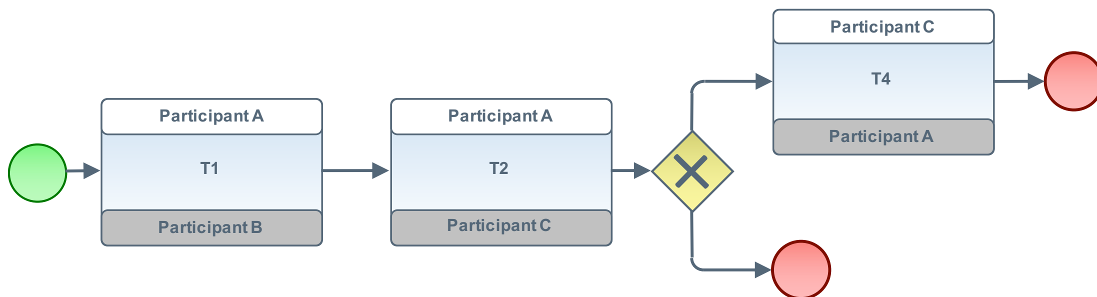


Figure 2.9: Projection of the sample choreography over Participant A

Fig 2.9 shows the result of the choreography projection carried out from the sample choreography in Figure 2.2 over *Participant A*. As said, it includes only the tasks in which *Participant A* is involved and the flows connecting them. From the set of participant models, the synthesis algorithm executed by the synthesis processor is able to generate in a fully-automatic way the set of CDs

needed for the choreography coordination. Referring to prosumer services, whose structure is well defined according to the interaction pattern described above, the synthesis processor generates their skeleton code, since both the provider-side and the consumer-side business logic are application specific and need to be implemented. However, developers have the only task of filling the skeleton code that realizes the business logic of prosumers according to the application needs, without having to focus on possible coordination and concurrency issues.

2.2.1 Choreography realizability enforcement

Coordination Delegates are not enough to coordinate the choreography if it is not compliant with the BPMN2 specification (i.e., unrealizable choreographies). In fact, CDs have only a partial view of the whole global interaction: their view is limited to the state of the coordinated services. In fact, according to the interaction pattern described above, each CD receives (and then forwards) business messages coming from (or directed to) its associated services, executing the coordination algorithm with a partial view of the choreography, just following the sequence of the exchanged messages that is prescribed by the choreography specification. In general, it can be required that the cooperation between services has to be driven not only by the business messages exchange sequence but also by the effects that some tasks may have on the whole system environment. For this reason, it can happen that the task sequence may not fully respect the BPMN2 standard. In similar cases, in order to enforce the choreography realization, it is needed that CDs exchange coordination messages so that they can have a larger view of the system and then coordinate the interaction according to the global state of the choreography. The exchange of coordination messages is performed in a fully-distributed way, i.e., without a central point of control.

As analyzed in [6], this coordination is needed when the choreography specification contains *independent sequences* of messages and *independent branches*. From a BPMN2 point of view, those cases are simply translated into task sequences that are not allowed in the BPMN2 specification. In particular, independent sequences and independent branches are those sequences of tasks in which the initiating participant of a task does not appear as a participant of the immediately previous task(s). In other words, at run time, during the execution of the interested sequences, there is no suitable message exchange between the involved service instances that can "naturally" imply the correct synchronization of their interaction from the inside, according to the choreography specification. In fact, the initiating participant of a task does not have any knowledge of the execution

state of the previous one(s), thus, there is no way for the participant to enforce the synchronization of the sequence.

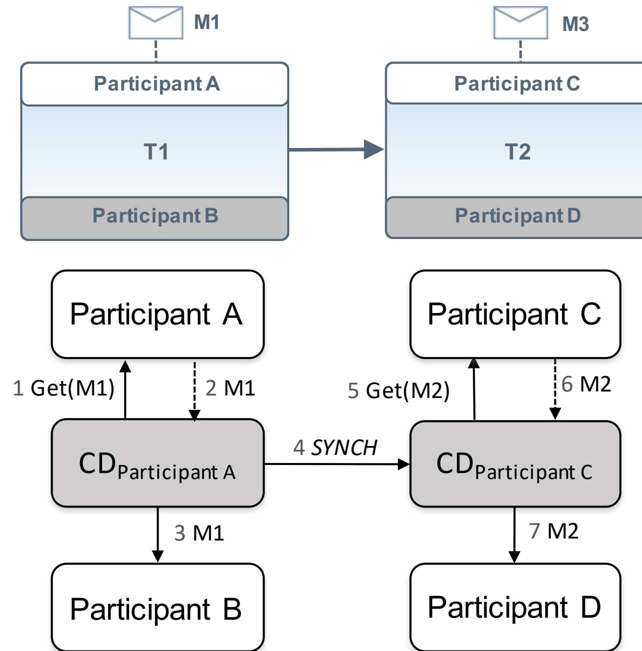
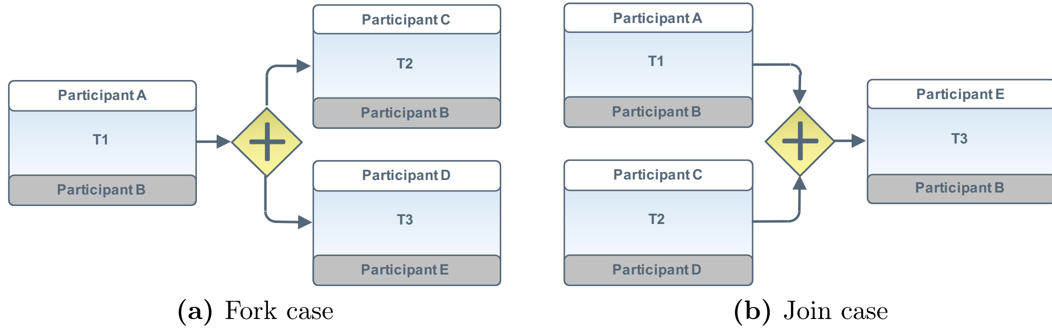


Figure 2.10: Coordination of independent sequence of tasks

Figure 2.10 shows the base case for an *independent sequence* of tasks, represented in BPMN2 syntax, together with its associated coordination architecture that we are proposing. Here, the initiating participant of `T2`, *Participant C*, does not appear as a participant of `T1`. The coordination of this kind of sequence is realized through a `synch` message, which is sent from the CD handling the first task of the sequence to the CD handling the following task, as soon as the first task of the sequence is completed. Referring to Figure 2.10, after that $\text{CD}_{\text{Participant A}}$ forwards the message `M1` to Participant B (interaction 3), it sends the `synch` message to $\text{CD}_{\text{Participant C}}$ (interaction 4). Only after that $\text{CD}_{\text{Participant C}}$ has received the `synch` message, it can ask Participant C for the message `M2` (interactions 5 and 6) that has to be sent to Participant D. In this way, $\text{CD}_{\text{Participant C}}$ can extend its view of the system including the status of the services that act in its environment and with whom it has to coordinate, hence enforcing the correct choreography execution.

The base case described above can be extended in order to realize the coordination mechanism for more complex cases, in which the independent sequence of tasks goes through a parallel gateway (*Independent sequences across independent branches/parallel flows* [6], as in Figure 2.11).



(a) Fork case **(b)** Join case

Figure 2.11: Independent sequences across independent branches

Following the same approach, the coordination mechanism can be realized with `synch` messages sent from all the CDs that are handling the tasks immediately preceding the gateway (as soon as each task is completed), to the CDs associated with the initiating participants of the flows immediately following the gateway. In particular, when the gateway node realizes a fork, like in Figure 2.11a, a `synch` message will be sent by $\text{CD}_{\text{Participant A}}$ to $\text{CD}_{\text{Participant C}}$ and $\text{CD}_{\text{Participant D}}$, letting them to continue the choreography execution in two parallel flows; otherwise, when the gateway realizes a join (Figure 2.11b), there will be two `synch` messages, sent by $\text{CD}_{\text{Participant A}}$ and $\text{CD}_{\text{Participant C}}$, that will be received by $\text{CD}_{\text{Participant E}}$: the latter will continue the execution of the choreography with the task *Choreography Task 3* only after it has received the two synch messages, realizing the join mechanism. When, in the choreography model, the two cases discussed are displayed together, i.e., a join node followed by a fork node, the `synch` messages are sent by all the CDs preceding the join node to all the CDs following the fork. In this way, CDs are aware of the state of the choreography including tasks in which they are not involved, and a mechanism for waiting for the correct choreography state before executing a task is realized (distributed coordination for choreography realizability enforcement).

Since the CD generation is performed starting from the participant model, the latter has to contain information about the coordination messages exchange. For this reason, the BPMN2 specification is extended by introducing a new flow node. It is automatically added in the participant model during the *Choreography Projection* when one of the sequences described above is found. This node, called *Synch* node, is defined through an extension of the BPMN2 metamodel, and it is placed in the participant model before (or after) each task that composes an *independent sequence* in which the participant is involved.

Figure 2.12 shows the metamodel of the BPMN2 extension needed to model

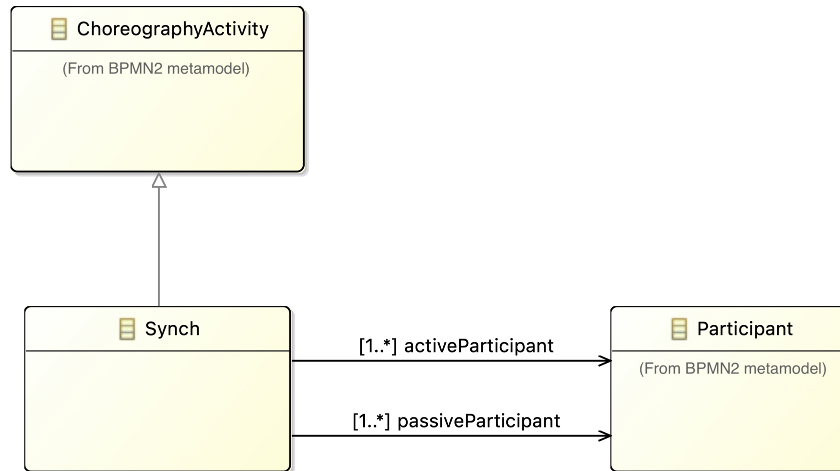


Figure 2.12: Synch node metamodel

the Synch node. As an extension of the BPMN2 metaclass *ChoreographyActivity*, the metaclass *Synch* is in relationship with the BPMN2 metaclass *Participant* through two different relations. The *activeParticipant* relation holds the information about which are the participants that are initiating the independent sequence and have to send the synch message (*active participants* means that they perform a task which causes a change in the global state of the choreography and send the synch message); the *passiveParticipant* relation holds the information about which are the participants that should receive the synch message (they are *passive participants* in the sense that they experience the context changes and only receive the synch message). For instance, referring to the example in Figure 2.11a, *Participant A* will appear as an active participant, while *Participant C* and *Participant D* will appear as passive participants in the synch model. During the generation process, when a *Synch* node is found, the *synthesis processor* includes in the CDs implementation the primitives required to send or receive `synch` messages, thus realizing the coordination logic of independent sequences.

2.3 Microservice Architecture

Microservice Architectures (MSAs) are an approach to building applications as a set of small, well-defined, distributed, loosely coupled, and autonomous services that are independently deployed and that work together by communicating through lightweight mechanisms. Each microservice implements a small and well-defined business capability and it runs its independent process [79, 93]. From the outside, each microservice of a MSA is an isolated black-box component –

embracing the concept of *information hiding* – that exposes its limited set of functionalities through networked interfaces. Microservices can be independently implemented, tested, and deployed in isolation, while the only constraint for their interoperation is the technology for letting them communicate (protocols and data encoding) [43, 93].

MSA has been proposed to overcome some of the problems arising from *monolithic systems*. In monoliths, all the interfaces, the business logic, and the databases are packaged together into a single executable artifact that is deployed and running on a single server. Due to their nature, specially when they are big-sized, monoliths are difficult to maintain, evolve, and scale [42, 79, 93, 107].

Driven by the many advantages brought by MSA in response to the limitations of monoliths (we discuss the most important in Subsection 2.3.2), companies are migrating their legacy monolithic systems into microservices [114].

2.3.1 Architectural principles

MSAs are built according to a set of core principles [43, 85, 93]. In the following, we describe the most important ones.

Bounded Context Bounded context involves grouping related functionalities into the same business capability that is implemented by one microservice [43]. Realizing microservices around the business domains allows the identification of service boundaries and their related functionalities: each microservice should realize an end-to-end slice of the business domain. In this way, business capabilities and system structure are aligned, thus making it easy to deliver functionalities independently, identify where functionalities are, update them, or enroll new ones [93].

Single Responsibility Functionalities related to the same business capability should be independently delivered by a single microservice without depending on other microservices. Each microservice should focus only on a precise feature without overlapping with other functions provided by different microservices. This principle can be summarized by saying that each microservice should *"do one thing, and do it well"*.

Lightweight Microservices should be small enough to be easily developed and maintained by a small agile development team. Note that the size of a microservice does not refer to the number of classes, lines of code, or computational

complexity of its operation. Rather, it refers to the fine granularity of the functionality that it provides, e.g., compared with the services composing a "classical" SOA system. However, a microservice must be big enough to provide a complete – yet single – business capability, being independent from other services and enhancing the loose coupling.

Independency As said, each microservice should be operationally independent from others, with the only form of communication occurring through the published interfaces. The communication between microservices should be reduced to the minimum to obtain high-cohesive and loose-coupled microservices. A change in one microservices should have no – or very marginal – impact on other microservices in the system. This independence can be achieved by separating the system functionalities in the right way according to well-defined boundaries.

2.3.2 Benefits

As already said, MSA brings a set of benefits – coming from the principles reported above – that allow overcoming the limitations of monoliths, thus making them advantageous over monolithic systems [42, 43, 79, 93, 107]. In the following, we discuss the main benefits of MSAs, also in comparison with monoliths.

Scalability The small size and isolation of microservices allow the easy replication of those services that are more exposed to high loads and that may represent a bottleneck for system performances. The reliance on domain-driven design and high cohesion means that growing loads are likely to be experienced for a well-defined and limited subset of microservices offering the functionalities that, in a given scenario, are more stressed. This means that microservices can be scaled independently according to their individual needs using *x-axis* scaling and *z-axis* partitioning [1, 107]. They can be suitably distributed on different servers, hosts, or containers, and, as a consequence, resources can be allocated efficiently by deploying microservices on hardware or platforms that provide the best resources according to their operational characteristics (e.g., CPU-intensive or memory-intensive). This property is not achievable in monolithic applications, where all the components are deployed together, and resource scaling or replication can only be performed on the whole system.

Robustness Being modular, loosely coupled, and independent of each other, a failure of one service does not have an impact on the entire system (or, in the worst

cases, it only has a marginal impact). Even if some service stops working, other services can continue to handle requests normally. Since microservices can be independently deployed and their instances replicated, a given functionality can be continued to be provided by the other running instances. Moreover, affected microservices can be quickly restarted, fixed, or replaced independently, while in monolithic systems it is needed to reboot or redeploy the whole application. These features enhance the fault tolerance, availability, and reliability of microservice-based systems.

Maintainability and ease of development Each microservice implements a restricted amount of functionalities, making its code base relatively small and easy to develop, debug, and maintain. Moreover, since they are independent and loosely coupled, they can be developed, tested, and deployed in isolation. This allows continuous delivery/deployment and continuous integration, with small teams responsible only for one or a few microservices each, with high autonomy (they only need to define the interfaces for microservices interoperability, if needed). New features or updates can be introduced with minimum effort by operating only on the affected microservices, and they can be put into the production environment by replacing old microservices with new ones, without stopping and re-deploying the whole system. This reduces the development velocity and, hence, the time-to-market.

Flexibility Each microservice composing a MSA can be implemented with a different technological stack. Developers are free to choose the language/framework/technology that is best suited for the microservice requirements, being the interface for the network communication the only constraint that limits the choice of developers. Moreover, thanks to their limited size, microservices are amenable to be rewritten using different languages or technologies. Also, the trial of a new technology is simpler and has a limited impact on the overall system, since instances of the same service running with different technological stacks can coexist. In monolithic systems, this can not happen, since the choice of the technology impacts the whole system.

2.3.3 Drawbacks

Despite having the benefits described above, as for any technology, MSA is not a *silver bullet* suitable for any context and any application. As for any architectural

style, besides benefits, there are some drawbacks that should let the choice of adopting microservices be done carefully [43, 107].

First, distribution introduces complexity to the system. Communication among services does not occur through simple method calls, but over the network, hence requiring the definition of interfaces that must be shared possibly among different development teams. The development of features involving multiple microservices is not straightforward, and their testing and deployment need to be planned carefully in order to coordinate all the involved teams. Also, it is required to ensure data integrity and coordination among the whole set of microservices, calling for an increase in the system's complexity. Moreover, operations that in a monolithic system can be simply performed locally, in a MSA may be split across different microservices. This involves the communication and transmission of data through the network, which can introduce latency to the system, being an issue if there are performance constraints over the system. Finally, skills are required to set up and manage the operational environment where microservices are deployed and whose instances are orchestrated (e.g., Docker, Kubernetes).

Another important issue concerning MSA is deciding whether or not to adopt this architecture. Starting the development of a new system from scratch by choosing the MSA, thus applying the so-called *greenfield approach* [122], may slow down the development process and increase the time-to-market. This is due to the extra effort required by the identification of the microservices, which is indeed a risky task that, if failing, may lead to incorrect system architecture. If this happens, a huge effort may be required to fix the wrong architecture. Also, splitting the domain model into microservices since the first rollout of a new system may make its evolution difficult. This is the case of startup companies, for which a *monolithic-first* approach [48, 100] can be advantageous: they can first implement their system as a monolith, migrating to microservices afterward to enable the system scale-up [93, 107].

However, as anticipated in the introduction of this work, the migration of a monolith into microservices is difficult. Establishing the boundaries of each microservice is not a trivial task and the refactoring process is time-consuming. We will dive into this topic in Chapter 4.

2.3.4 SOA vs. Microservices

As reported by Richardson, some critics of MSA claim that they are nothing new than SOA [107]. At a high level, they have some similarities, since both in SOA and MSA the main driving principle is to build systems as a collection of loosely

coupled services that communicate over the network.

SOA has been developed to break the difficulties in maintaining large monolithic applications, by focusing on the reuse of services that can be shared in different applications as black-box entities. SOA eases the maintenance of the systems since services can be replaced in an effortless manner as long as the new service shares the same communication protocol and semantics [93]. However, SOA services are not required to be independent and self-contained with their own interfaces and databases. They put the focus on the B2B communication and on the service composition (orchestration and choreographies, as in Section 2.1) rather than on the development and deployment [32, 85].

When Netflix presented its first MSA-like architecture, in 2013, it used the term *fine-grained SOA*[2]. Also according to this definition, in a certain sense, microservices inherit the SOA principles and push them forward as a way of *"doing SOA right"* [93]. MSA differ from SOA mainly for the technological stack underlying the service communication, and for their granularity. In fact, while SOA applications typically exploit heavyweight communication protocols like SOAP or leverage message processing through Enterprise Service Bus (ESB), microservices communicate through lightweight protocols like REST or make use of simple message brokers. Importantly, while each microservice is designed around a single business capability that makes it reasonably small, services involved in SOA are typically large, complex, monolithic services. As a consequence, a MSA may result to be composed of dozens of tiny – yet completely independent – services, while SOA systems are built of few larger, coarse-grained, services [107]. Given this, it is easy to understand that when it comes to decomposing monolithic applications into microservices, we may consider as "monolith" not only the service that realizes a standalone application but also one of the coarse-grained services composing a SOA.

2.4 Load balancing approaches

Load balancing is a fundamental element for scalability in MSAs [1, 12]. Since in these systems scalability is enhanced by the possibility of replicating microservice instances, load balancing allows the distribution of the workloads among the instances to optimize resource usage. Load balancing prevents a single microservice instance from resulting overloaded by incoming requests, hence maximizing system performance. This is done through one or more *load balancers*, that suitably

[2]http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html

route requests toward one of the multiple running instances of a microservice. The state of the art for microservice load balancing distinguishes between *server-side* load balancing and *client-side* load balancing [80, 115]. Also, an *hybrid* approach, made by composing these two, can be realized [7]. In the following, we describe the main architectural characteristics of the three approaches.

2.4.1 Server-side load balancing

Server-side load balancing is a centralized approach for distributing requests among microservice instances. Here, a load balancer acts as a proxy that receives requests from clients or consumer microservices and forwards them to target microservices, distributing the workload. In a fully-centralized setting, a central load balancer manages the workload of the instances of all microservice types, having all the traffic passing through it.

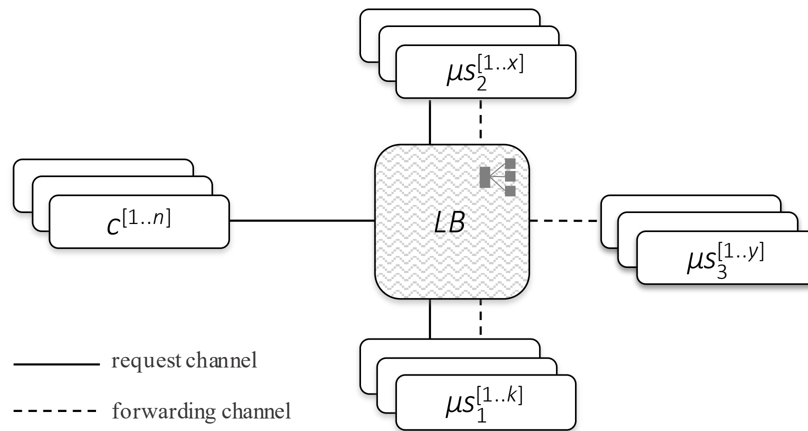


Figure 2.13: Centralized server-side load balancer

Figure 2.13 shows a simple microservice-based system having a centralized load balancer LB for the microservices μs_1, μs_2, and μs_3. The labels $\mu s_1^{[1,\dots,k]}$, $\mu s_2^{[1,\dots,x]}$, and $\mu s_3^{[1,\dots,y]}$ indicate that we are considering k, x, and y running instances for the microservices μs_1, μs_2, and μs_3. In this example, LB receives requests through the request channels (depicted with solid lines) from n instances the client c and from the instances of μs_1 and μs_2 (note that microservices can interact with each other, being in turn *consumers* of other microservices, hence they can be *prosumers*). LB forwards the requests through the forwarding channels (depicted with dashed lines) to the instances of μs_1, μs_2, and μs_3 in such a way that their workload is distributed.

It is easy to see that, in the aforementioned load-balancing architecture, all the traffic passes through the centralized load balancer. This enhances the security

of the system since the server-side load balancer "hides" the system's internal structure and prevents clients from directly interacting with services. However, the load balancer may represent a bottleneck or a *single-point of failure* for the system. A more decentralized setting for server-side load balancing considers having a load balancer per microservice type instead of a central one. Each load balancer proxifies the requests directed to a specific microservice type and manages the workload of its instances only.

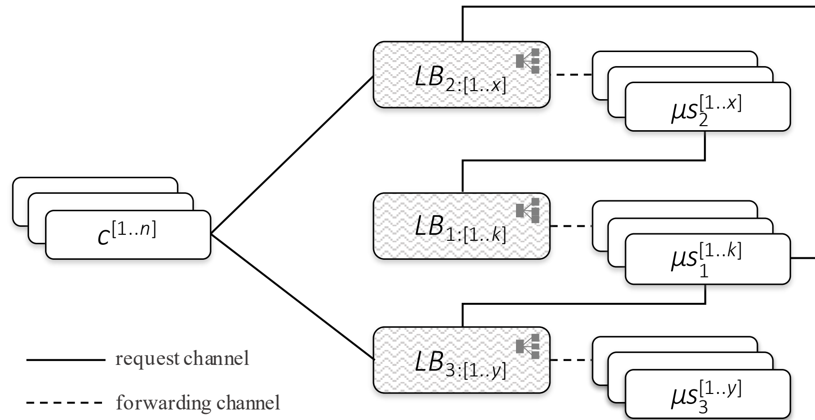


Figure 2.14: Decentralized server-side load balancer

Figure 2.14 shows the decentralized approach to server-side load balancing. Each of the load balancers LB_1, LB_2, and LB_3 proxifies the microservices μs_1, μs_2, and μs_3, respectively, and are in charge of balancing the workload of their instances. This setting reduces the possibility of bottlenecks in the system, but still, if one of the load balancers fails, all the instances of the controlled microservice become unreachable. Moreover, the server-side load balancers (in both the centralized and the decentralized setting) introduce an extra hop on the network that may increase latency. These issues can be solved by adopting a client-side load balancing approach.

2.4.2 Client-side load balancing

Client-side load balancing is a fully distributed approach, in which each client or each instance of a prosumer microservice has its local load balancer. Each local load balancer handles the requests coming from the client/prosumer microservice and routes them to an instance of the target microservice balancing the target's instances workload.

Figure 2.15 shows the sample microservice-based system with client-side load balancers. Differently from the server-side approach, in this setting, each client

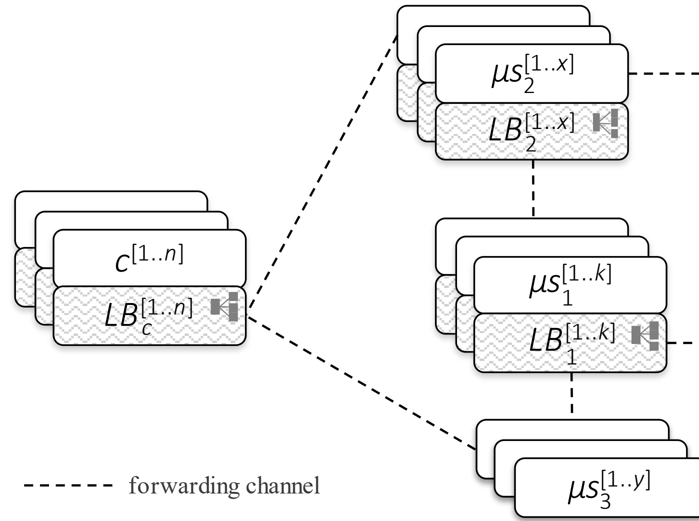


Figure 2.15: Client-side load balancer

or *prosumer* service instance has its own local load balancer, which is responsible for routing the requests toward one of the target service instances.

In this setting, there are no bottlenecks, single points of failure, or extra hops. On the other side, it is more difficult to evenly distribute the load among the available microservice instances, while the information about the internal network of the system can not be hidden and clients can directly interact with services. Moreover, the client-side approach is not realizable if the clients are not known, are not able to, or can not perform the load balancing.

2.4.3 A hybrid approach

The hybrid approach is realized by combining both client and server-side load balancers and permits balancing their pros and cons. The main idea is to have multiple instances of a server-side load balancer attached to client-side load balancers that route the traffic toward them. Each instance of a server-side load balancer proxifies only a subset of the instances of a given microservice.

Figure 2.16 shows a sample microservice-based system with hybrid load balancing. Here, the client-side load balancers for the instances of the microservice μs_2, $LB_2^{[1,\dots,x]}$ are all connected with the server-side load balancers $LB^1_{1:[1,\dots,k]}$ and $LB^2_{1:[k+1,\dots,z]}$. They proxify the interactions with two different subsets of instances of the microservice μs_1: the first balances the workload among the instances μs_1^1 to μs_1^k, while the latter balances the workload among the instances μs_1^{k+1} to μs_1^z.

In this setting, the server-side load balancer does not represent a single-point of failure. Rather, new load balancer instances (and new proxified microservice

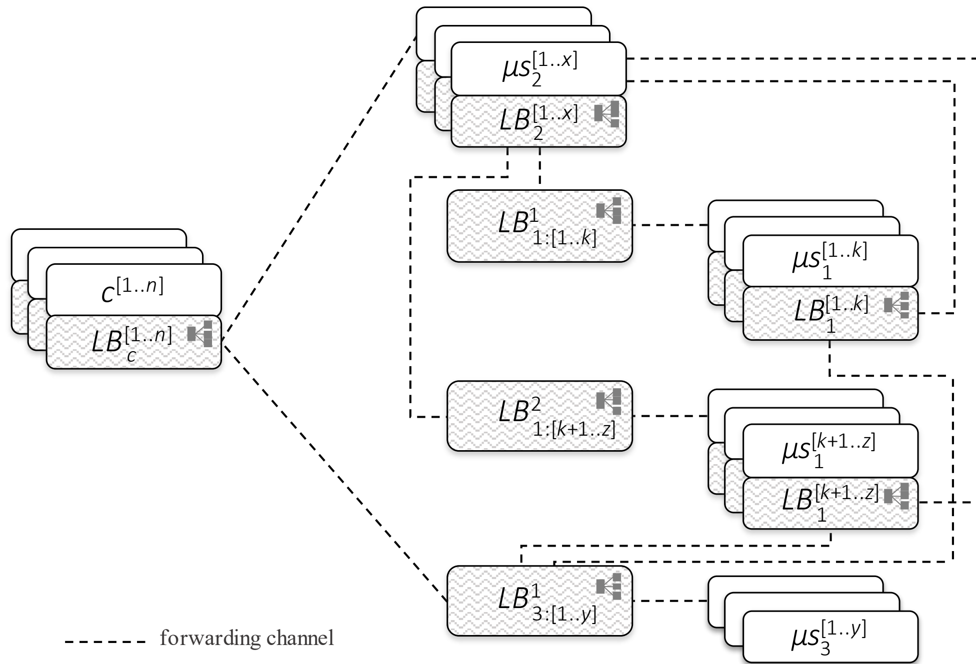


Figure 2.16: Client-side load balancer

instances) can be activated to enhance the scalability and performance when the other instances are under heavy workloads. Also, microservice instances can be suitably partitioned and their partitions dimensioned in such a way that for each of them dedicated Service Level Agreements (SLAs) are provided, hence realizing the so-called *z-axis* scaling [1, 107].

Summarizing, server-side and client-side approaches have different pros and cons, with many factors influencing their effectiveness and their fitness to the system requirements. Hybrid load balancing is not meant to be a substitutive alternative to server-side or client-side approaches. It is a complementary alternative that provides some tradeoffs between the two and mitigates their cons. They can all coexist in the system. However, the different strategies require different architectures for supporting load balancing. For this reason, an architectural style able to support all the presented approaches is desirable.

In Chapter 5 we present an architectural style capable of supporting the load balancing strategies described above, we compare them, and we discuss in detail the features emerging from their usage.

Chapter 3

Synthesis of context-aware choreographies

As discussed in the previous Chapter, SOA systems are realized by composing third-party and ready-to-use services. They enable the realization of added-value applications capable of benefitting from the interactions among the involved services and offer complex and dynamic functionalities to users. In those systems, context needs to be considered to provide the desired functionalities by adapting the system behavior according to the current conditions of the execution environment and users' needs. However – being the services usually black-box and owned by third parties – context awareness and proper adaptation can not be realized by acting directly on the composed services; rather, it has to be realized by acting on their interaction.

This chapter addresses the problem of realizing context-aware choreographies. The proposed solution extends the CHOReVOLUTION approach for the automated composition and synthesis of service choreographies described in Section 2.2. In the following, we first introduce the context and the challenges that arise when dealing with context awareness. Then, we describe how we leverage the BPMN2 notation to express the variability required to define the adaptable behavior of context-aware choreographies. Next, we present the reference scenario and discuss its intrinsic complexities. Finally, we present the approach in detail, evaluate it against the reference scenario and discuss the experimental results.

3.1 Dealing with context-awareness

Consistently with the definition given by Dey [41], we consider *context-awareness* as the ability, of software systems, of using context information in order to dynamically adapt according to the user needs. Context-aware systems are able to "sense" the *context* and react to changes by adapting their behavior [110].

Through the years, many definitions of *context* have been given in the literature, as many are the application domains in which context can be involved. However, context is often defined by enumerating the characteristics describing the application domain that is considered in each work [127].

Bauer and Novotny [20] reviewed the notions of context across the literature and grouped all the different dimensions that have been taken into account when representing context in three main categories: (i) *social context*, which includes the information about user identity, preferences and habits, emotional state, social and cultural environment, presence and behavior of other people; (ii) *technology context*, including all the characteristics of the physical and virtual platform on which the system runs, such as computing resources, performances, network; (iii) *physical context*, including all the observable elements of the environment in which the system operates, like the functional environment (e.g., indoor/outdoors, car, home), weather, lighting, time, location, movement. A general definition of context, which can in principle be accepted beyond any specific application domain, can be found in [41] where the context is defined as *"any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves"*.

A series of challenges that arise when dealing with context-aware systems specifically concerns the representation of the context, the acquisition of the information, and the overhead introduced for the context management [109]. For instance, the *context model* has to consider the heterogeneity of the sources of information that are exploited in order to acquire context data. In fact, some context data can be acquired from sensors placed in the environment, while other data can be provided by the user and mobile devices that interact with the system, or can be obtained from external sources like databases and web services. Moreover, the context model has to allow deriving higher-level context facts from the raw data obtained by, e.g., sensors and other external sources, and to allow reasoning to decide whether any adaptation is necessary. In addition, the formalism used to define the context models has to be easily understandable by system

designers [23]. Thus, in order to allow context-aware systems to adapt according to the runtime context, a mechanism capable of acquiring information from different sources, delivering it to the system, and providing context reasoning functions is needed [23, 29, 60, 75].

With particular reference to SOA and choreographies, realizing context-awareness means being able to adapt the interactions between participants according to the context conditions. The adaptation has to be autonomous and automatic and has to consider the dynamic nature of the context, which can change continuously during the system runtime [96]. Accordingly, the notion of context-aware adaptation we are considering in this work goes beyond the pure *reconfiguration* of choreographed systems, which can be realized by modifying the choreography specification and evolving the system accordingly (in response to changes of functional or non-functional requirements [77]). Dynamic adaptation means that the portions of the choreography requiring adaptation may not be fully specified at design time. Rather, the choreography specification must be able to express *variability*, hence allowing system designers to specify where adaptation is required and what are the adaptation possibilities. Then, the variable context-aware portions of the choreography (the ones that realize the adaptation) can be left *underspecified* and be *concretized* at runtime when the context conditions are known. That said, we consider three different levels of runtime adaptation for context-aware choreographies:

- **Message-level adaptation** - to adapt the content of one or more messages exchanged between the participants (*context-aware messages*);
- **Participant-level adaptation** - to select, at runtime, the instance(s) of a participant service that has to be involved in a task. Participant instances are selected among a pool of service instances that can play the participant's role (*context-aware participant instantiation*);
- **Task-level or Task-flow-level adaptation** - to select, at runtime, a suitable flow of tasks for a choreography portion that has to be executed, among a set of possible interactions (*choreography variants*).

Note that, in performing the *participant-level adaptation*, we assume for the sake of simplicity that all the selectable service instances share the same interface and interaction protocol, i.e., they can be integrated without any interface adaptation. This concern is already addressed in the CHOReVOLUTION framework as described in [4]. Moreover, we assume that the selection of participant services or variants can only occur when involved parties (i.e., the services to be selected) are stateless or inactive, hence their substitution does not introduce

inconsistencies in the system. These assumptions are discussed in Section 3.7.

In the following, we present how variability can be expressed in the BPMN2 choreography specification.

3.2 Expressing variability

As said in the previous section, realizing dynamic adaptation requires expressing *variability* in the choreography specification. This calls for modeling constructs that allow modeling *underspecified* portions of the BPMN2 choreography model.

Concerning the message-level adaptation realized through context-aware messages, there is no need to leave unspecified portions of the choreography. In fact, the structure of the message must always be defined in the choreography specification since already the beginning and the content of the message is always a runtime matter. To realize this kind of adaptation, it is enough to suitably extend the message construction logic (as described in Section 2.2) in order to take into account the runtime context information.

When defining tasks that require context-aware participant instantiation, we express variability by using *multi-instance participants* (as described in Section 2.1 and in Figure 2.6) to specify that there exist several related participant services that can be involved and dynamically selected.

In order to define at design time the choreography variants required to realize the task-level or task-flow-level adaptation, we provide a novel choreography specification construct, called *variation point*, that allows system designers to specify the alternative flows of message exchanges for those portions of the choreography that are mission-critical or that can be more affected by context changes. In these cases, the choreography alternatives realizing adaptation do not depend on the content of business messages or the behavior of users and involved services. Thus, alternatives cannot be specified by means of the usual *alternative flows* of the BPMN2 choreography diagram. Yet, they represent pure *variations* of the choreography and can be properly specified through the definition of *variation points* [95]. Our concept of variation point differs from the one proposed for SPLs and Systems Families [11, 86, 123], in the sense that, as explained, variants do not represent different specifications of a choreography. They are only alternative portions of the same choreography that are dynamically selected and executed at runtime according to the current context conditions.

Variation points are not natively supported by BPMN2. In order to provide developers with a means to specify them, we extend the BPMN2 metamodel with

a novel modeling construct. A variation point contains information about the participants involved in the alternative variants of the variation point, which are defined as choreographies associated with it. As shown in Figure 3.1, variation points are introduced through a novel ad-hoc metaclass in the BPMN2 metamodel, realized as an extension of the *ChoreographyActivity* metaclass trough the inheritance relation. The new *VariationPoint* metaclass owns a one-to-many relationship with the novel *ChoreographyVariant* metaclass, which is an extension of the *Choreography* metaclass. It allows associating the definition of the variants with the variation point element.

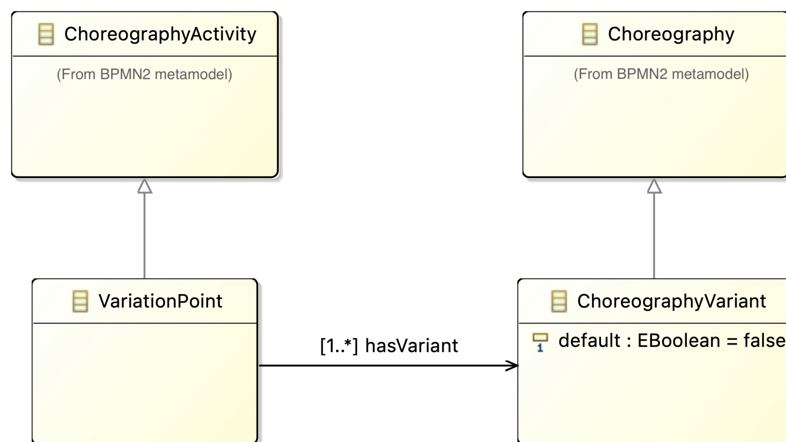


Figure 3.1: BPMN2 metamodel extension for variation points

Variation points are represented graphically by using a construct that is similar to the sub-choreographies – without the "+" mark in the box body – containing the information about the participants involved in the alternative variants: the initiating participant of the first task of each variant is put into a white round-cornered box; the other participants in the variants are put into grey boxes. Figure 3.2 shows the graphical notation of the variation point in a sample choreography with a task (*Task*) and a variation pooint (*Variation Point*). The first tasks of the variants associated to the variation point are initiated by *Participant A* and *Participant B*; the other participant involved in the variants are *Participant C* and *Participant D*.

From here on, we will refer to both variants and participants as *adaptable entities*. The concretization of choreography variants and multi-instance participants is realized at runtime by selecting, among a set of candidate variants or participant services, those which are suitable according to the runtime context of the system.

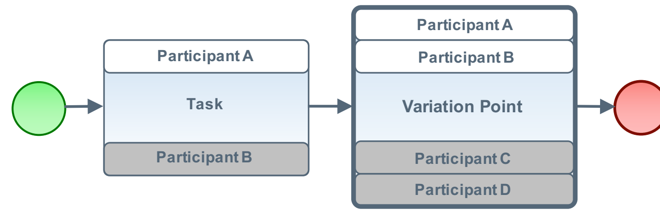


Figure 3.2: Sample choreography with variation point

3.3 Reference scenario

The scenario described in this section refers to a real use case implemented in the ConnectPA project and shows how Public Administration (PA) systems can be realized through composition of a number of services exposed by different authorities. The system in our scenario aims to dematerialize and digitalize the whole process, from the request to the payment, for billposting of posters in public spaces of towns.

The "classical" procedure for public posting requires a citizen submitting a request to the municipality. If free posting spaces are available, then the municipality sends the bill back to the citizen for enabling the payment. In order to ease and dematerialize this process, municipalities expose their own services that manage the billposting, allowing users to search for the available spaces in the municipality's territory, hence sending the posting request for a set of selected spaces, and receiving the related bill. However, in most cases, the postings involve different municipalities. Even if the municipalities expose their own services, citizens have to repeat the process many times for each municipality, and there is no integration with a unified payment service. For this reason, our system allows interoperability among the services of different municipalities and other services offered by the PA (e.g., a payment service for the PA).

The user interacts with the system through a mobile application that allows his/her to pinpoint the geographical area in which he/she wants to affix posters (by selecting a radius starting from his/her current position) and specify the duration of the postings. Then, the system interacts with the services that handle the billposting of each involved municipality and retrieves the list of available spaces. The application shows the available spaces to the user so that he/she can filter them and select the ones where he/she wants to affix posters. The selected spaces are then added to a virtual cart. After the user has confirmed the request, the system communicates with the services of each municipality in order to get the payment bills. In the end, the user proceeds with the payment through the mobile app. If the payment service is available, the payment is processed, a

receipt is sent back to the user and the confirmation of the payment is sent to the involved municipalities. Otherwise, a payment invoice is sent to the user that can pay the bill offline.

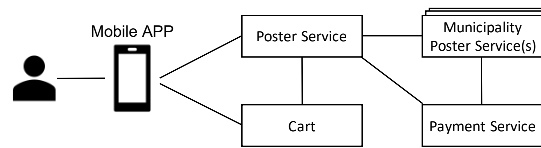


Figure 3.3: System connections

With reference to Figure 3.3, the system is realized as a composition of the following participants:

- *Mobile APP*: it is the client of the system. It interacts with the other participants for the request of posting spaces, the selection of the spaces, and the confirmation of the request. Moreover, it allows the user to pay for the bills required to affix posts.

- *Poster Service*: it is a *prosumer* service that includes some of the business logic of the system. As a provider, it receives requests from the Mobile APP to get the list of available spaces, the confirmation of the selected ones, and the billings for the payments. As a consumer, it communicates with the services of the municipalities and with the payment service.

- *Payment Service*: it is a *prosumer* service that offers the interfaces that are needed to receive the information for the payments and to pay them. As a consumer, it interacts with the municipalities in order to send the confirmation of the paid bills.

- *Cart*: it is a *provider* service that exposes the interfaces needed to add, remove, and get all the elements of a virtual cart.

- *Municipality Poster Service(s)*: these are the *provider* services, offered by the municipalities, that allow sending requests for public postings.

Figure 3.4 shows the BPMN2 diagram of the choreography specifying the way the above mobile app and services must interact.

The *multi-instance* marker (i.e., a set of three vertical lines) is added to the participant band of the choreography tasks to express *variability* needed for the *participant-level adaptation* when the instances of a participant that have to be involved in a task are not known *a-priori*. This is the case of the *Municipality*

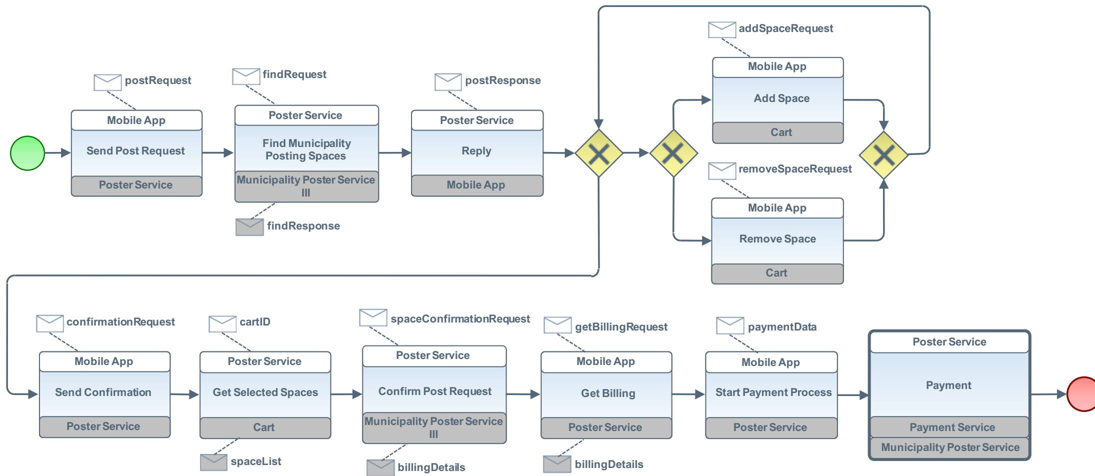
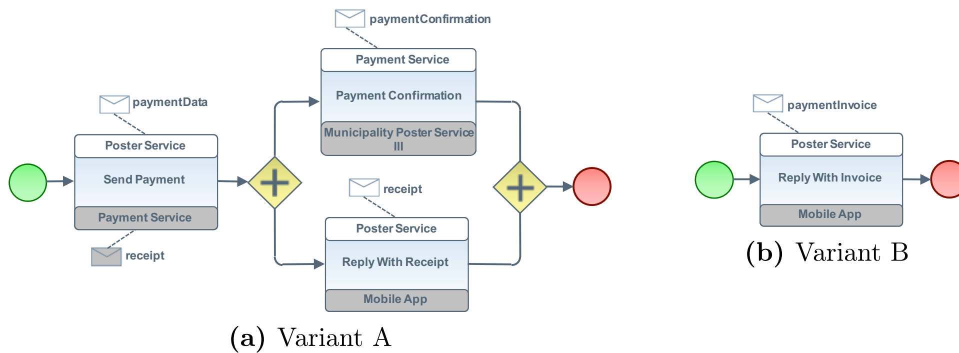


Figure 3.4: Reference scenario: Public billposting choreography

Poster Service. In the proposed use case, each municipality provides its own service holding information related to the posting spaces in the municipality territory: the service to be involved at runtime in the process is not known since it depends on the locations selected by the user.

Variation points represent the underspecified portions of the choreography needed for the *task-level adaptation* when a portion of the choreography to be executed has to be selected at runtime. This is the case of *Payment*, which, according to our scenario, has to consider the availability of Payment Service. Figure 3.5 shows the two sub-choreographies that have been defined as *variants*. The first (Variant A) models the interactions that occur if Payment Service is available; the second (Variant B) models the interactions that occur otherwise.



(a) Variant A

(b) Variant B

Figure 3.5: Choreoraphy variants for Payment variation point

3.3.1 Arising complexities

The proposed scenario shows some complexities due to the possibility of having race conditions, the presence of multi-instance participants, choreography variants, and independent sequences of message exchanges. Depending on the sequence or timing of the message exchange, a race condition arises when more than one *Mobile App* is simultaneously attempting at gaining free posting spaces from the same municipality. It is easy to see how this kind of race condition is more difficult to manage in the presence of multi-instance participants. In fact, multi-instance participants require dealing with *multicast* group communication where the related message exchange is addressed to a group of participants "simultaneously". This is the case of the multiple *Municipality Poster Service* instances (see for instance the task *Confirm post order*). Possible race conditions, unknown service instances, and independent sequences of message exchanges make things worst. The sequence of tasks *Confirm Post Request* $\rightarrow$ *Get Billing* represents an *independent sequence* for which coordination is needed in order to enforce the correct choreography realization and prevent *undesired interactions*. The situation here is even more complicated due to the presence of multi-instance participants, whose number of instances is not known in advance. Race conditions and independent sequences are then the main cause of undesired interactions, i.e., those interactions that are not prescribed by the choreography but may occur if the services were left free to interact without any control. That said, considering the reuse of black-box services, and the risk of having undesired interactions, external coordination exploiting additional synchronization messages is required. Moreover, for what concerns the mobile app, another important aspect, which further complicates matters, is that it can autonomously engage in the interaction, out of the blue, in the middle of the independent sequences, without synchronizing with either the *Poster Service* or the *Municipality Poster Service*.

Beyond pure coordination issues, which can be addressed as explained in Section 2.2.1, the scenario has also context-aware coordination issues. In fact, since each municipality offers its own service for billposting, the system needs to ask for the availability of posting spaces to the specific (location-based) service that covers the area surrounding the user. For this reason, *Municipality Poster Service* has been modeled as a multi-instance participant to support *participant-level adaptation*. Thus, a dynamic service selection mechanism is needed in order to correctly access and coordinate the right service at the right location. Also, the request message sent to each service may differ in its content since the set of involved services is not known a priori; rather, it requires *message-level adaptation*

depending on the runtime context (user's location). Moreover, the choreography considers the availability of the Payment Service in order to let the user complete the billposting process, even in case of unavailability of the payment system. The alternative interactions that can occur according to the availability of the service have been modeled through choreography variants that are associated with the variation point. A mechanism for the dynamic selection and execution of the right variant is needed in order to realize the *task-level adaptation* according to the runtime context (service availability) and coordinate the service interactions in the different scenarios. For these reasons, basic CDs described in Section 2.2 need to be extended in order to handle those cases in which (i) a service or a variant selection needs to be performed at runtime, (ii) the request messages have to be composed according to the runtime context and then (iii) suitably coordinated with the running choreography. This is where the novel approach described in this thesis comes into play and makes the difference by introducing a mechanism for the definition, acquisition, and manipulation of the context. A further advance with respect to the state-of-the-art is represented by the new notion of Context-aware Coordination Delegate (caCD) that, extending the behavior of CDs as mentioned, permits bringing together the ability to solve the above-described coordination issues together with the context-related ones.

3.4 Approach description

The novel approach for the automated synthesis of context-aware choreographies proposed in this thesis extends the *sota* on choreography synthesis (Section 2.2) by:

1. Defining a suitable context model;
2. Realizing means to acquire and manipulate the needed context information;
3. Implementing a method to select the candidate *adaptable entities* for the context-aware adaptation at runtime, thus concretizing *underspecified* choreography portions;
4. Executing the resulting adaptation.

Figure 3.6 overviews the approach. It considers three phases: *system design*, *choreography synthesis*, and *choreography refinement & enactment*. In the design

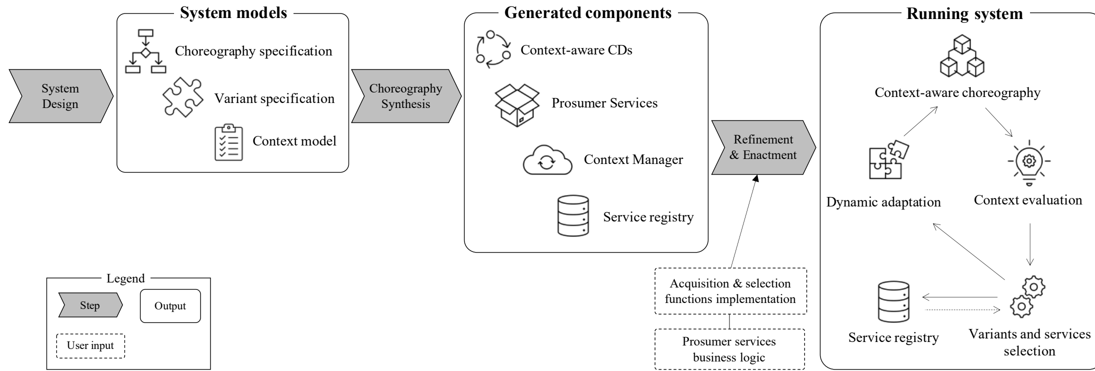


Figure 3.6: Approach overview

phase, system designers realize the BPMN2 choreography specification, the specification of choreography variants (if any), and the model of the context. These artifacts are taken as input for the synthesis phase, during which a synthesis algorithm performs a model-to-code transformation and generates: (i) the skeleton code of the prosumer services, (ii) the set of *Context-aware Coordination Delegates (caCDs)*, (iii) a *Context Manager* service and (iv) a *Service Registry.* CaCDs are an enhancement of the basic CDs introduced in Section 2.2. Beyond coordinating the service interactions, they allow sending context-related messages and performing the runtime adaptation by invoking the selected service instances or executing the selected choreography variant. The Context Manager is a service that gets and holds information about the current context conditions by holding an instance of the context model. The Context Manager acquires the context by receiving *context-carrying messages* or by executing *ad-hoc* written *acquisition functions* and selects the adaptable entities through *selection functions.* The Service Registry holds the description of all the service instances that can be selected for the tasks with multi-instance participants. The Service Registry offers an interface that is exploited by the Context Manager – in order to get the service descriptions – and by the system administrators – in order to add new service instances to the registry.

The implementation of the generated components is completed in the choreography refinement phase. Here, developers provide the business logic of prosumer services and the implementation of acquisition and selection functions of the Context Manager. Then, the choreography can be deployed on a cloud infrastructure and enacted: the resulting system will be able to "sense" the context and dynamically adapt by dynamically instantiating the underspecified portions of the choreography.

In the following, we will describe how: (i) the context is modeled, (ii) the

generated system handles the context information, and (iii) the adaptation is performed. We will also portray some examples from our reference scenario to better explain the process.

3.4.1 Context model

The context model allows defining:

- The context of the system and of adaptable entities;
- How the context is acquired;
- How the context is evaluated and how entities are selected.

Representation of the context information

The information that is needed to represent the context can be grouped into two categories:

- *System context*, which includes all the attributes that are common to the whole system (e.g., user preferences, physical and technological characteristics of the environment, weather, location, availability of services, devices, and common resources);
- *Entities context*, which describes the dynamic conditions of each adaptable entity (e.g., performances and available resources) and their static properties that never change at runtime (e.g., service cost, location reference, resource requirements).

Figure 3.7 shows the metamodel that allows defining the context model. The model describes the context as a composition of the metaclasses `SystemContext` and `EntityContext`. The metaclass `ContextAttribute` is used to define the dynamic characteristics of the context of both system and entities, while `StaticProperty` metaclass allows the definition of the static properties of the entities context. Each entity context is associated with an *Entity Class*. The latter allows the description of the set of service instances of a multi-instance participant, or the set of variants associated with a variation point.

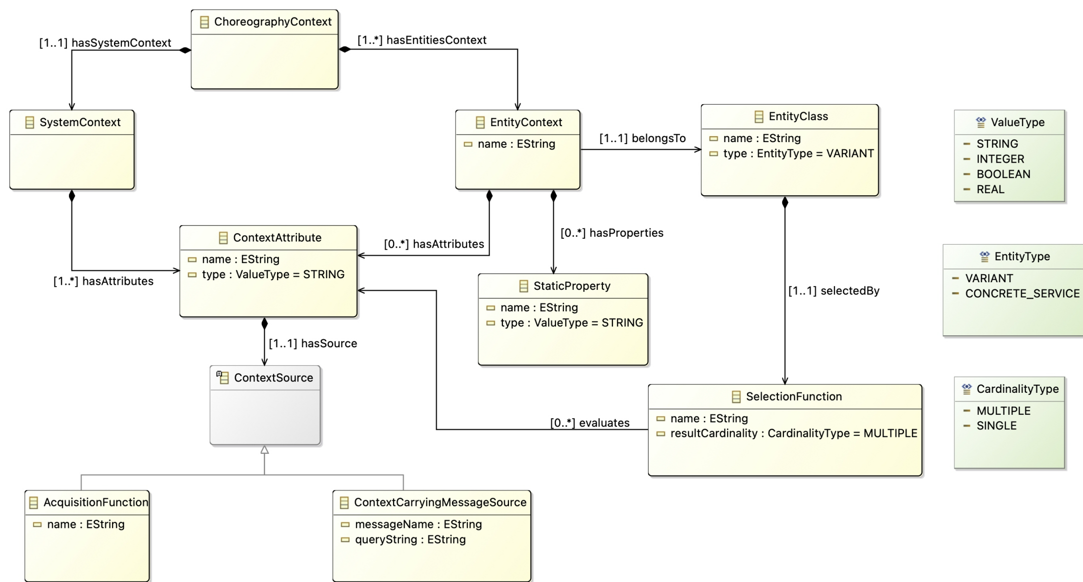


Figure 3.7: Context metamodel

Support for context acquisition

In order to deal with the heterogeneity of context sources, our approach considers two different means for acquiring the runtime context data: ad-hoc *acquisition functions*, or *context-carrying messages*.

Acquisition functions allow acquiring context data from any source, like the external environment, devices, or web services. They are declared as *Context Source* for a context attribute through the `AcquisitionFunction` metaclass. During the choreography synthesis, the skeleton code of the declared acquisition functions is generated, while their implementation needs to be provided by developers in the choreography refinement phase.

Further information about the context can be found in the content of business messages exchanged by participants. In fact, choreographies may involve devices equipped with sensors, which are able to provide runtime context data. For instance, messages sent by client applications running on smartphones may contain data about the user's location. Those messages are already defined in the BPMN2 choreography specification: we call them *context-carrying messages*. Thus, when the value of a context attribute can be acquired from a context-carrying message, the latter can be used as a context source instead of defining an acquisition function. The definition of a `ContextCarryingMessageSource` requires the *XPath* expression that allows extracting the needed context information from the message. Unlike acquisition functions, the code needed for the

acquisition of the context data from context-carrying messages is generated from the context model and does not need any further refinement.

Support for context evaluation and entities selection

In order to provide support for the evaluation of the context and the selection of the adaptable entities, the metamodel allows the definition of a *selection function* for each entity class. The definition of an instance of the `SelectionFunction` metaclass requires the reference to the context attributes that have to be evaluated and the cardinality of the result. For instance, a single result is required when selecting a variant, while multiple results may be required for the selection of the service instances for a multi-instance participant.

Note that the context model only allows the definition of the context attributes and entity properties to be evaluated by the selection function and the cardinality of the result. In fact, as for acquisition functions, during the modeling phase, the selection functions are only defined, while their implementation is left to the programmers and needs to be provided in the refinement phase. This means that the selection function implementation is always responsible for, e.g., always providing a default choice, ensuring consistency on the selected entities, and being able to always return (at least) an adaptable entity for any possible combination of the values of the context attributes and entity properties.

Figure 3.8: Context model for the reference scenario

Figure 3.8 shows the context model of our reference scenario. The characteristics that constitute the context are the user's position, the selected searching

radius, and the availability of the Payment Service. These are modeled into the *System Context* through the context attributes *userLatitude*, *userLongitude*, *searchingRadius* and *paymentAvailability*. Concerning the context sources of these attributes, note that the data about the user's position and the selected searching radius can be provided by the Mobile App, which exploits the user's phone GPS sensors in order to acquire the GPS position, and obtains the searching radius from the user's input. The message *postRequest*, sent by the Mobile App through the task *Send post request*, contains these data. Figure 3.9 shows the XML schema of the message. *postRequest* can be exploited as context source for the context attributes *userLatitude*, *userLongitude* and *searchingRadius*. Thus, a `ContextCarryingMessageSource` is defined in the context model for each of them, with the related *XPath* string: `/locationCoordinates/latitude` for the user latitude, `/locationCoordinates/longitude` for the user longitude, and `/searchRadius` for the searching radius. In contrast, an acquisition function is defined in order to get the runtime value of *paymentAvailability*: `getPaymentAvailability`. The function allows checking the availability of the service by contacting it.

```
<xsd:complexType name="postRequest">
   <xsd:sequence>
      <xsd:element name="requestID" type="xsd:string"/>
      <xsd:element name="startDate" type="xsd:date"/>
      <xsd:element name="endDate" type="xsd:date"/>
      <xsd:element name="locationCoordinates">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element name="latitude" type="xsd:double"/>
               <xsd:element name="longitude" type="xsd:double"/>
               </xsd:sequence>
         </xsd:complexType>
      </xsd:element>
     <xsd:element name="locationName" type="xsd:string"/>
     <xsd:element name="searchRadius" type="xsd:integer"/>
   </xsd:sequence>
</xsd:complexType>
```

Figure 3.9: XML schema of the context-carrying message "*postRequest*"

Since the scenario involves a multi-instance participant and a variation point, two different `EntityClasses` have been defined: *MunicipalityPosterService* and *PaymentVariant*. For each of them, an entity context is defined. *MunicipalityPSContext* defines the properties of the Municipality Poster Service instances. Since they are characterized by the location of the municipality – which never changes at runtime – their entity context is composed by the static properties *referenceLatitude* and *referenceLongitude*. *PaymentVariantContext* defines the properties of the variants defined for the variation point. They are characterized

by the availability of the Payment Service, i.e., the condition used to select a variant. Thus, their entity context is composed by the *paymentServiceRequired* property. Finally, for each of these two entity classes, a selection function has been defined. For *MunicipalityPosterService*, the selection function *selectMunicipalityServiceInstances* is defined. It evaluates the context attributes related to the user location and the searching radius, and returns a result with a multiple cardinality, selecting all the services whose reference location is in the selected area. For the variation point, the function *selectPaymentVariant* is defined. It evaluates the *paymentAvailability* context attribute and returns a single result (the selected choreography variant).

3.4.2 Context-aware CDs

Context-aware Coordination Delegates (caCDs) are introduced to execute the choreography adaptation at runtime. They manage:

- The execution of variants;
- The context-aware participant instantiation;
- The exchange of context-aware and context-carrying messages;
- The communication with the Context Manager.

The whole logic of caCDs is automatically generated.

For "normal" tasks, caCDs behave as basic CDs (see Figure 2.8), by also exchanging `synch` messages (Section 2.1) to coordinate independent sequences of tasks. CaCDs execute an extended version of the interaction pattern of basic CDs. This extension accounts for: caCDs involved in tasks with multi-instance participants; CDs that handle the execution of a variant; caCDs whose supervised participant sends a context-carrying message. Thus, in the following, we describe the interaction pattern of the caCDs when: (i) receiving context-carrying messages, (ii) performing *context-aware participant instantiation*, and (iii) executing choreography variants. In so doing, for the sake of simplicity, we consider the Context Manager as a black-box entity. Its behavior will be discussed in Section 3.4.3.

Exchange of context-carrying messages

Figure 3.10 shows the interaction pattern for a general case represented by a task T1 in which a context-carrying message (message M1) is sent by a *consumer* A to

a *prosumer* B. When receiving M1, the $caCD_A$ forwards it (as a basic CD would also do, interactions 1 and 2), then it sends M1 to the Context Manager in order to be processed (extension of the basic pattern, interaction 3). When handling the response message, it behaves like a basic CD (interactions 6 and 7).

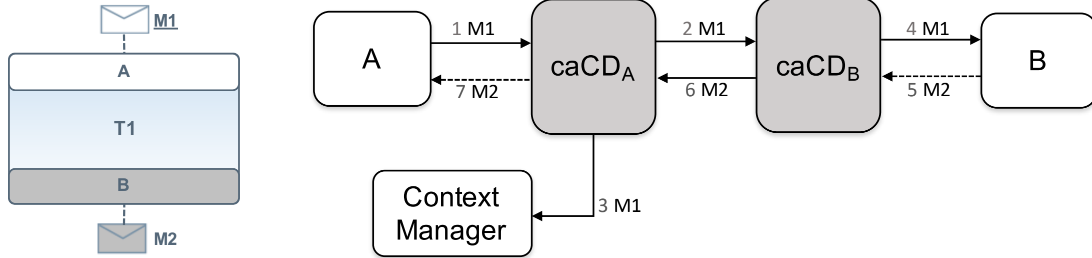


Figure 3.10: Context-aware CD interaction pattern for context-carrying messages

In our reference scenario, this interaction occurs for the task *Send post request* since it involves the context-carrying message *postRequest*. When receiving this message, the caCD of *Mobile App* forwards it both to the caCD of *Poster Service* and to the Context Manager. The latter will get the values of the context attributes *user latitude*, *user longitude*, and *searching radius* from the message content.

Context-aware participant instantiation

Figure 3.11 shows the interaction pattern for a general task T2 in which a context-aware participant instantiation is needed for a multi-instance participant C. In order to execute the task, the $caCD_B$ asks the Context Manager for selecting the instances of C that have to be involved (interaction 1). The Context Manager returns the set of the endpoints of the selected instances, together with their entities' context and the system context (interaction 2). Then, $caCD_B$ will execute the context-aware participant instantiation. For each service instance C_i in the list, $caCD_B$ will: (i) get from B the instance of the message M3, related to the context associated with the instance C_i (interactions 3 and 4); (ii) send the message to C_i, receiving back the response (interactions 5 and 6); (iii) forward the response message to service B (interaction 7).

The participant instances are invoked in parallel. Thus, interactions from 3 to 7 are concurrently executed for each selected service instance. Note that the message M3 is built by considering the runtime context, which is provided by the Context Manager alongside each selected service instance endpoint. In this way, also the *message-level adaptation* can be realized, as discussed in section 3.1.

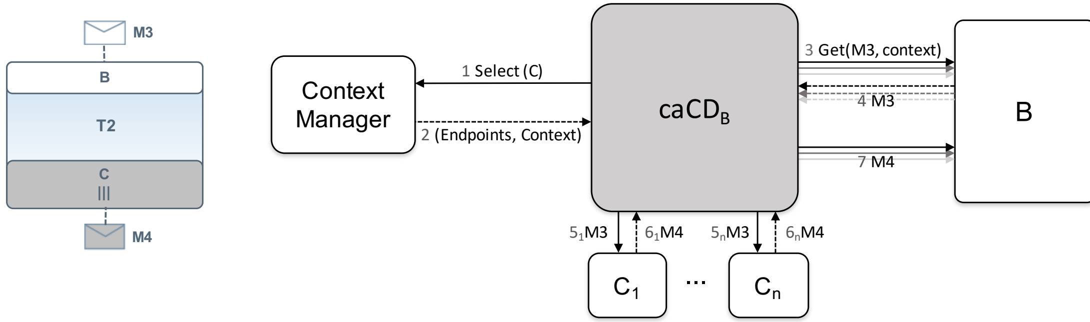


Figure 3.11: Context-aware CD interaction pattern for context-aware participant instantiation

In our reference scenario, the context-aware participant instantiation is performed for the tasks *Find Municipality Posting Spaces*, *Confirm Post Request*, and *Payment Confirmation* (in the Variant A of Figure 3.5). They all require the runtime selection of the instances of *Municipality Poster Service*. For the first two tasks, the interactions described above are performed by the caCD of *PosterService*, since the latter is the initiating participant of the tasks. For the task *Payment confirmation*, the instantiation is handled by the caCD of *Payment Service*.

Variants execution

Figure 3.12 shows the interaction pattern for the selection and execution of a choreography variant. Here, it is important to know which are the initiating participants of the first task of the variants. That is, A and B in Figure 3.12. After the execution of T1, the caCD of A (caCD$_A$) asks the Context Manager for the selection of the variant associated with the variation point VP1 (interaction 1). The Context Manager returns the name of the selected variant (interaction 2). Then, if the initiating participant of the selected variant is B, caCD$_A$ communicates with caCD$_B$ in order to start the execution of a variant; otherwise, caCD$_A$ starts the execution of the first task of the selected variant.

In our reference scenario, the selection of a variant (Figure 3.5) has to be performed for the variation point *Payment*. The caCD of *Mobile App* asks the Context Manager for the variant selection after the execution of *Start Payment Process*. *Variant A* is selected if the payment service is available, *Variant B* otherwise. Since for both of them the initiating participant is *Poster Service*, in both cases, the caCD of *Mobile App* informs the caCD of *Poster Service* about which variant has to be executed. Then, if the selected variant is *Variant A*, the

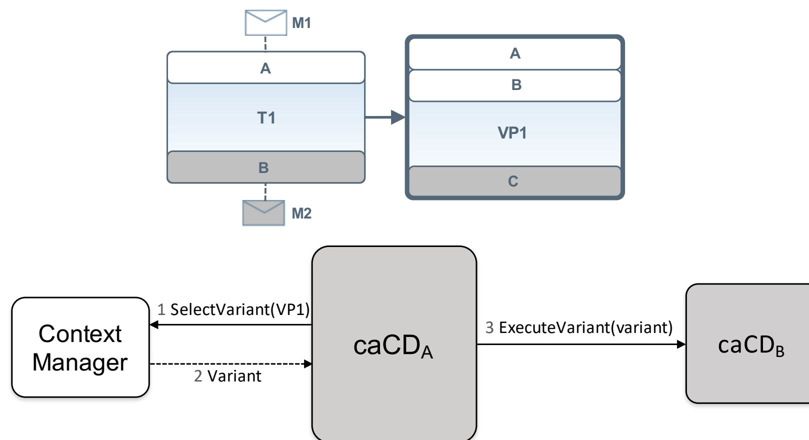


Figure 3.12: Context-aware CD interaction pattern for variant selection

caCD starts the task *Send Payment*; *Reply With Invoice*, otherwise.

3.4.3 Context Manager

The Context Manager holds the model instance of the context of both system and entities, receives the context-carrying messages that are exchanged during the choreography execution, and executes the acquisition and selection functions. In the following, we describe how: (i) the context is represented in the Context Manager; (ii) the context acquisition is performed; (iii) the entities are selected at runtime.

Context model instance

The synthesis process transforms the context model into a set of Java classes, which are instantiated at runtime. The Java classes contain, as class attributes, the context attributes that have been defined in the model. At runtime, there will be:

- An instance of the system context class;
- An instance of an entity context class for each adaptable entity (i.e., an "entity context" instance for each instance of a participant service and for each choreography variant).

The list of service instances is obtained by querying the Service Registry. Moreover, when a new service is published into the registry, the latter will interact with the Context Manager in order to update the list and create a new instance of the entity class context.

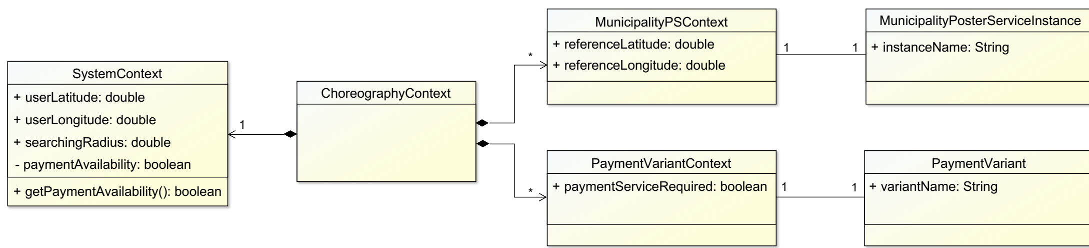


Figure 3.13: Class diagram of the context representation in the Context Manager

Figure 3.13 shows the class diagram that represents the context of our reference scenario. The context attributes defined in the system context model (shown in Figure 3.8) are translated into the attributes of the class `SystemContext`. The acquisition function *getPaymentAvailability* is translated into the method `getPaymentAvailability()`. The class `MunicipalityPSContext` contains the information of the homonymous entity class defined in the context model (Figure 3.8). The same holds for the `PaymantVariantContext` class. An instance of `SystemContext` will dynamically hold the runtime information of the system context, while there will be an instance of `MunicipalityPSContext` for each instance of Municipality Poster Service in the registry, and an instance of `PaymentVariantContext` for each of the two variants.

Context acquisition

As explained above, context information is acquired through context-carrying message or acquisition functions. Once obtained the context information, the Context Manager updates the values of the related attributes in the context model instance.

When a context-carrying message is sent, the involved caCD forwards it to the Context Manager. By means of an aptly defined *XPath* expression, the latter extracts the value of all the context attributes for which the received message has been defined as `ContextCarryingMessageSource`. In contrast, the values of the attributes that have an acquisition function as a context source are obtained on demand, when they are used for the entity selection.

Coming back to our scenario, the Context Manager receives the context-carrying message *postRequest* that is used as a context source for the attributes *userLatitude*, *userLongitude*, and *searchingRadius*. The Context Manager extracts the value from the message and updates the values of the *userLatitude*, *userLongitude*, and *searchingRadius* attributes in the instance of the class *SystemContext*. Instead, the value of the context attribute *paymentAvailability* is

obtained by executing its defined acquisition function. This is done before the execution of the selection function of the variants.

Entities selection

When a choreography variant needs to be executed or a context-aware participant instantiation needs to be performed, the Context Manager selects the adaptable entities by executing the selection function. When selecting a variant, the Context Manager simply returns to the involved caCD the name of the selected variant. When dealing with context-aware participant instantiation, once selected the service instances through the selection function, the Context Manager obtains their description (i.e., the service endpoints) from the service registry. Then, the Context Manager returns the list of the endpoints of the selected instances, together with their runtime context.

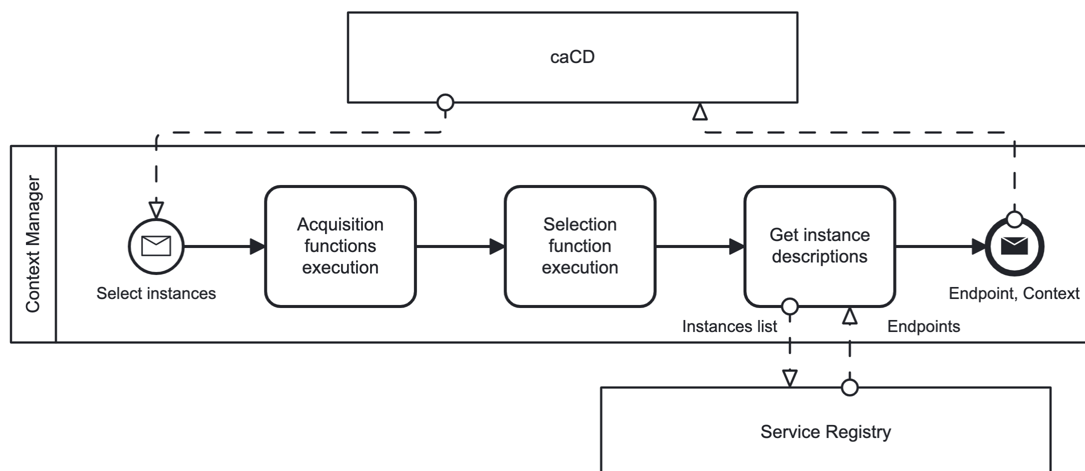


Figure 3.14: Context Manager behavior for the context-aware participant instantiaton

Figure 3.14 shows, through a BPMN diagram, the behavior of the Context Manager for a general case of context-aware participant instantiation. When a caCD needs to start the execution of a task requiring context-aware participant instantiation, it asks for the selection of service instances to the Context Manager (the interaction pattern of the caCD is depicted in Figure 3.11). After receiving the message, the Context Manager executes the acquisition functions related to (i) the adaptable entities to be selected and (ii) the system context attributes that have to be evaluated. The results of the acquisition functions are stored into the context model instance. Then, the Context Manager executes the selection function, obtaining the list of the selected service instances. Next, it asks the service registry for the endpoints of the selected instances and returns them to

the caCD together with their associated context. For the sake of simplicity, we avoid reporting the behavior for the variant selection: the behavior of the Context Manager is the same, except for the communication with the Service Registry, which is avoided since the Context Manager selects and returns to the caCD the name of the variant to be executed.

Note that both the context evaluation and the entity selection are performed before that participants are involved in the interaction. Thus, service instances are selected only when they are "inactive" (i.e., only when it is not in the middle of serving a request), hence their substitution cannot affect the consistency of the overall choreography state. In order to avoid inconsistencies, we assume that:

- the service instances to adapt are stateless, i.e., there is no internal state to be restored; rather, there can be a conversational state that is maintained across every single invocation through message passing and/or "external" session handling;

- the context conditions that are evaluated for the selection cannot change between an instance selection and the next one (e.g., the context attributes are acquired from context-carrying messages that are sent only once, as in our reference scenario).

Moreover, as stated in Section 3.1, we also assume that all the service instances of a multi-instance participant share the same interface and the same interaction protocol, thus we do not focus on possible interface incompatibilities.

Concerning our scenario, the entity Context Manager performs the context-aware participant instantiation when the instances of *Municipality Poster Service* have to be dynamically selected. Since there are no acquisition functions to be executed in this particular case, the Context Manager just executes the selection function `selectMunicipalityService-Instances`. Then, it gets the endpoints of the selected service instances from the Service Registry and returns them to *caCD Payment Service*, which is in charge of performing their invocation. In order to perform the adaptation needed for the variation point *Payment*, the Context Manager first executes the acquisition function `getPaymentAvailability` to get the current value of the *paymentAvailability* context property, then it executes the selection function `selectPaymentVariant`. The name of the selected variant is then returned to *caCD Poster Service*, which will execute the selected variant.

3.5 Synthesis and refinement processes

This section describes how the synthesis process generates the components described in the previous sections and how they are refined in the refinement process. Figure 3.15 overviews the synthesis process. It is an extension of the process described in Section 2.2, now including the generation of the software artifacts described in the previous Section.

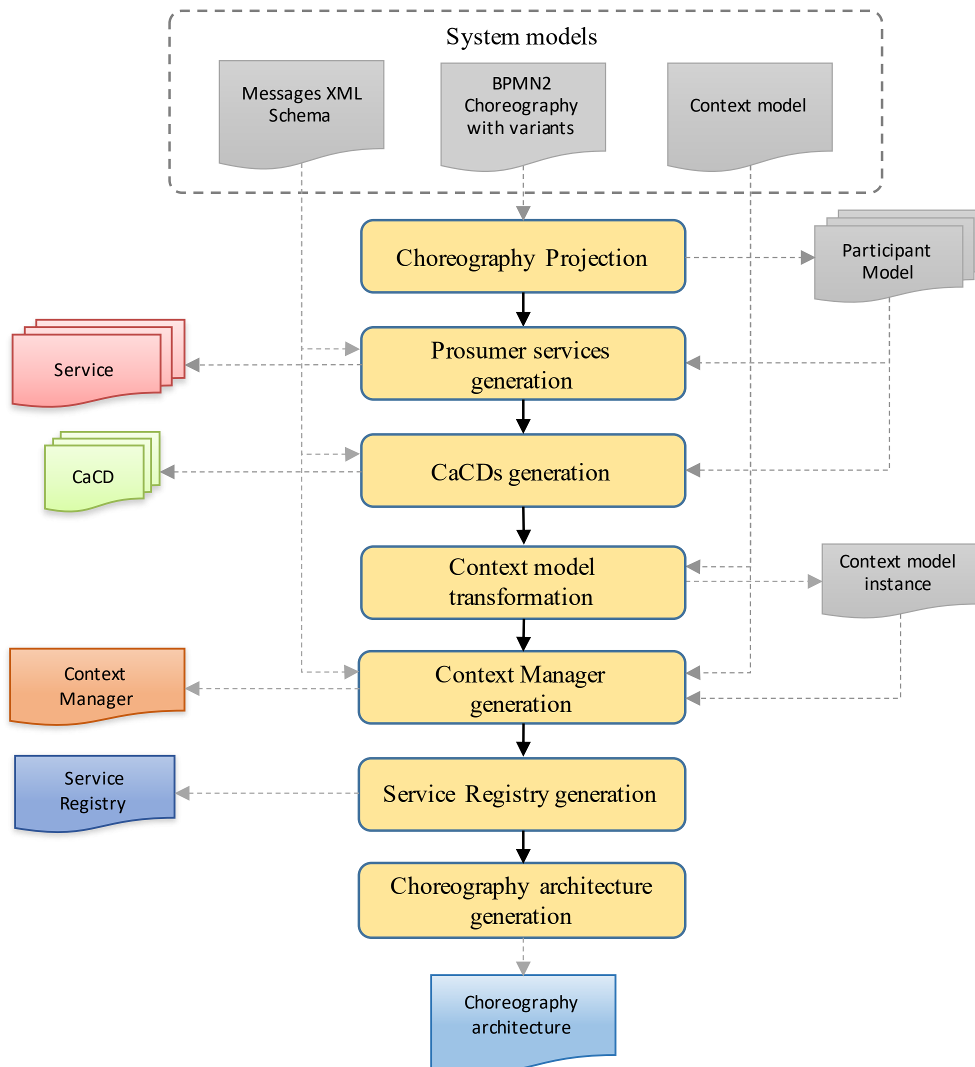


Figure 3.15: Synthesis process overview

The process is performed by a *synthesis processor*, which takes as input the choreography specification (Figure 3.4) including the specification of the variants (Figure 3.5), the context model (Figure 3.8), and the XML schema of the messages (Figure 3.9).

The synthesis process starts with the *Choreography Projection*. As explained

in Section 2.2, it is a model-to-model transformation where the BPMN2 choreography specification is used to create a participant model for each consumer or prosumer service involved in the choreography. The participant model, still in BPMN2 format, describes the sequence of message exchanges performed by a participant. This model serves as input for the subsequent steps: prosumer service generation and caCDs generation.

In the prosumer service generation and caCDs generation steps, prosumer services and caCDs are generated for each participant whose model was obtained in the previous step. This step utilizes the participant model and the XML schema definition of the business messages exchanged with other participants. It is important to note that only the skeleton code for the prosumer services is generated, as the complete implementation of their business logic is application-specific and requires manual completion. In particular, as described in Section 2.2, the provider-side business logic of prosumers can be either implemented from scratch or reused from an existing service, while the consumer-side business logic needs to be implemented.

The next phase is the *context model transformation*. Here, given the context model as input, a model-to-code transformation outputs the code of the Java classes representing the context model instance, as described in Section 3.4.3. The context model instance serves as input for the next Context Manager generation step.

The Context Manager is generated by taking as input the context model, the Java classes of the model instance, and the XML schema of the messages. These inputs are needed to generate the skeleton of the acquisition and selection function, the code required to hold the runtime context data, and the management of context-carrying messages, respectively. The Context Manager, as it is generated in this phase, requires further refinement through the implementation of the selection and acquisition functions.

Next, the Service Registry is generated. The automated synthesis ends with the generation of the architectural representation of the choreography, which highlights the interdependencies among all the generated software artifacts and the other third-party participants.

Figure 3.16 shows choreography architecture resulting from the synthesis phase applied to the reference scenario. Besides the Context Manager and the Service Registry, the synthesis processor generates the following artifacts:

- *caCD MobileAPP*, associated to the participant Mobile APP.

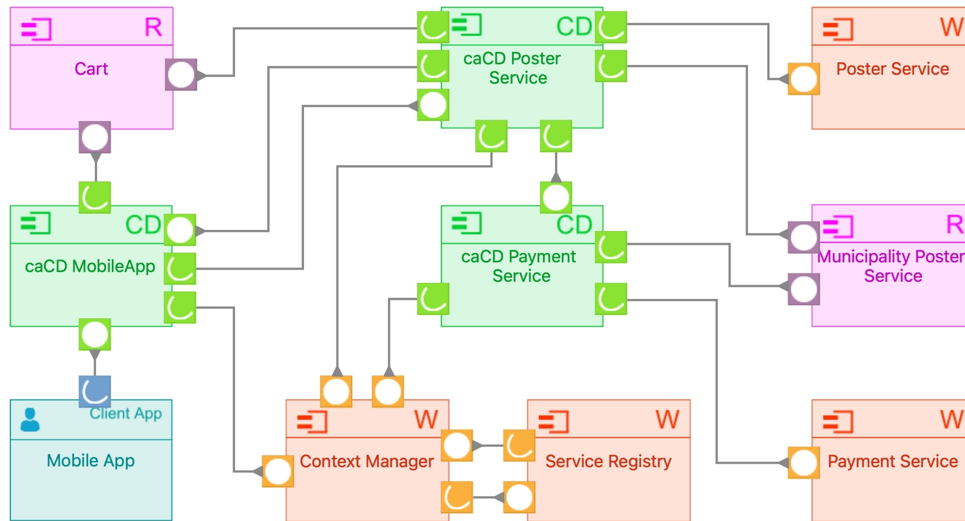


Figure 3.16: Generated system architecture

- *caCD Poster Service*, associated to the participant Poster Service.
- *caCD Payment Service*, associated to the participant Payment Service.
- The skeleton code of the *prosumer* services *Poster Service* and *Payment Service*. The former contains the logic needed to interact with the Mobile App and handle the requests for posting spaces. It needs to be implemented by following the skeleton code that has been generated. The latter can be realized by reusing an existing payment service for its provider-side business logic and implementing the consumer-side business logic only.

After the synthesis process, in the *refinement* phase, the implementation of prosumer services logic and the selection and acquisition functions of the Context Manager are completed. Figure 3.17 overviews this phase. Differently from the synthesis, which is automatically performed, the refinement needs the manual intervention of developers to fill the skeleton code of prosumer services and the Context Manager. Concerning prosumer services, the code implementing the *message construction logic* and the *provider side business logic* (Section 2.2) need to be added to the already generated code. Concerning the Context Manager, developers provide the implementation of the selection and acquisition functions required to perform adaptation. After the refinement process, the choreography is ready to be deployed and enacted.

With reference to our use case, the implementation of the prosumer services is refined by providing the provider-side and the message construction logic of *Poster Service* and *Payment Service* prosumers as explained above. Concerning

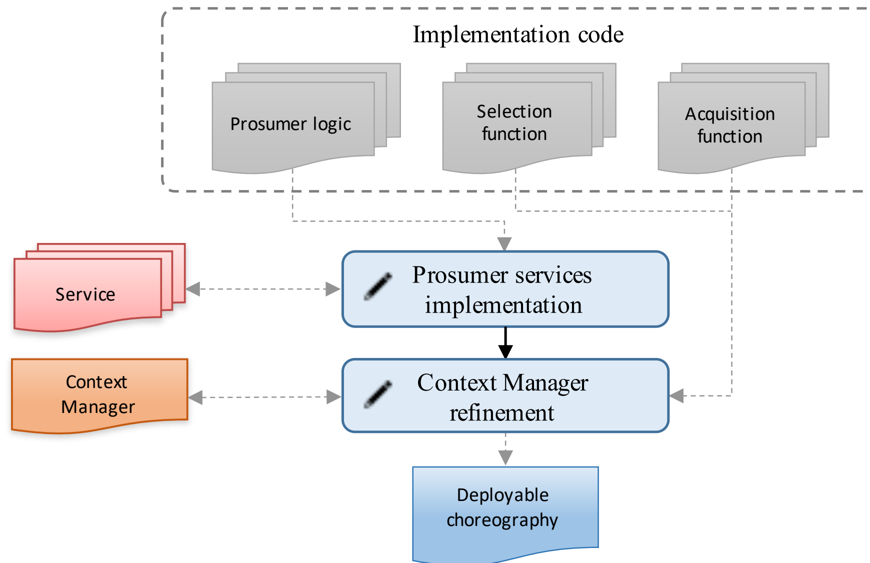


Figure 3.17: Refinement process overview

the Context Manager, the synthesis process outputted the method declarations for two selection functions (`selectMunicipalities` and `selectPaymentVariant`) and an acquisition function (`getPaymentAvailability`), as defined in the context model in Figure 3.8.

Figure 3.18 shows the implementation of the two selection functions. The `selectMunicipalities` method takes as parameters the system context attributes that have been defined in the context model (*userLatitude*, *userLongitude*, *searchingRadius*), the list of candidate adaptable entities (i.e., the list of *MunicipalityPosterServiceInstance*), and returns the list of selected ones. In particular, it returns the list of municipalities (adaptable entities) whose distance from the user's position is less than *searchingRadius*. The distance between the municipality location and the user position is obtained by using the `distance()` method. It has been aptly coded and computes the cartesian distance between the user coordinates (*userLatitude* and *userLongitude*) and the reference coordinates of the service, which are obtained from their context, as defined in the context model and shown in the class diagram in Figure 3.13. For the sake of simplicity, the proposed implementation of this function leverages the assumption that, in our particular scenario, at least one instance of Municipality Poster Service is available in the searching area defined by the user through his/her position and the selected searching radius. However, in general, as explained in Section 3.4.1, the implementation of the selection function should always provide at least a default choice: this can be achieved by improving the proposed selection function by returning, e.g., the closest municipality whenever there are no municipalities in

the searching area.

Concerning the implementation of the `selectPaymentVariant` method, it returns the variant that requires the Payment Service (if the latter is available); the one that does not require the service, otherwise.

```
public List<MunicipalityPosterServiceInstance> selectMunicipalities(
      double userLatitude,
      double userLongitude,
      double searchingRadius,
      List<MunicipalityPosterServiceInstance> instances) {

  return instances.stream().filter(e ->
      distance(e.getContext().referenceLatitude, e.getContext().referenceLongitude, userLatitude, userLongitude) <= searchingRadius
  ).collect(Collectors.toList());
}


public PaymentVariant selectPaymentVariant(boolean paymentAvailability, List<PaymentVariant> variants) {

  return variants.stream().filter(e -> e.getContext().paymentServiceRequired == paymentAvailability).findFirst().get();
}
```

Figure 3.18: Selection functions implementation

The acquisition function `getPaymentAvailability` has been implemented to simulate the interaction with the service and return test values.

3.6 Evaluation

In this section, we describe how the system obtained from the previous section has been set up in order to run experiments and we discuss the obtained results. The experimentation purpose is to evaluate that: (i) the choreography execution is correctly enforced by the caCD, i.e., the participant services interact as prescribed by the choreography specification; (ii) caCD are able to achieve dynamic adaptation of the system with respect to the context; (iii) the distributed coordination and adaptation layer of the system does not have any significant impact on the choreography execution time, even when the number of concurrent requests or the number of adaptable entities grows, i.e., the overhead introduced by the Context Manager and caCDs for the adaptation is negligible.

The complete implementation of the system used for evaluation and the related documentation is publicly available for replicating the performed experiments[1]. In the following, we describe the experimental setting. Then, we define the metrics computed with the collected data. Finally, we report the obtained results.

[1]https://github.com/sosygroup/connectpa-billposting-choreography

3.6.1 Experimentation settings

For experimentation purposes, we deployed five different instances of *Municipality Poster Service*, each referring to a different municipality. Each instance holds information about the posting spaces available in its territory. Figure 3.19 shows the geographical distribution of the five municipalities that have been considered.

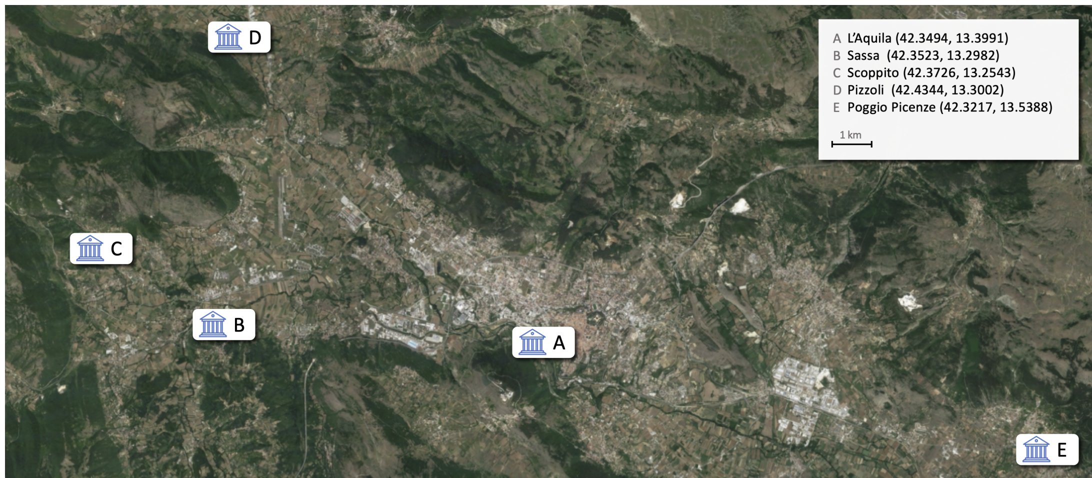


Figure 3.19: Geographical distribution of the municipalities involved in the choreography

The information about the instances of the Municipality Poster Service has been put into the Service Registry through its API. When the choreography is enacted, the Context Manager initializes the context of each of the instances with information about the location of the municipalities.

In order to perform tests, the *Mobile App* is realized as an application that simulates the presence of a varying number of users that concurrently interact with the system. For each simulated user, the Mobile app sends a request by selecting randomly the current user's position and the searching radius. Then, in order to simulate its autonomous behavior, the Mobile App attempts to perform an interaction 1.5 seconds after the end of the previous one.

The sequence diagram in Figure 3.20 details the timing of the requests. For the sake of simplicity, the diagram portrays only the interactions that occur between the *Mobile App*, *Poster Service*, and *Cart*, without detailing the interactions that occur between the other composed services. For each simulated user, the Mobile App invokes the *Add Space* operation 1.5 seconds after that the *Reply* message is received from *Poster Service*, then after 1.5 seconds it invokes *Send Confirmation*, and so on, thus simulating the reasoning time of the human user.

The experimentation has been performed using six Virtual Machines (VMs)

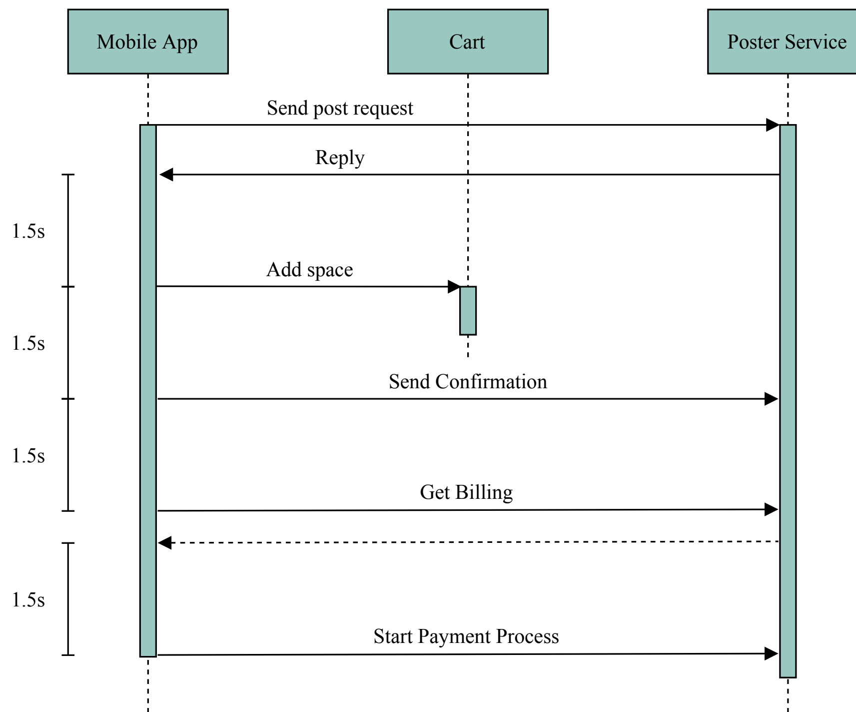


Figure 3.20: Sequence diagram illustrating timing of the simulated *Mobile App* interactions

installed in three distinct Server Machines (SMs). Each SM is equipped with 2 CPUs Intel Xeon E5-2650 v3, 2.3 GHz, 64 GB RAM, and 1 Gb/s LAN network; each VM has 4 CPU cores and 4 GB of RAM. The operating system is Ubuntu Server 20.10. Open Stack is the cloud infrastructure provider. Participant services, CDs, and Context Manager have been deployed on the VMs as described in Figure 3.21.

This deployment setting permitted us to experiment with a real scenario, where services are offered by different providers, and also the instances of *Municipality Poster Service* are geographically distributed, hence running simultaneously with real parallelism. The same holds for caCDs, Context Manager, and Service Registry.

3.6.2 Data collection and metrics

Experiment data are collected by caCDs, Context Manager, and participant services, which locally log the start and end timestamps of each task. They are obtained only after that the execution is completed, in a way that the collection of data does not interfere with the message exchange between participants and CDs. Figure 3.22 shows the interaction pattern of services and CDs for the most

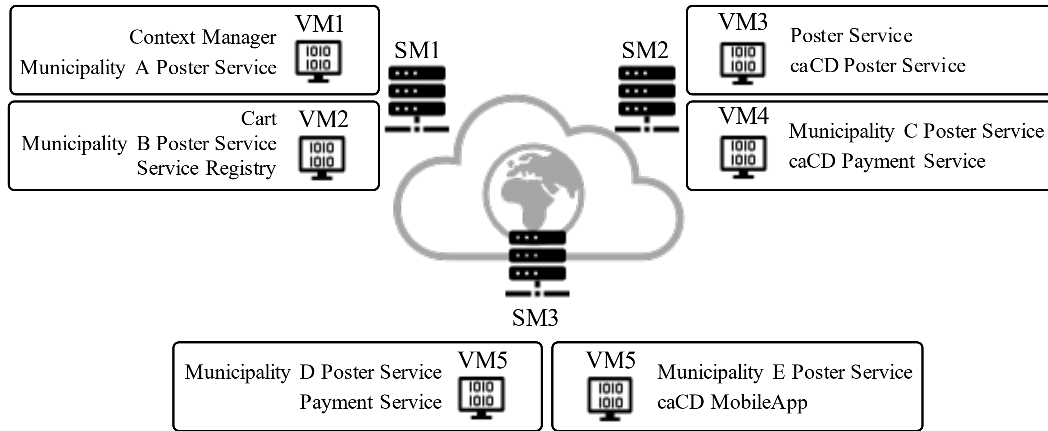


Figure 3.21: Services deployment scheme for the experimentation

general case as defined in Section 2.8 and the timestamps logged by the services. Pure consumer services log the timestamp when sending the request message to the receiving participant ($start(T1)$ in the figure, being $T1$ the name of the task for which the log has been generated), and when receiving the response message, if any ($end_c(T1)$). Pure provider services log the timestamps when receiving the request messages and when replying with a response message, if any ($rec(T2)$ and $end_p(T2)$ in the example), or when their computation is complete, otherwise. Prosumer services log the timestamps when receiving the request messages (and when replying) if they play the role of provider ($rec(T1)$ and $end_p(T1)$). If they play the role of consumer, they log the timestamp when receiving the request from the CD ($start(T2)$), and when receiving the response message, if any ($end_c(T2)$). The Context Manager logs the timestamps when receiving the request for the selection of an adaptable entity and when replying back with the list of selected ones (interactions 1 and 2 in Figures 3.11 and 3.12), respectively defined as $sel_start(T_n)$ and $sel_end(T_n)$ where T_n is the name of the activity (task or Variation Point) requiring adaptation. CaCDs log the timestamp when sending and receiving SYNCH messages.

By using the logged timestamps defined above, we measured the following metrics:

- *Task Execution Time*: time between the first and the last logged timestamps. It is computed for each task as the difference between the timestamp of the request received by the initiating prosumer and the timestamp of (i) the received response (if the task is request-response) or (ii) the received request message by the receiving participant (otherwise):

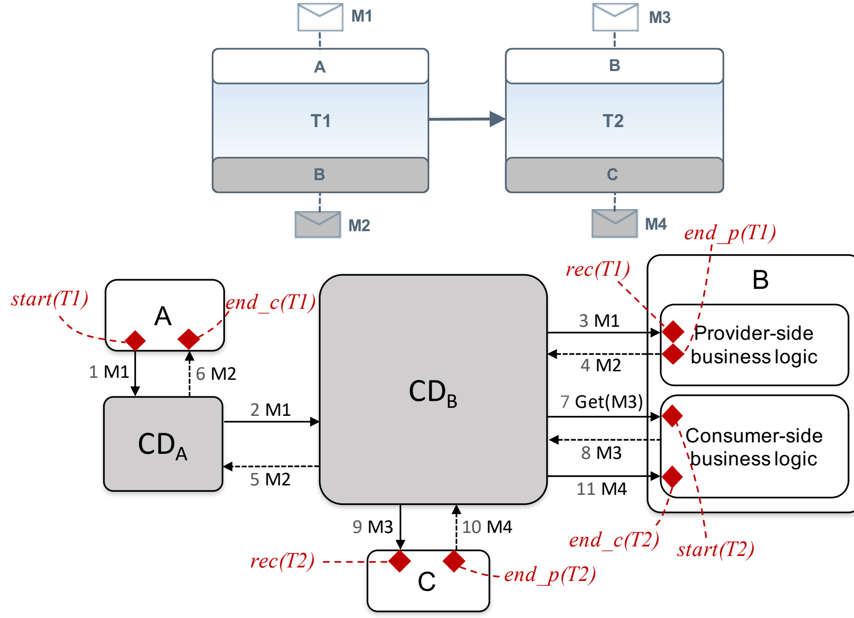


Figure 3.22: Timestamps logged by participant services

$$ExecTime(T_n) = \begin{cases} start(T_n) - end_c(T_n) \;\; if\; T_n \;\; has\;\; a\;\; response\;\; message \\ start(T_n) - end_p(T_n) \;\; otherwise \end{cases} \tag{3.1}$$

where T_n is the name of the task;

- *Coordination Overhead*: time between the end of a task and the start of the following task, excluding the tasks that are initiated by full-consumer services (i.e., the Mobile App in our use-case) or that require adaptation. It is computed as the difference between the timestamp for the end of the previous task and the timestamp of the request received by the initiating prosumer:

$$Coord(T_n) = start(T_n) - max\{end_c(T_{n-1}), end_p(T_{n-1})\} \tag{3.2}$$

where T_n is the name of the task and T_{n-1} is the name of the previous one;

- *Adaptation Overhead*: time between the receiving of the request for the entity selection by the Context Manager and the start of the activity requiring adaptation:

$$Adapt(T_n) = start(T_n) - sel_start(T_n) \tag{3.3}$$

where T_n is the name of the activity that requires adaptation;

- *Entity Selection Time*: time needed by the Context Manager to select the adaptable entities and return the selection to the caCDs. It is computed as the difference between the two timestamps logged by the Context Manager:

$$Sel(T_n) = sel_end(T_n) - sel_start(T_n) \tag{3.4}$$

where T_n is the name of the activity that requires adaptation. Note that this measure is already considered by the Adaptation overhead, but if observed individually it can give us information about the specific overhead introduced by Context Manager for the entities selection, in particular when considering sets of selectable service instances of different sizes.

Data are collected by running the system with an increasing level of concurrency through the simulation of a varying number of interacting users. In particular, we run the test multiple times by simulating, at each run, a different number of concurrent users from 1 up to 500. Moreover, we also simulate the presence of a growing number of instances of Municipality Poster Service by replicating the entries in the Service Registry. All the tests are also run by simulating the availability or unavailability of the Payment Service. These settings stress both the Context Manager in the selection process and the caCDs in the runtime participant instantiation. In each of the described settings, the choreography is run 20 times in order to reduce the impact of outliers and random performance variations, hence improving the significance and consistency of the collected data. Also, this allows us to compute the average value for all the metrics defined above.

3.6.3 Experiment results

This section reports the results obtained by running the experiments as set in the previous section. Concerning our use case, experimental results show that: (i) the choreography execution is enforced in the correct way; (ii) the context-awareness capabilities are successfully realized; (iii) the overhead introduced by caCDs and Context Manager is negligible even with a large number of concurrent users and selected service instances, and its growth is linear with respect to the increase of both the number of users and the number of selected instances.

Choreography execution enforcement

Figure 3.23 shows a timeline of the average execution times of the tasks when a single user is simulated. For the sake of simplicity, we report the tasks that are executed when Variant A (Figure 3.5) is selected. For each task, the blue bar represents the time during which the task is executed (i.e., the time between the first and the last logged timestamps of the involved services). If the task belongs to an independent sequence, a striped bar shows the time between the sending of the message by the initiating participant to its associated CD (interaction 1 in Figure 2.10) and the forwarding of the message to the receiving participant (interaction 2). For the tasks with multi-instance participants (marked with a "*" in Figure 3.23) the blue bar shows the average execution times for all the involved instances of Municipality Service.

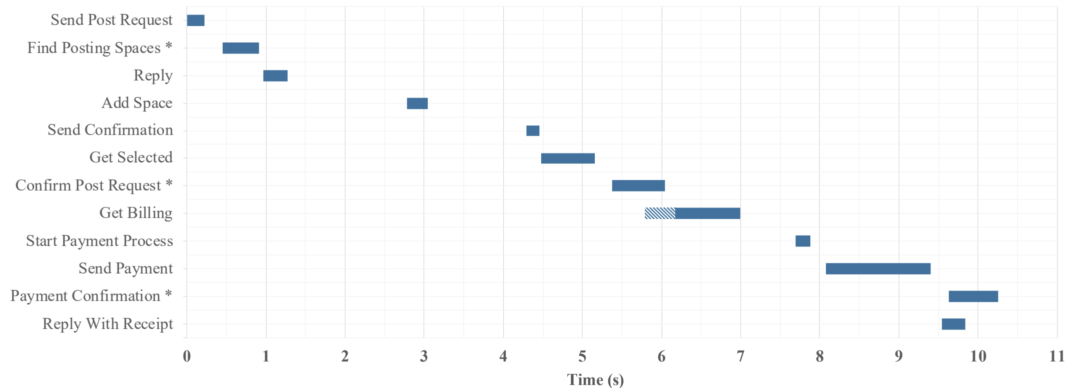


Figure 3.23: Average execution time of choreography tasks

Tasks are executed by fully respecting the choreography specification. In fact, although Mobile App can try to invoke tasks in a totally autonomous way, their execution is enforced to follow the specification. Figure 3.24 shows the details of the execution of the independent sequences of the choreography. The rhombus represents the sending time of the SYNCH message by the caCD of Poster Service to the caCD of Mobile App, for enforcing the sequence *Confirm Post Request* → *Get Billing*. We can note that, since Mobile App tries to execute autonomously the task *Get Billing* while *Confirm Post Order* is still being executed, the message is not forwarded until the latter is completed and the SYNCH message is sent (striped portion of the bar). Only after that the SYNCH message is received, the caCD MobileApp forwards the request message and *Get Billing* is effectively executed (filled portion of the bar for *Get Billing*).

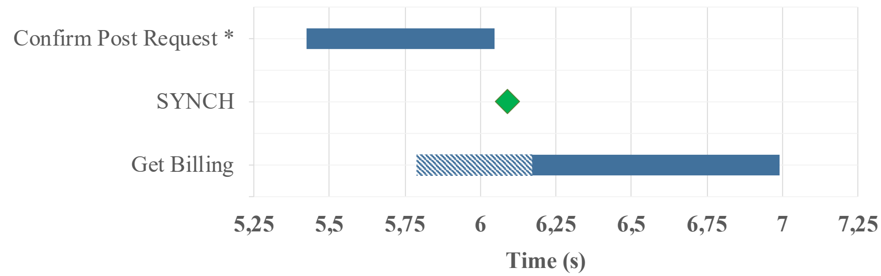


Figure 3.24: Details of the independent sequence execution

Impact on system performances

Table 3.1 reports the results of our experiments. For each experimental setting, we report the average, minimum, and maximum values for the total task execution time, total coordination overhead, total adaptation overhead, and total entity execution time of each choreography run. The first six runs have been performed by increasing the level of concurrency by simulating a growing number of users, while the number of service instances in the Service Registry is fixed to the five instances that we deployed. In contrast, in the last three runs, we fixed the number of simulated users and we considered a growing number of service instances in the service registry.

With a single user and 5 service instances, we measured an average total adaptation overhead of 563ms. Of these, 476ms are needed for the evaluation of the context and the selection of the adaptable entities, while the remaining 87ms are spent for the execution of the adaptation performed by the caCDs. If considering the execution of a single adaptation, it requires an average of 141ms, of which 119ms are needed for the entity selection. As a comparison, we measured that the task *Send Confirmation* lasts 164ms, while the average execution time of a choreography task is 501ms. The same can be said also with a higher degree of parallelism: with 500 instances, we measured a total overhead for the adaptation of 880ms (220ms for a single selection), while the average total time for the task execution is 9200ms.

Figure 3.25 shows how the average times reported in Table 3.1 increase when the degree of parallelism increases. We observe that, when the number of concurrent requests grows, the mean total overhead for the adaptation has a decreasing weight with respect to the mean total execution time of the choreography tasks. On average, without concurrent users, the total time spent for the selection of the adaptable entities measures the 9.37% of the total task execution time; with 200 concurrent instances, it measures the 8.9%; with 500 concurrent instances,

Testing conditions	**Tasks Execution Time**			**Coordination Overhead**			**Adaptation Overhead**			**Entity Selection Time**		
	Avg	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**
1 user	6009	5602	6311	86.4	81	92	563	478	611	476	417	559
10 users	6022	5693	6304	86.7	82	91	568	484	609	474	425	541
50 users	6206	5923	6898	87.9	82	96	577	520	628	495	440	520
100 user	6704	6182	7348	92.1	88	99	599	521	673	501	462	546
200 users	7642	7002	8141	99.9	93	115	681	600	775	552	497	613
500 users	9198	8652	10036	113.8	95	149	802	732	904	671	585	729
1 user, 15 service instances	6042	5599	6284	86.5	81	90	573	481	623	478	409	510
1 user, 25 service instances	6055	5626	6341	87.5	83	92	585	502	668	481	413	543
1 user, 50 service instances	6078	5701	6400	88.3	83	92	597	511	695	488	426	544

Table 3.1: Experiment results (data in ms)

the 8.7%. For what concerns the overhead for the coordination, we note that this overhead is even more negligible, since it represents only the 1,4% of the total task execution time without any concurrent instances, while it lowers to the 1,3% with 500 instances running concurrently.

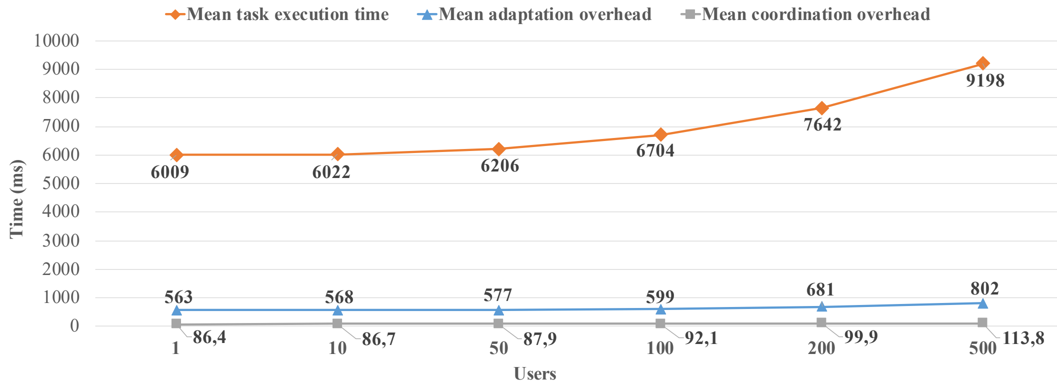


Figure 3.25: Trend of task execution time, adaptation overhead, and coordination overhead at the increasing number of users

A linear regression analysis performed using the mean values of the overhead shows that both adaptation and coordination overhead grow linearly with respect to the number of concurrent users. In particular, concerning the adaptation overhead, we observe that the value of the slope of the line is low (0.4933) – meaning that the growth is very smooth – and the coefficient of determination r^2 is very close to 1 (0.9843) – meaning that the linear growth provides a good approximation. The same holds for the coordination overhead: here we observe that the slope is very low (0.0564); whereas, r^2 is 0.9872. The results concerning the coordination overhead support the findings that have been obtained in previous work concerning the enforcement of the correct choreography coordination [8], thus confirming the high scalability of the coordinated system when handling a high number of concurrent requests.

Finally, we observe how the adaptation overhead changes when the number of service instances that are selected increases. Figure 3.26 shows how the adaptation overhead and the entity selection time grow when the number of selected instances for the tasks with multi-instance participant increases. The linear regression highlights that both the adaptation overhead and the selection time have linear growth. Interestingly, we can note that the latter grows with a lower slope than the whole adaptation overhead (0.2715 against 0.7741). This hints at the fact that the selection process is only marginally affected by the number of instances that can be selected. In contrast, the only instantiation process takes

more time to be completed. In fact, by referring to Figure 3.11, we observe that, while the selection function is executed only once regardless of the number of service instances in the registry (interactions 1 and 2), the caCD asks the request message to the initiating participant of the task for each selected instance (interaction 3). This results in a higher number of interactions that are executed in parallel and that (slightly) increase the overhead. On the other hand, we can observe that the constant part of the overhead – represented by the y-intercept of the line – is mainly related to the selection time (474.3ms, out of 561.83ms of the whole adaptation constant part). This is due to the fact that, as explained, the selection process is always executed once even if no suitable service instances are found and selected: the context acquisition, the query on the service registry, and the execution of the selection function represent computational steps that are always needed and executed only once, when adaptation is required.

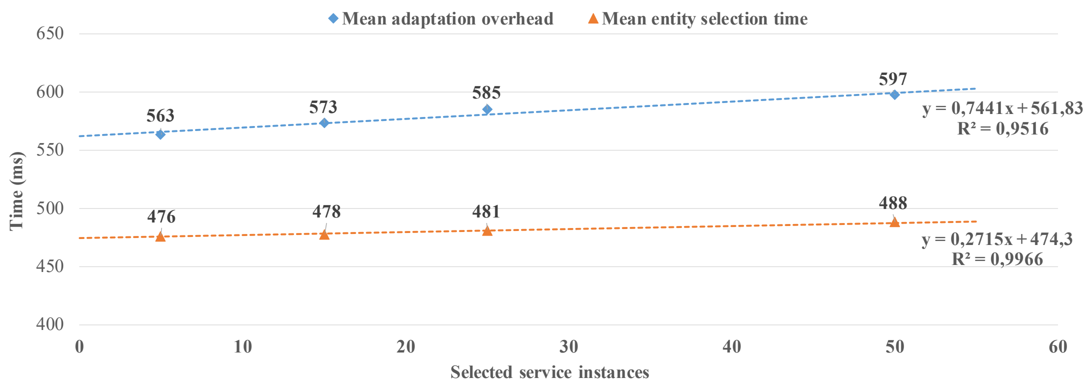


Figure 3.26: Trend of adaptation overhead and entities selection time at the increasing number of service instances

3.7 Discussion

This Chapter described the proposed approach for the automated composition of services in context-aware choreographies. It is shown at work and evaluated on the use case implemented in the context of the ConnectPA project. This system be considered as a proof-of-concept of the proposed approach, showing its benefits and functionalities. In particular, it showed how local-grade services owned by public authorities can be composed in a wider, value-added system, benefitting from the decentralization coming from choreographies and the context-aware adaptation capabilities introduced with this work.

Results obtained from the experimentation show that, in the scope of our use case, the overhead introduced for the adaptation of the system is negligible, especially if compared with the execution time of the choreography tasks. Moreover, the scalability of the coordination and adaptation layer is satisfiable, since the time required for the adaptation grows linearly (with a very low rate) when the number of users and service instances grows. On the other hand, experimentation highlighted that the services realized and composed in the considered use-case suffer from the raise in workload when a high number of users concurrently interact with the system. This results in a degradation of the performance of the system. However, it is important to note that this issue does not concern our approach; rather, the scalability of composed services (and, as a consequence, of the system) can only be addressed by the institution or company that provides them.

In general, the scalability of such systems can be enhanced by leveraging the benefits brought by MSA, as explained in Section 2.3. This requires (i) the decomposition of the services that are more affected by heavy workloads to support better management of the resources, and (ii) load balancing to distribute the workload among the replicated instances of microservices. This calls for both an approach to the decomposition of monolithic services and an architecture supporting load balancing alongside the coordination needs of the system. In the next chapters, we will address these issues by presenting an automated approach to the decomposition of monoliths (Chapter 4) and an architectural style supporting both coordination and load balancing (Chapter 5).

Before going into these aspects, we now discuss some aspects that may threaten the validity of the proposed approach and experimental results.

3.7.1 Threats to validity

Internal Validity

The evaluation of the approach has been performed on a real use case related to the ConnectPA project. Although this allowed us to test the approach on a real scenario, this may limit the validity of the obtained results. In fact, in the use case, there has not been the need to realize a context acquisition function since the context information required for the adaptation features was only in the content of the business messages (we only defined context-carrying messages without the need for context acquisition functions). The execution acquisitions function may increase the adaptation overhead, in particular with a high number

of concurrent users or adaptable entities. On the other hand, experiments highlighted that the selection function (which is executed with the same frequency as adaptation functions, i.e., when adaptation needs to be performed) is very marginally affected by the number of users or adaptable entities. However, more tests are needed to have a more complete validation of the approach, in particular concerning the adaptation process on use cases with more stringent performance requirements. However, to overcome any possible loss in performance due to the execution of both selection and acquisition functions, they may be executed in an asynchronous way (when possible) or the Context Manager can be replicated in multiple instances to support scalability, just as suggested above when commenting on the scalability of the composed services.

External Validity

For the sake of simplicity, we presented the participant-level adaptation leveraging on the assumption that all the instances of a multi-instance participant service share the same interface and the same interaction protocol. Although this holds in our use case, often different service instances can offer the same functionalities while exposing different interfaces. This assumption allowed us to focus on context-aware adaptation rather than on other issues. However, in order to deal with interface incompatibilities, we can introduce service Adapters, already supported by the CHOReVOLUTION framework, as shown in previous works [4]. Adapters are software entities that are automatically synthesized from an ad-hoc realized adapter model that takes into account the different behavior and interfaces of services. Adapters are interposed between services and bridge the gap between their interfaces and their abstract description in the choreography model. When a service instance with an incompatible interface (described in the adapter model) is added to the Service Registry, an adapter can be synthesized to be used when the instance is selected at runtime.

The service selection is a dynamic adaptation mechanism that does not rely on pre-defined alternatives. Services can be added or removed at runtime from the service registry without requiring any reconfiguration or redeployment. In fact, the service selection function selects the services that are suitable according to the context (of both system and entities) without knowing a priori the set of candidate services instances. The instances are selected among those that are in the registry right when the selection is performed. Context acquisition functions can also provide information about the current service availability and/or their QoS levels to allow the adaptation also according to the dynamic conditions of

the participant services. In contrast, our task-level and task-flow level adaptation (i.e., choreography variants) relies on a set of well-defined alternatives that are defined since the early stages of the choreography modeling. The selection function selects among a set of known alternatives, by returning the name of the selected variant to the caCDs. It is clear that, in this case, adding a new variant at runtime requires a reconfiguration of the system. However, we need to only update the coordination protocol of the caCDs that handle the new variant, in order to suitably coordinate the participant services, and – if needed – the selection function in the Context Manager to consider the updated set of adaptable entities (i.e., variants among which the selection has to be performed).

Chapter 4

Monoliths decomposition

As explained in Section 2.3, the principles driving Microservice Architecture (MSA) support a series of characteristics that make this architectural style suitable for developing modern applications or re-engineering and modernizing existing monoliths systems. In particular, the refactoring process allows systems to gain flexibility, scalability, and ease of maintenance (among the other advantages), thus resolving some of the issues that may affect those systems. As a consequence, enterprises can take advantage of these benefits by migrating their systems to MSA. However, this process is complex, time-consuming, and error-prone [67, 114].

One of the main challenges that any monolithic-to-MSA strategy has to face is the identification of high-cohesive and loose-coupled microservices with the *right granularity* and within the *right bounded context* [43]. Service granularity can be defined as "the service size and the scope of functionality a service exposes" [62]. The problem in finding service granularity "is to identify a correct boundary (size) for each service in the system" [63]. As for the bounded context, the problem is to identify related functionalities and combine them into a single business capability, or responsibility, which is then implemented as a service [43]. This holds both for the migration of legacy and long-running monolithic systems, for "monolithic" services composing a SOA-based system, and for recently-built systems realized through a *monolith-first* approach [48, 93, 100, 107].

There exist different strategies to refactor a monolithic system into an MSA [49]. They can be classified into three categories: (i) top-down, forward-engineering strategies, in which high-level monolithic domain artifacts (e.g., use cases, activity diagrams, etc.) are accounted for and decomposed to model and implement the targeted microservices; (ii) bottom-up, monolithic re-engineering strategies, in which the dependencies of a monolithic system are analyzed to extract reusable

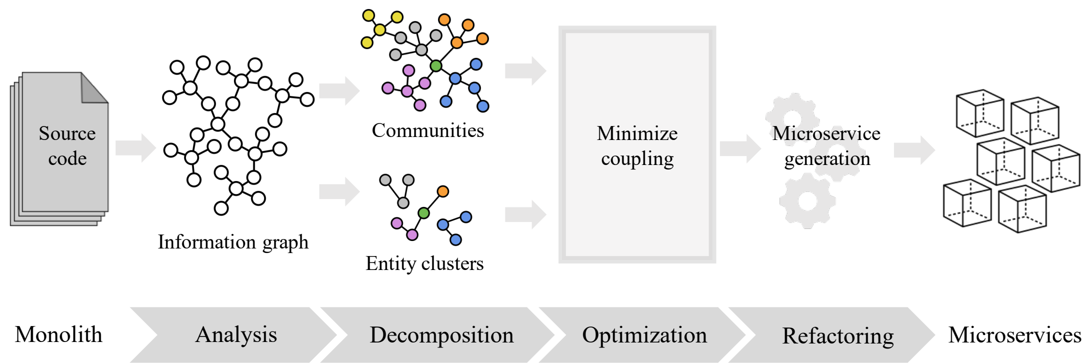


Figure 4.1: Overview of the decomposition approach

components, remove dependencies, and rewrite some existing applications by using the newly created microservices; and (iii) hybrid strategies, which suitably combine (i) and (ii).

With the aim of automatizing the whole migration process, without any specific input, system model, or knowledge, this chapter proposes a bottom-up automated decomposition approach. The approach covers all the phases of the migration (i.e., system analysis, architecture extraction, microservices implementation) in an automated way.

In the following, we describe in detail the phases of the proposed methodology and we report the results obtained by comparing with other decomposition approaches at the state-of-the-art on well-known and publicly-available monolithic systems.

4.1 Approach description

The methodology comprises four main phases (Figure 4.1):

1. *Analysis*: the source code is analyzed through static code analysis to produce an information graph representing the system at the method level;

2. *Decomposition*: the nodes of the information graph are clustered conveniently to find cohesive communities of both the complete graph and the subgraph of the domain entities only;

3. *Optimization*: an optimization model is built to distribute different communities into microservices in a loose-coupled way;

4. *Refactoring*: the source code of the microservices is generated by distributing methods and entities from the monolith to the identified microservices.

4.1.1 Analysis

In analyzing the system, we leverage the fact that most of the languages and frameworks used for building web services rely on layered architectures [106, 107, 117, 128]. In such architectures, methods in the presentation layer expose public interfaces and catch incoming requests. Then, methods of the lower layers are called, down to the persistence layer and the database [106]. Given this, we analyze the system and produce a graph representation that holds the relationships among components (methods and classes) by taking into account the role of each layer in the system. For the sake of generality, we abstract the exact number of system layers by considering only two main layers: a "logic" layer that comprises both presentation and business layers and a "persistence" layer.

The analyzer tool we have aptly realized parses the code to allocate the classes containing the system's business logic into each of the two layers and find classes that define the domain entities of the system (i.e., classes that represent a concept of the domain model, e.g, `User`, `Person`, `Pet`, etc.). In doing this, we leverage the annotations that are used in most of the frameworks to define the technical role of the classes (e.g., `@Controller`, `@Service`, `@Repository`, `@Entity` annotations provided by Spring Framework[1]) to identify the layer of each class or whether it represent a domain entity. The automated identification of such layers and entity classes can be manually refined to obtain a more accurate analysis.

After having allocated classes in the layers, the tool inspects the classes to find the declared methods and, going recursively deep into the code's syntax tree, collects the method calls and the references to the entities. Then, a node of the graph is created for: (i) each method from classes of the logic and repository layer, (ii) each entity class. Relationships between methods and entities are put in the graph as directed arcs. We define five types of arcs, each representing a particular relationship type:

- *Calls* arcs are added between method nodes when there is a method call from one method to another;
- *Uses* arcs are added between methods and entities if there is a reference to an entity in a method's code, e.g., if an object is instantiated or its values are accessed;
- *Persists* arcs are added between methods and entities if a method from the persistence layer reads or writes a given entity on the database;

[1] https://spring.io

- *References* arcs are added between entities if an entity references another entity, thus representing the association, aggregation, and composition relationships in the domain model;
- *Extends* arcs are added between entities if an entity extends another entity, thus representing the generalization relationship.

The output of this phase is an information graph, realized as a directed graph $G = (V, A)$ in which a type $t_i \in$ {Method, Entity} is assigned to each node $i \in V$, a relationship type $r_{ij} \in$ {Calls, Uses, Persists, References, Extends} is assigned to each arc $(i, j) \in A$. Moreover, a weight w_{ij} is assigned to each arc (i, j). Weights define the relevance of the relationships between nodes: the higher the weight of an arc, the more likely methods and/or entities at its endpoints should be put into the same microservice. After the analysis, default weights are assigned according to the type of the relationship. The default weight of the Persists relationship is 1, while those of the Calls, Uses, References, and Extends relationships are 0.8, 0.6, 0.2, and 0, respectively. In this way, we set the Persists relationship as the most important, and the method call relationship next to it. The reference and extends relationship weights are lower to favor domain entities to be distributed into different microservices. As for the system layers, weights can be manually refined to comply with specific application domains and requirements.

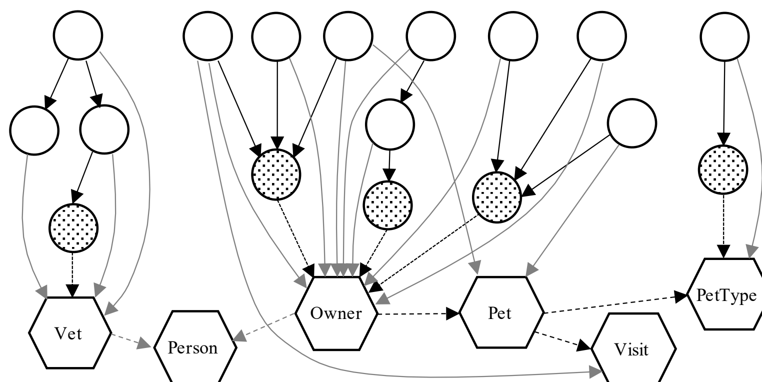


Figure 4.2: Information graph obtained for the Spring Petclinic project

Figure 4.2 shows a portion of the information graph obtained after the analysis of the project Spring Petclinic[2], which we will use as a reference use case throughout this Chapter. For the sake of simplicity, the figure only portrays a small portion of the complete graph. Circles represent methods: empty circles are the

[2]https://github.com/spring-projects/spring-petclinic

methods from the logic layer, while dot-filled ones are from the persistence layer. Hexagons represent entities. Black continuous arrows depict `Calls` relationships, while grey continuous arrows depict `Uses` relationships. Dashed black arrows outgoing from persistence-layer methods represent `Persists` relationships, while the ones between entities depict `References` relationships. Dashed grey arrows depict `Extends` relationship.

Such a graph can be seen as a reverse-engineered model of the whole system. The `Calls, Uses`, and `Persists` relationships represent all the interdependencies in the code, while the entity-to-entity relationships (`References, Extends`) reproduce the structure of the domain model of the system. This allows the identification of clusters of entities that possibly belong to the same bounded context and portions of the application in which methods are strongly related to each other and to a precise set of entities.

4.1.2 Decomposition

The problem of microservice extraction from a monolithic system naturally becomes a community detection problem when we represent it in the form of an information graph [82]. Newman describes the term community as: "a subgraph containing nodes which are more densely linked to each other than to the rest of the graph, or equivalently, a graph has a community structure if the number of links into any subgraph is higher than the number of links between those subgraphs" [92]. Therefore, selecting an appropriate community detection algorithm is very important to identify microservice candidates of high quality.

To identity community candidates within a large network, the Louvain community algorithm has some better features than its related algorithms [24] such as the GN algorithm [53, 92], the K-L algorithm [69], and the Spectral Bisection algorithm [15]. In fact, it has the lowest time complexity, higher cohesion, and stability [82]. To detect communities, the Louvain algorithm maximizes a modularity score, i.e., evaluating how much more densely connected the nodes are within a single community but fewer connections between different communities [91]. Equations (4.1) and (4.2) define the modularity heuristic function [92] that the Louvain algorithm optimizes to evaluate the partition of a community:

$$Q = \frac{1}{2m} \sum_{(i,j)} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta\left(c_i, c_j\right), \tag{4.1}$$

$$\delta(u,v) = \begin{cases} 1 \ when \ u = v \\ 0 \ when \ u \neq v \end{cases}, \tag{4.2}$$

where A_{ij} represents the weight on the edge connecting nodes i and j. For an unweighted graph, a constant value is set such as one. The k_i represents the sum of the weights of all edges connected to node i:

$$k_i = \sum_j A_{ij}, \tag{4.3}$$

where m is the sum of the weights of all edges in the graph, and c_i is the community to which node i currently belongs.

Being unsupervised, the Louvain algorithm does not require the number of communities to be found or the size of the communities [28].

By applying the Louvain algorithm to our representation of the system, we obtain a cohesive set of communities whose nodes are from all the system's layers. While this is good since communities consider relationships between nodes with different roles in the system (i.e., entities, controller interfaces, business logic, all required to offer complete functionalities) they cannot be generally taken as acceptable microservice candidates. In fact, since there are no constraints over the structure of the candidate microservices to be identified, communities may be built only of `Entity` type nodes, or – on the contrary – only of `Method` type nodes, hence, they may not realize complete and standalone microservices. Moreover, there are no assurances about the right granularity of the communities. Since there are no constraints over the number of microservices to be identified, they are likely to be too fine-grained, especially if obtained from graphs representing large systems with a high number of methods. Finally, given the above, communities may not succeed in wrapping all the nodes required to realize a business functionality, i.e., to build microservices around bounded contexts.

Therefore, to overcome these issues, we applied the Louvain community detection algorithm on the information graph in two different ways: (i) on the complete information graph, which overlaps all the architecture layers, to obtain cohesive sets of nodes; (ii) on the subgraph having only the entities related with the persistence layer, as it plays the most important role in maintaining the domain context [36, 78, 81, 108].

Figure 4.3 overviews the applied decomposition phase. The whole information graph obtained with the analysis phase is given as input to the Louvain Algorithm to find communities of nodes. Moreover, the subgraph of the domain entities

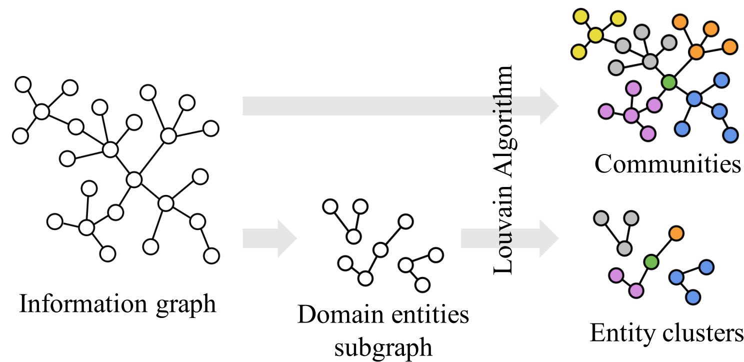


Figure 4.3: Overview of the decomposition phase

is obtained from the information graph and is given as input to the Louvain Algorithm to find communities of entity nodes only. For the sake of clarity, from here on, we will use the term *community* for the communities obtained from the whole graph, while we will use *entity cluster* for the communities obtained from the domain entities subgraph.

Graph Communities from the complete graph

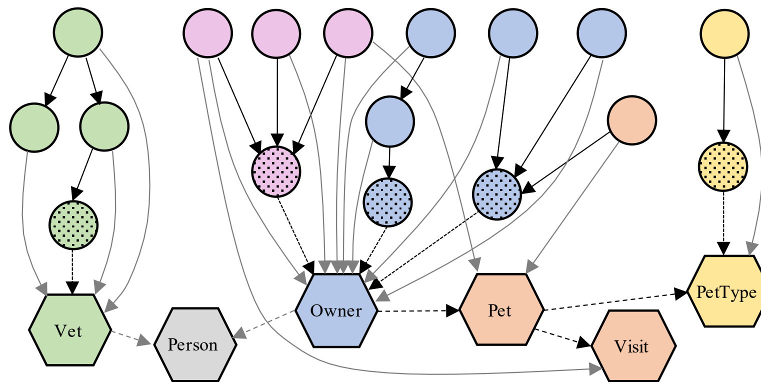


Figure 4.4: Communities obtained from the complete information graph

The execution of the Louvain algorithm on the complete information graph returns a set $\mathbb{C}$ of n communities $C_1, ..., C_n$ of nodes representing entities and methods spreading across all architectural layers. Figure 4.4 shows the communities obtained for the Spring Petclinic application: each node in the figure is color-coded to indicate the community it belongs to. As discussed above, these communities can not be considered as complete microservices since they are too fine-grained. For instance, methods related to the *Owner* entity are spread in three different communities. This means each of them may not contain the full

set of functionalities required to realize a business capability, i.e., they do not identify a bounded context. Moreover, while being cohesive (e.g., blue nodes are related to each other and all referencing the *Owner* entity) they would result in a high-coupled microservice architecture. However, we can leverage the cohesiveness of the communities to obtain cohesion in the final microservice architecture.

Entity Clusters

We obtain the sub-graph of the domain entities by considering the set of nodes of type `Entity` having relationships with the persistence layer and the arcs of type `References` and `Extends`. The Louvain algorithm on this graph returns a set $\mathbb{K}$ of m communities $K_1, ..., K_m$ of nodes of type `Entity` (entity clusters).

Figure 4.5 shows the entity clusters obtained for Spring Petclinic. We obtained two clusters. The first includes *Vet* and *Person* entities, and the second includes *Owner*, *Pet*, *Visit*, and *PetType* entities. These two clusters identify as many application contexts.

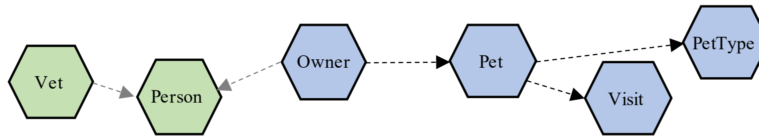


Figure 4.5: Communities obtained from the sub-graph of persisted entities

We use communities and entity clusters to, respectively, obtain cohesion and identify the granularity of the microservices in the optimization phase.

4.1.3 Optimization

The optimization of the microservice architecture is realized through an *Integer Linear Programming* (*ILP*) model that finds a solution to a variant of the Multiway Cut problem formulation [22] to which we added constraints over the structure of the microservices to be identified. The goal of the optimization is to partition the nodes of the information graph into a set of m microservices – where m is the number of entity clusters identified in the previous step – in such a way that the coupling among microservices is minimized. We define coupling as the sum of the weights of the arcs that connect microservices, as we will better describe in this section.

We state the problem as follows:

Given a graph $G = (V, E)$, a type $t_i \in$ {`Method`, `Entity`} for each node $i \in V$, a weight w_{ij} for each edge $(i, j) \in E$, **find** a partition of V in m sets $\{M_1, ..., M_m\} = \mathbb{M}$ such that:

(i) $\bigcap M_i = \emptyset$;

(ii) $\bigcup M_i = V$;

(iii) each $M_k \in \mathbb{M}$ contains at least one node i s.t. $t_i \neq$ `Entity`;

(iv) the sum of weights w_{ij} s.t. $i \in M_h, j \in M_k$, $h \neq k$ is minimized.

Each of the sets $M_1, ..., M_m$ is a candidate microservice.

The problem stated above has been modeled with an *ILP* formulation through the following decision variables, as in [22]:

$$x_{ik} = \begin{cases} 1 \text{ if node } i \text{ is in the microservice } M_k \\ 0 \text{ otherwise} \end{cases} \tag{4.4}$$

$$y_{ij} = \begin{cases} 1 \text{ if edge } (i, j) \text{ has its endpoints into the same MS} \\ 0 \text{ otherwise} \end{cases} \tag{4.5}$$

$$z_{ij}^k = \begin{cases} 1 \text{ if edge } (i, j) \text{ has both its endpoints into MS } M_k \\ 0 \text{ otherwise} \end{cases} \tag{4.6}$$

We define the following constraints:

$$z_{ij}^k - x_{ik} \leq 0 \ \ \forall M_k \in \mathbb{M} \ \forall (i, j) \in E \tag{4.7}$$

$$z_{ij}^k - x_{jk} \leq 0 \ \ \forall M_k \in \mathbb{M} \ \forall (i, j) \in E \tag{4.8}$$

$$x_{ik} + x_{jk} - z_{ij}^k \leq 1 \ \ \forall M_k \in \mathbb{M} \ \forall (i, j) \in E \tag{4.9}$$

$$y_{ij} = \sum_{M_k \in \mathbb{M}} z_{ij}^k \quad \forall (i, j) \in E \tag{4.10}$$

$$\sum_k x_{ik} = 1 \quad \forall i \in V \tag{4.11}$$

$$\sum_{i \in V | t_i = \text{Method}} x_{ik} \geq 1 \quad \forall M_k \in \mathbb{M} \tag{4.12}$$

Constraints (4.7) to (4.11) are from the Multiway Cut problem formulation[22],

while constraint (4.12) is a new one. We added it to ensure that methods are put inside the microservices since they are needed to expose interfaces, offer functionalities, realize microservice logic, etc.

The objective function (4.13) minimizes the *coupling* among microservices. As mentioned before, it is defined as the sum of the weights of the arcs whose endpoints belong to different microservices.

$$\min \sum_{(i,j)\in E} w_{ij}\,(1-y_{ij}) \tag{4.13}$$

However, microservices obtained from the solution of this optimization model, while having the lowest coupling, show a series of problems. In fact, the model does not lead to an actual domain-driven decomposition, since domain entities are free to be placed into any of the identified microservices. Hence, it may happen that all the entities are put into a single all-containing microservice, while other microservices may be small and built of only a few low-cohesive nodes, without a real meaning from both an architectural and functionalities point of view. To obtain a domain-driven decomposition and build microservices inside bounded contexts, we distribute entities into microservices according to the entity clusters identified in the previous phase (Section 4.1.2). Hence, we force each entity i in the entity cluster $K_k \in K$ to be in the microservice k. We realize this by fixing the x variables in (4.4) as follows:

$$x_{ik} = 1 \quad \forall i \in V \text{ iff } i \text{ is in the entity cluster } K_k. \tag{4.14}$$

In order to gain cohesion, we force all the nodes from the same community to be in the same microservice, so as to build microservices as a composition of the high-cohesive communities found in the previous phase (Section 4.1.2). Hence, we fix the y variables in (4.5) as follows:

$$y_{ij} = 1 \quad \forall (i,j) \in E \text{ iff } i \text{ and } j \text{ are in the same community.} \tag{4.15}$$

Notice that fixing x and y as in (4.14) and (4.15) can produce an infeasible model if two entities from the same community are in different entity clusters. To avoid this, we use a *Fix-and-Relax* approach. As shown in Algorithm 1, we first fix x variables according to the entity clusters in $\mathbb{K}$. Then, after fixing y variables for a given community $C_c \in \mathbb{C}$, we check the model feasibility. If infeasible, we relax the fixed y variables for all the nodes in the community C_c to keep the model feasible.

Algorithm 1 Fix-and-Relax algorithm for x and y variables

```
for entity cluster K_k ∈ 𝕂 do
    for i ∈ k do
        Fix x_ik = 1
    end for
end for
for community C_c ∈ ℂ do
    for (i, j) ∈ E do
        if i ∈ C_c & j ∈ C_c then
            Fix y_ij = 1
        end if
    end for
    if model is infeasible then
        for (i, j) ∈ E do
            if i ∈ C_c & j ∈ C_c then
                Relax y_ij
            end if
        end for
    end if
end for
```

Results obtained as a solution of this optimization model allow assigning nodes of the information graph (i.e., methods and entities) to microservices. Figure 4.6 shows the architecture resulting from the optimization phase of the Spring Petclinic application. Communities shown in Figure 4.4 have been merged in bigger sets of nodes according to the entity clusters (Figure 4.5) and in such a way that the coupling among obtained microservices is the minimum. We can also notice that the identified microservices represent a vertical decomposition of the system, in which each microservice contains methods from all the layers. Moreover, nodes of each microservice are mainly connected to the domain entities that are persisted by methods of the same microservice. These characteristics allow microservices to be autonomous, to be focused on a single responsibility and within a bounded context, and to fully realize functionalities since each of them contains methods to expose interfaces, realize the business logic, and access the database.

4.1.4 Refactoring

The solution of the optimization problem produces a graph representation of the microservice architecture, in which methods and domain entity classes (i.e., nodes) are grouped into microservices. The implementation of the identified microservices can be automatically realized by suitably "moving" methods and

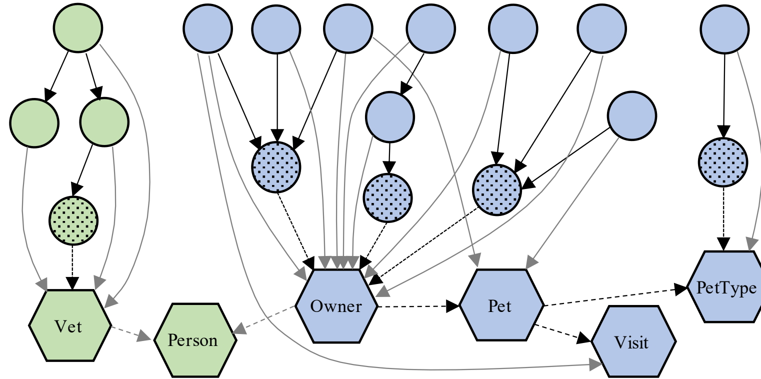


Figure 4.6: Result of the optimization

classes into the right microservice – hence reusing the monolith code – and generating the new API controllers needed to realize the (new) inter-microservice communication. However, it is important to remark that this automated process can be actually performed only if the code of the monolith is suitable to be reused; if, for any reason, this can not be done, the skeleton code of the methods and API interfaces can be still generated. These concerns are discussed in Section 4.3.

The refactoring step generates the implementation of the identified microservices by (i) "moving" the code of each method into the right microservice; (ii) adding new methods for exposing and consuming APIs (*API controller synthesis* and *API consumer synthesis*); (iii) placing (and replicating) the domain entity classes into all the microservices requiring them.

In the following, we describe the rules that allow the generation of API controllers and API consumer methods and the replication of domain entities.

Rule 1: API Controller synthesis

An API controller method is generated for each method that is called from at least one different microservice. That is, in each microservice M_k, an API controller method i' is generated for each method $i \in M_k$ s.t. exists an arc $(j, i) | j \notin M_k$. The generated API controller will receive the API calls from outside the microservice and will invoke the method i. If needed, the methods in the microservice M_k will invoke the method i without using the API.

Figure 4.7 shows an example of the application of this rule. As in the previous figures, methods are represented as circles, entities as hexagons, and microservices – built as a set of methods and entities – are represented as dashed rounded-corner boxes. The node representing the method $m2$ in the microservice M_2 has an ingoing edge from $m1$, which belongs to a different microservice (left-hand side).

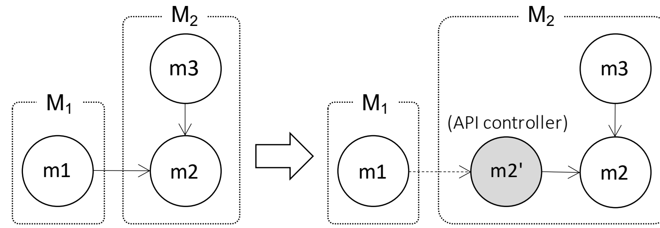


Figure 4.7: Example of API controller synthesis

The method $m2'$ is generated as an API controller for $m2$ (right-hand side). It allows exposing the API to the outside of the microservice and will be used to invoke the method $m2$.

Rule 2: API Consumer synthesis

An API consumer method is generated to call methods that have been placed into different microservices. For this reason, in each microservice M_k, a method j' is generated for each arc (i, j) s.t. $i \in M_k$ and $j \notin M_k$. The new method contains the code needed to invoke the API related to the method j.

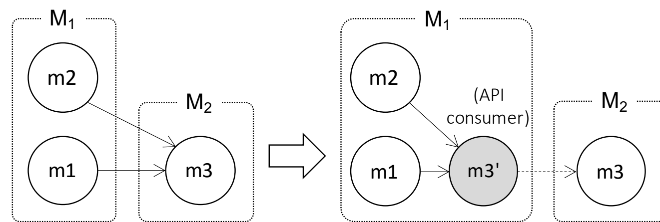


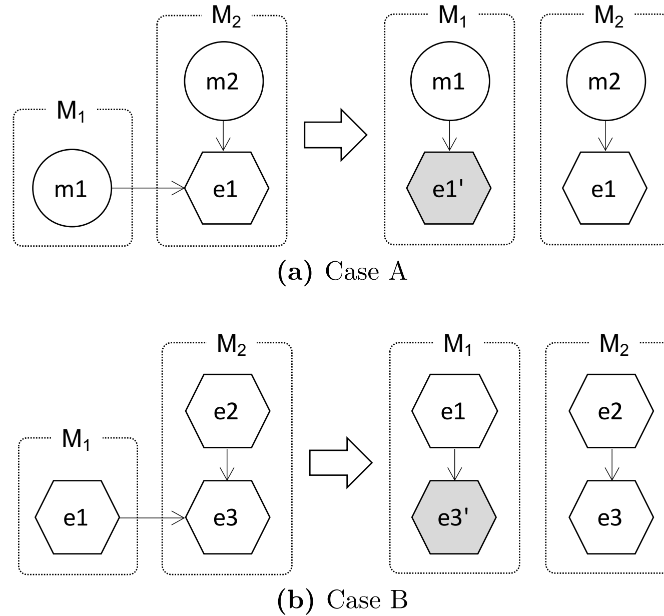
Figure 4.8: Example of API consumer syntesis

Figure 4.8 shows an example of the application of this rule. The nodes representing the methods $m1$ and $m2$ in the microservice M_1 have an outgoing edge towards $m3$, which belongs to a different microservice (left-hand side). The method $m3'$ is generated in M_1 as an API consumer for $m3$ (right-hand side). It allows consuming the exposed API for j (as described in Rule 1) and, hence, invoking the method j.

Rule 3: Entities replication

Alongside the API controllers and consumers, methods must have access to all the required entity classes. Thus, all the entity classes that are referenced by a method are copied into the microservice that hosts the method, together with all the other entities connected to the first entity class. Moreover, also entities that

are referenced by other entities have to be copied into the same microservice. In other words, for each microservice M_k, an entity class e is put into the microservice M_k iff there exists a method-to-entity or an entity-to-entity edge (i, e) s.t. $i \in M_k$.



(a) Case A

(b) Case B

Figure 4.9: Example of entities duplication

Figure 4.9 shows the application of the rule. It considers two cases. In the first case (Figure 4.9a), the entity $e1$ in M_2 is "used" by the method $m1$, which is in the microservice M_1 (left-hand side). The entity is replicated into the microservice M_1 (right-hand side). In the second case (Figure 4.9b), the entity $e3$ in M_2 is referenced by the entity $e1$, which is in the microservice M_1 (left-hand side). The entity is replicated into the microservice M_1 (right-hand side). This refactoring rule is applied iteratively until there are no dependencies between entities spanning across different microservices.

4.2 Evaluation

The main goal of the evaluation is to assess whether our approach produces a decomposed system that follows the main principles of microservice architectures described in Section 2.3. In particular, we check that each microservice: (i) has a well-modularized structure with low coupling and high cohesion, (ii) follows the single responsibility principle, and (iii) independently offers functionalities. To do this, we use cohesion and coupling metrics to quantify how much those design principles are respected and evaluate the structure of the decomposition. Also,

we use the *IFN* metric [65] to evaluate how much independently functionalities are provided, as detailed below. Moreover, we assess the composition of the bounded contexts realized tanks to the entity clustering. We compare the results of the decomposition – before the refactoring phase is performed – with those of other approaches at the state-of-the-art and reference architectures proposed in the literature.

The complete implementation of the approach, its documentation, and all the reported results are publicly available[3].

In the following, we define the evaluation metrics. Then, we describe the performed experiments. Finally, we report the obtained results.

4.2.1 Evaluation metrics

We compute the *average coupling* to quantify how much coupled are the microservices in the obtained architecture. It is computed from the graph representation of the obtained architecture (before the refactoring phase) as the ratio of the sum of the weights of the arcs whose endpoint nodes are in different microservices and the number of identified microservices:

$$AverageCoupling = \frac{1}{|\mathbb{M}|} \sum_{(i,j) \in E | i \in M_h, j \in M_k, h \neq k} w_{ij}, \tag{4.16}$$

where $\mathbb{M}$ is the set of identified microservices, $M_h, M_k \in \mathbb{M}$, and w_{ij} is the weight of the arc (i, j). The less the average coupling, the less coupled a microservice architecture is.

Similarly, we compute the *average cohesion* to quantify how much cohesive are the obtained microservices. It is computed from the graph representation of the obtained architecture as the ratio between the sum of the weights of inners arcs (i.e., those arcs whose endpoints are in the same microservice) and the sum of the weights of all arcs outgoing from the nodes of a microservice:

$$AverageCohesion = \frac{1}{|\mathbb{M}|} \sum_{M_k \in \mathbb{M}} \frac{inner_k}{outer_k}, \tag{4.17}$$

where $\mathbb{M}$ is the set of identified microservices. The values of $inner_k$ and $outer_k$ are obtained as follows:

$$inner_k = \sum_{(i,j) \in E | i,j \in M_k} w_{ij}, \tag{4.18}$$

[3] https://github.com/sosygroup/from-monolith-to-microservices

$$outer_k = \sum_{(i,j) \in E | i \in M_k} w_{ij}. \tag{4.19}$$

The optimal value that a microservice architecture can obtain for the average cohesion is 1, indicating that the elements of each microservice are strongly interrelated. A high cohesion value suggests that all the methods inside a microservice have strong relationships with entities included in the same microservice. That implies that a microservice with high cohesion may be well concerned with a specific bounded context. Conversely, a poor microservice decomposition may show a low cohesion since, e.g., many methods need to reference entities placed in other microservices.

Regarding the independence of functionality, we use the *IFN* metric [65], which measures the average number of interfaces exposed by a microservice. It is defined as follows:

$$IFN = \frac{1}{|\mathbb{M}|} \sum_{M_k \in \mathbb{M}} ifn_k, \tag{4.20}$$

where $\mathbb{M}$ is the set of identified microservices and ifn_k is the number of interfaces exposed by the microservice k. According to our system representation, we define an interface as a set of methods of the logic layer, without ingoing edges, that are connected to the same set of entity nodes. A microservice focused on the single responsibility principle is expected to have only one interface, i.e., to offer functionalities related to a single entity. Hence, the lower the value of *IFN* (down to 1), the most likely the microservice architecture is supposed to follow the single responsibility principle.

Concerning the evaluation of the bounded contexts, we check the entities belonging to each microservice and compute the *precision* and the *recall* score with respect to a reference microservice architecture used as a baseline. To obtain precision and recall, we reverse-engineer each microservice to identify its bounded context(s) and collect its domain entities. We define the *precision* for the microservice k in the architecture d as the ratio between (i) the number of domain entities that are both in the microservice k and in the baseline microservice(s) concerning the same bounded contexts, and (ii) the number of domain entities of the microservice k:

$$Precision(k, d) = \frac{|E_d^k \bigcap E_B^k|}{|E_d^k|}, \tag{4.21}$$

where E_d^k is the set of domain entities into the microservice k in the decomposition d, and E_B^k is the set of domain entities into the microservice k of the baseline.

Similarly, we define the *recall* for the microservice k in the architecture d as the ratio between (i) the number of domain entities in the microservice k provided by the baseline and (ii) the number of domain entities that are both in the microservice k and in the microservice k from the baseline:

$$Recall(k,d) = \frac{|E_B^k|}{|E_d^k \bigcap E_B^k|}. \tag{4.22}$$

A high value of precision and recall means that a given microservice owns the complete set of entities required to fully realize the bounded context, without including "spurious" entities that may be related to different bounded contexts.

4.2.2 Performed experiments

To run the experiments, we have chosen publicly-available systems, implemented in Java, whose architectures are monolithic, and that have been already used in evaluating related works on the decomposition to microservices. Table 4.1 lists the applications that we considered. For each of the considered projects, we show the number of classes (test classes excluded), the number of lines of code, and the number of domain entities and methods identified in the system analysis phase.

Project	Classes	Lines of code	Entities
JPetStore	24	1409	9
Spring Petclinic	23	741	10
SpringBlog	46	1487	9
Cargo Tracking	104	4001	35

Table 4.1: Application used in the reported experiments

JPetstore[4] is a Java web application built on top of the Spring Framework realizing a simple e-commerce system. It is one of the most popular systems and it is used as a reference application in many works [28, 66, 128, 129] (just to mention a few). In [128], an expert decomposition has been proposed for this system. It is a functionality-based vertical decomposition that "cuts" into slices

[4]https://github.com/mybatis/jpetstore-6

all the layers of the system. We use it as a baseline for the comparison of the results.

Spring Petclinic[5] is a sample Spring Boot application realizing a veterinary management system. Spring also released a version of this system featuring a microservice architecture[6] as an example of how to split monolithic applications into a microservice-based system that uses the Spring framework. Being proposed by the same organization that realized the monolithic system, it can be considered a baseline for comparing our results.

SpringBlog[7] is a simple blog system realized using many frameworks such as Spring Boot, Hibernate, and Thymeleaf.

Cargo Tracking[8] is a medium-sized system built on top of the Spring framework. It is used as a reference system for many works, e.g., [13, 59, 81, 82]. In [13], the expected decomposition of the domain model of this system is presented and discussed. We use it as a baseline to evaluate the identification of the bounded contexts and the clustering of the domain entities.

We performed experiments by running our approach for each of the four applications described above and obtained the decomposed microservice architecture of the system. We collected the results and computed the value of the metrics defined in Section 4.2.1. Moreover, to allow the comparison of our results with the other state-of-the-art (*sota*) approaches and baseline solutions, we: (i) collected the architectures resulting from each *sota* and the baselines, (ii) represented them through the graph-based representation that we employ in this work, and (iii) computed the value of the metrics by using the same default arc weights that we used for the decomposition with our approach.

4.2.3 Experimental results

Table 4.2 reports the results of the decomposition of JPetstore. We report the values of the defined metrics for the obtained microservice architecture according to: our approach, four different *sota* approaches [28, 66, 111, 128], and the baseline obtained from [128]. Besides the *average coupling*, *average cohesion*, and *IFN* metrics, we also report (i) the number of method calls (`Calls` relationships) across different microservices and (ii) the number of references from methods to entities (`Uses` relationships) across different microservices. These two values al-

[5]https://github.com/spring-projects/spring-petclinic
[6]https://github.com/spring-petclinic/spring-petclinic-microservices
[7]https://github.com/Raysmond/SpringBlog
[8]https://github.com/citerus/dddsample-core

Approach	**# Microservices**	**Average Coupling**	**Average Cohesion**	**IFN**	**# Method Calls**	**# Entity References**
Baseline	3	1.26	0.95	2.0	*3*	2
Zaragoza et al. [128]	3	3.13	0.90	2.0	6	7
Selmadji et al. [111]	2	3.8	0.93	2.5	*3*	8
Jin et al. [66]	4	2.25	0.82	1.75	7	5
Brito et al. [28]	4	3.1	0.85	1.75	9	8
Our	3	*0.87*	*0.97*	*1.7*	*3*	*0*

Table 4.2: Result comparison for the decomposition of JPetstore

low assessing how much microservices are autonomous and how much the offered functionalities are self-contained in a single microservice. All the considered approaches produced 2 to 4 microservices. Interestingly, our approach outperforms the four approaches and the baseline in all the metrics. It obtained the lowest average coupling (0.87), the highest average cohesion (0.97, very close to the optimal value, 1), and the lowest *IFN*. The number of method calls is 3, as those of the baseline, while there are no entity references from one microservice to another. This suggests that: (i) the decomposition complies with the high-cohesion and loose-coupling principles of microservice architectures, (ii) the functionalities of each microservice are independent and self-contained, and (iii) the microservices are built in the right way according to the bounded contexts of the system.

Table 4.3 reports and compares the results for the decomposition of Spring Petclinic application. We report the metrics computed for the microservice version of this application[9] provided by Spring (baseline) and for the decomposition approach presented by Kamimura et al. [68]. Our approach was able to find automatically two totally-independent microservices: the decomposition obtained the lowest possible value for the coupling (0) and the highest possible value for the cohesion (1), while there are no method calls or entity references across microservices. In contrast, both Kamimura et al. and the baseline show a higher coupling and a lower cohesion. On the other hand, the *IFN* obtained by our approach is slightly higher (2.0) than the others (1.7). This suggests that, in this case, we found microservices with a lower granularity, being slightly less accurate in following the single responsibility principle, but obtaining complete independence.

Regarding the results obtained for the SpringBlog application, our approach produced three microservices with a satisfiable low coupling (0.67) and high cohesion (0.99) with two method calls and two entity references across different microservices. The *IFN* value is 1.67. As a comparison, Jin et al. [65] reported the value of 2.167 for *IFN* for the decomposition obtained with their approach.

Besides the evaluations discussed above, we also assess the results in the identification of bounded contexts and their consequent identified microservices. While JPetstore, Spring Petclinic, and SpringBlog are relatively simple systems with few entities, Cargo Tracking System features a more complex domain model and a wider set of functionalities. Thus, the identification of bounded contexts for this system is not a trivial task, and it may require specific knowledge of the system. In order to perform the evaluation, we applied our approach and reverse-

[9] https://github.com/spring-petclinic/spring-petclinic-microservices

Approach	**# Microservices**	**Average Coupling**	**Average Cohesion**	**IFN**	**# Method Calls**	**# Entity References**
Baseline	3	1.2	0.75	*1.7*	2	3
Kamimura et al. [68]	3	2.07	0.76	*1.7*	2	7
Our	2	*0.0*	*1*	2.0	*0*	*0*

Table 4.3: Result comparison for the decomposition of Spring Petclinic

Approach	# Microservices	Identified services	Included domain entities
Baseline	4	Voyage	Voyage, CarrierMovement
		Location	Location
		Planning	Cargo, Leg, Itinerary, RouteSpecification
		Tracking	HandlingEvent, Delivery
Baresi et al. [13]	4	Trip	CarrierMovement, RouteSpecification
		Tracking	HandlingEvent, Delivery, Voyage
		Planning	Location, Itinerary, Leg, Voyage
		Product	Cargo
Service Cutter [59]	3	Location	Location
		Tracking	HandlingEvent, Delivery
		Voyage & Planning	CarrierMovement, Itinerary, Voyage, Leg, RouteSpecification, Cargo
Our	3	Tracking	HandlingEvent
		Planning & Location	Itinerary, RouteSpecification, Delivery, Cargo, Leg, Location
		Voyage	Voyage, CarrierMovement

Table 4.4: Comparison of the microservices identified for Cargo Tracking system

engineered the obtained decomposition to find the distribution into microservices of the domain model entities. We then labeled the microservice with the name of its emerging bounded context. Table 4.4 reports the distribution of domain entities in each identified service by: our approach, Service Cutter [59], the approach by Baresi et al. [13], and the manually-obtained expected decomposition provided in [13] as a baseline. The baseline consists of 4 microservices (i.e., 4 bounded contexts according to microservices are built): *Voyage*, *Location*, *Planning*, and *Tracking*. The interface analysis run by Baresi et al. was able to find 4 microservices as well, although with different bounded contexts (except for *Planning*). Service Cutter found 3 microservices: *Location*, *Tracking*, and *Voyage & Planning*. As one can notice, bounded contexts are distributed into microservices with a lower granularity (e.g., instead of having *Voyage* and *Planning* in separate microservices, they have been put together). Also, our approach found 3 microservices: *Tracking*, *Planning & Location*, and *Voyage*, thus resulting in a lower granularity than the baseline.

The *Voyage* microservice identified with our approach got 1 for both precision and recall, while *Tracking* microservice got 1 for the precision and 0.5 for the recall. Furthermore, if we compute the metric's values for *Planning & Location* by considering the union of the services *Planning* and *Location* from the baseline solution, we obtain a precision of 0.83 and a recall of 1. The average value for the precision and the recall on the three identified microservice is 0.94 and 0.83, respectively. In comparison, Service Cutter was able to get 1 for the values of both precision and recall for the microservices *Location* and *Tracking*. Also, if we compute the metric's values for *Voyage & Planning* by considering the union of the services *Voyage* and *Planning* from the baseline solution, we obtain 1 for both the metrics. These results suggest that, at least in this case, and despite not being optimal in the identification of bounded contexts, our approach still obtained comparable results. However, we remark that all the results have been obtained in an automatic way, without requiring any further input or specific knowledge of the system.

4.3 Discussion

Results obtained from the performed experiments highlight that the proposed approach is able to identify architecturally-meaningful microservices such that the main principles of MSA are held. Measured metrics suggest that microservices obtained through this approach are independent and self-contained, outperform-

ing the related approaches whose results have been compared to. We have to remark that these results have been obtained in a fully-automated way, hence without any knowledge of the system, models, or the manual intervention of a system expert.

4.3.1 Threats to validity

There are a series of factors that may challenge our approach. In the following, we discuss the most important ones.

Internal validity

The evaluation has been performed on a set of metrics that are computed on the same graph representation that we used in our approach. Besides the number of used metrics (that we plan to expand in the future for a more accurate evaluation), computing all of them by relying on the same system view may lead to biased results. However, in the comparison of our solution with other microservice architectures, in order to avoid other biases, we have taken baselines from other works, so we have not been involved in their definition. Also, cohesion and coupling-related metrics, together with *IFN*, despite being widely used for evaluating decomposed architectures [28, 82, 117], do not guarantee that the obtained architectures are the best-fitting for the applications domains and needs, as many different architectures there may exist [51]. Moreover, we have tested our approach only on a small set of projects, that were not very big both in the number of classes and entities. Thus, we still need to test it on a wider variety of systems in terms of both dimension and domain model complexity in order to better assess the scalability of the approach.

External validity

By default, our tool assigns predefined weights to build the information graph after the static code analysis. This permits to fully automate the process, while sparing developers from the task of manual weight specification. However, the default values cannot be generally taken as the best-fitting weights, since they may be less accurate for specific systems, e.g., very big or complex systems. As mentioned in Section 4.1.1, our tool allows developers to refine weights, e.g., according to application-specific requirements.

The use of static analysis as the only mean of system analysis may represent a limitation of the approach. Without dynamic analysis, the tool is not aware of the

actual number of method calls that are performed at runtime. This may lead to a decomposition that may not optimize the number of runtime inter-microservices calls. This may affect the system's performance since inter-microservices calls introduce overhead due to network communication. However, the approach can be easily extended to consider data coming from dynamic analysis, by, e.g., including the number of method calls performed at runtime into the information graph and considering it in the optimization's objective function.

The quality of the decomposition may naturally be affected by the bad quality of the system's source code (e.g., it does not follow OOP principles, it has very badly designed classes, or there are code smells that reduce the separation of concerns between classes). In fact, if the relationships among classes representing the domain model are not reported in the code according to the OOP paradigm, we may lose cohesiveness and fail to identify the clusters of entities. This may lead to a bad identification of the system's domain contexts and, as a consequence, to failure in adhering to the single responsibility principle. The same holds if the domain model is very big, complex, and highly interconnected. However, we allow developers to adjust arc weights individually to identify the domain entities that are more closely related, and we also allow them to adjust the entity clusters to overcome these issues.

Finally, the presented approach has been designed and tested for analyzing systems that are built by leveraging layered architectures. This limits the applicability of the approach if systems to be decomposed follow other architectural styles, e.g., the MVC pattern [106]. However, systems realized through the MVC style are amenable to being decomposed with the approach proposed in this work. In fact, like the three-layered architectures, also in the MVC there are three components with well-defined roles and concerns. Classes from the "model" component of the MVC pattern could be treated similarly as we do with domain entities classes, and their clustering could drive the decomposition of the system just like it is described in this work. A light refinement in the analysis step could open this possibility. However, the applicability of the approach in those cases would require more experimental evidence for its validation.

4.3.2 On the applicability of the approach

Experimental results concerning the proposed approach are encouraging and hint at the fact that the approach can be furtherly developed and applied to the refactoring of monolithic systems. In particular, it is suitable in those situations in which the system to be migrated requires a reverse-engineering phase due to

the lack of models or expertise. On the other hand, it may not be suitable whether the system architecture is particularly eroded or the technological stack of the system is getting obsolete. In that case, the automated refactoring phase may not be a real advantage since the whole codebase of the system should be rewritten or moved to a new technology or language. Interestingly, this approach is particularly promising for all the cases in which systems are built by following the "monolith-first" approach [48, 89, 93], requiring the migration only in order to support, e.g., its growth or improve its scalability. In these cases, the system to be migrated is likely to be fresh enough to adopt a modern technological stack, use modern frameworks, and have a sound architecture. This allows the approach to be able to perform the complete migration process, even implementing the whole set of microservices by suitably moving or generating the code into the right microservice.

Chapter 5

Architectural style for scalable choreography-based systems

Monolithic systems, like those considered for the migration in the previous chapter, are not always meant to be *standalone* services that run in isolation without interacting with any other service. Rather, as described in Section 2.3.4, they can be involved in SOA systems and even composed as choreographies. As widely discussed in this thesis, choreography-based systems require a proper coordination mechanism to ensure the correct system behavior and avoid undesired interactions.

Importantly, systems are required to be *scalable*. They need to be able to provide satisfiable performances under heavy loads [26], thus enhancing the user-perceived performances and positively influencing the overall dependability of the system [54]. Benefits brought by MSA in terms of scalability are one of the key factors that may induce companies to migrate their monolithic services into microservices [114]. Either if obtained after having decomposed monoliths or not, systems can be realized by composing (micro)services into choreographies, obtaining the desired scalability and furtherly enhancing loose coupling and flexibility.

In such systems, it is needed that, beyond the coordination layer, a load balancing layer is added to the system architecture to support the service scalability, allowing the replication of service instances and distributing the workload among them, thus supporting the realization of choreography-based systems in a microservice style.

Given the above, in a choreographed microservice-based system, coordination and load-balancing concerns coexist. They both must be taken into account to realize the required coordination and allow the scalability of the system. However, they represent two distinct functionalities that should remain independent and

loosely coupled.

This chapter addresses this problem by presenting an architectural style capable of totally decoupling these two concerns. We first present a case study able to highlight the complexities and the challenges to be addressed, then we present the architectural style and show it at work on the case study. Finally, we evaluate the architecture by discussing the several design alternatives enabled by the architecture and report on the experiment conducted on the case study system.

5.1 Case study

In this section, we present a case study that allows us to instantiate the challenges described above through an example system related to an online ticketing platform. The system has been inspired by well-known applications such as Train Ticket System[1] and Sockshop[2]. It is designed with the specific purpose of realizing a motivating scenario whose characteristics allow emphasizing the needs for both coordination and load balancing and stress the capabilities of the architectural style. The system taken as an example consists of an application that allows customers to browse, select, and buy tickets for sports events or shows. When a customer selects the tickets, the system checks if the customer is allowed to buy tickets, and if so, it will add the selected tickets to her cart and will reserve them for a short period of time until it provides the requested checkout information. If the user proceeds with the checkout within 10 minutes from the ticket selection, the payment is processed and the selected tickets are issued, printed, and prepared for shipping. Otherwise, the temporary reservation is released and the user cart is emptied.

The system is realized as a choreography resulting from the collaboration of several microservices:

- *Event Catalog*: it owns the information about the events whose tickets are sold on the platform;
- *Ticket Portal Manager*: it manages the ticket selling process by interacting with the customer;
- *Blacklist Service*: it owns the list of blacklisted people from events (e.g., fans that are not allowed to enter stadiums, etc.);

[1]https://github.com/FudanSELab/train-ticket
[2]https://microservices-demo.github.io

- *Reservation Service*: it owns the information about the tickets that have been sold and the ones that are temporarily reserved because selected by customers while the checkout is performed;
- *Cart*: it manages the user's online cart;
- *Checkout Manager*: it manages the checkout process, by receiving the customer checkout information and arranging the payment for the tickets that the customer has in her online cart;
- *Payment*: it processes the payment;
- *Ticketing Manager*: it manages the ticket issuing process;
- *Ticket Service*: it emits tickets;
- *Print Service*: it manages the printing of the tickets;
- *Shipping*: it manages the shipping of tickets to customers.

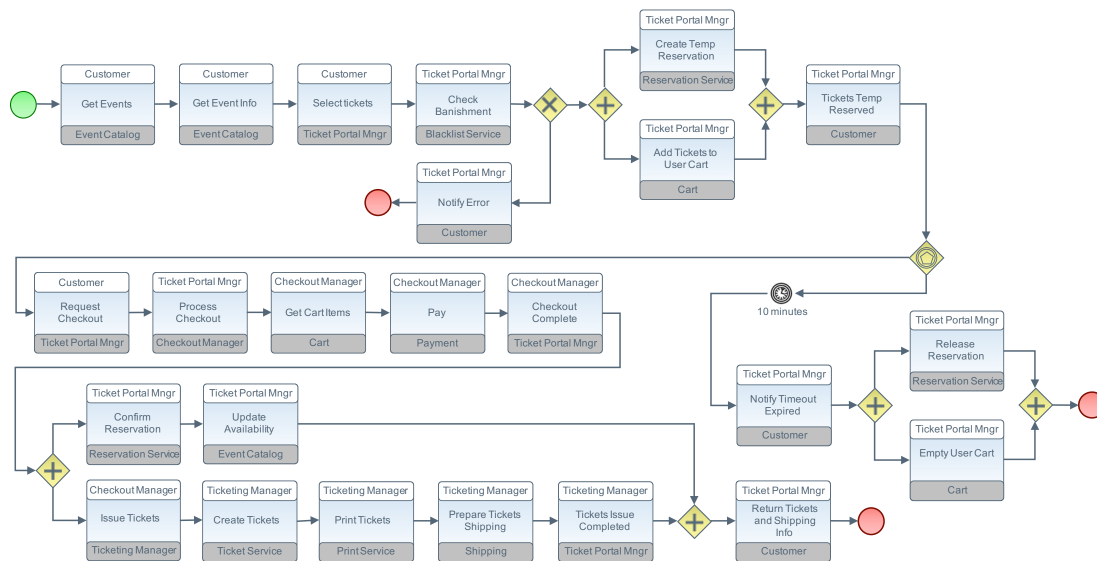


Figure 5.1: Online ticketing system choreography

Figure 5.1 portrays the BPMN2 choreography diagram of the system. Note that, besides the microservices listed above, we modeled an extra participant, *Customer*, representing the actor of the system that interacts with the application by sending messages and receiving back replies through an application (e.g., a web-based application, mobile app, etc.).

5.1.1 Arising complexities

The realization of this system poses a series of challenges that have to be addressed. Some of them concern the problems of ensuring the correct service coordination, in order to realize the interactions prescribed by the choreography specification. Others concern the ability of the system to scale and remain responsive even under high traffic demands.

First, it has to be considered that, since there are many customers attempting to buy tickets, there are race conditions to be dealt with. In fact, when tickets are selected by a customer, they are temporarily reserved in order to let her complete the checkout process. For this reason, the checkout process needs to be performed only after that the tickets have been reserved: if this does not happen, some other customer may reserve the same tickets and proceed with the checkout at the same time. This may happen if a customer attempts to perform the checkout after that the 10-minute timeout has expired. These situations may lead to errors and conflicts in the tickets assignment. For this reason, they clearly represent undesired interactions that require proper coordination in order to be avoided. As described in Section 2.2, these issues can be solved through the use of CDs, suitably introduced in a *Coordination Layer*, capable of enforcing the correct choreography behavior.

Moreover, the system has to be able to scale in order to guarantee responsiveness, availability, and satisfactory user-perceived performances. Stress to system performances happens when the platform opens the sales for an important event with many customers attempting to buy the tickets right on the opening. In these situations, the number of requests that the system is required to handle explodes. The performance of the system may get significantly worse for both customers attempting to buy tickets for the new event and users that are on the platform for other reasons. Hence, the system can result unresponsive or unavailable. In addition, different service-level agreements (SLAs) can be applied to provide some users with high-performance guarantees. This means that the interactions of certain kinds of users might be prioritized. For these reasons, a load balancing system is needed to properly distribute the requests among the available microservice instances, so as to keep the system available, responsive, compliant to different SLAs, and to avoid the degradation of user-perceived performances.

5.2 Architectural style

The architectural style we propose, shown in Figure 5.2, is aimed at supporting the scalable composition of microservices as choreographies.

For the sake of simplicity, in the description of the architectural style, we consider the coordination layer without taking into account the challenges (and related solutions) concerning context-awareness discussed in Chapter 3. For this reason, in the following, we will consider only the "classical" CDs (Section 2.2) when referring to the coordination layer of the architecture. It is easy to understand that, as a straightforward extension, the coordination layer can be improved by using caCDs to realize the context-awareness capabilities.

5.2.1 Architectural layers

The architectural style is organized in several layers:

- **client-side layer:** comprises pure client applications as well as the client-side of prosumer services;
- **coordination layer:** contains the coordination logic;
- **client-side load balancing layer:** consists of the client-side load balancers;
- **server-side load balancing layer:** constituted by server-side load balancers;
- **server-side layer:** comprises pure provider services as well as the server-side of prosumer services.

The coordination layer is enhanced to cope with the microservice style. In fact, CDs now interact with many instances of fine-grained microservices. Hence, they are replicated as multiple instances employing lightweight communication mechanisms to support the replication of the instances of the coordinated microservices: an instance of a CD is deployed for each instance of the coordinated microservice. As a consequence, also the scalability of the coordination layer is improved with respect to the one presented in Section 2.2. However, CDs still interact only with the other architecture components (i.e., there is no communication between the instances of the same CD) and they are agnostic of the number of their replicated instances. Moreover, the association between CD instances and

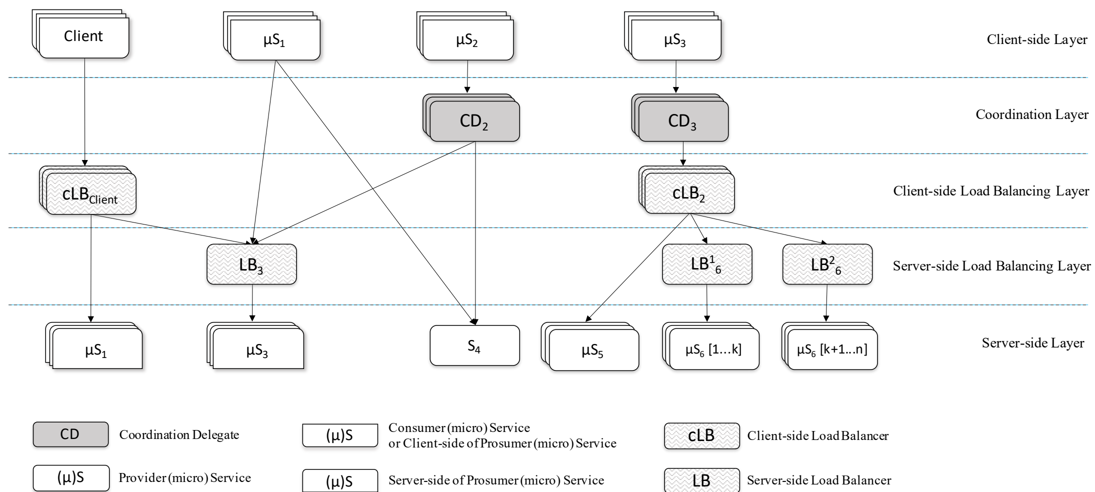


Figure 5.2: Architectural style

microservices instances prevents the possibility of the coordination layer acting as a bottleneck and representing a single point of failure. The multi-instance nature of both CDs and microservices requires load balancing mechanisms to support scalability by distributing the workload across microservices instances. Thus, a load balancing layer is added between the coordination and server-side layers.

The architectural style is flexible with respect to the coordination and the balancing of interactions. In fact, the interactions are coordinated and/or balanced only when needed. Moreover, the two load balancer layers can be combined, leading to a hybrid approach for load balancing. As a result, the architectural style enables the following architectural configurations:

1. interactions neither coordinated nor balanced,
2. interactions not coordinated but client-side balanced,
3. interactions not coordinated but server-side balanced,
4. interactions not coordinated but hybrid balanced,
5. interactions coordinated but not balanced,
6. interactions coordinated and client-side balanced,
7. interactions coordinated and server-side balanced,
8. interactions coordinated and hybrid balanced.

These configurations can be divided into two main categories: coordinated and not coordinated interactions. As explained in Section 2.2, the coordination of interactions, when needed, enforces the control flow specified in the choreography and avoids undesired interactions. Within each category, we can further identify three sub-categories related to the load balancing approaches applied: client-side, server-side, and hybrid load-balancing. As anticipated in Section 2.4, the load balancing approaches have several pros and cons, for example, the server-side approach provides security benefits but represents a single point of failure and a bottleneck. On the contrary, the client-side approach does not represent a bottleneck or a single point of failure but does not address security concerns. The hybrid approach allows providing different resources availability, costs, or SLAs through dedicated server-side load balancers, but at the same time, it impacts the latency due to the proxy extra hops represented by the server-side load balancers. The flexible nature of the architectural style permits designing a system using the suitable load balancing approach(es) according to the system's needs.

Concerning the relationships between coordination and balancing layers, we highlight that layers are arranged so that CDs coordinate interactions before the load balancing is performed, if needed. In other words, load balancers only handle already-coordinated interactions. Hence, the coordination does not impact the load balancer layer(s) since it is performed before that a load balancer routes the request toward the target service instance. Also, the choreography ID, required for correlating messages for the purpose of coordination (as explained in Section 2.2) is completely transparent to the balancing layers. In synthesis, each layer has a specific concern and does not account for the functionalities offered by other layers, being reciprocally agnostic and their functionalities completely decoupled.

5.2.2 Architectural style at work

Figure 5.3 shows the architectural style applied to the online ticketing case study described in Section 5.1. The resulting architecture, beyond handling normal traffic conditions, is capable of dealing with the high traffic related to the ticket sale for an important event by providing different SLAs to users. In fact, premium users are guaranteed better performance and prioritized access to the system by using dedicated microservices instances. In this scenario, the most stressed parts of the system are *Event Catalog*, *Reservation*, and *Ticket Service*. Thus, for each microservice, two server-side load balancers are employed to distribute the workload across the related microservices instances: $\text{LB}^1{}_{\text{EventCatalog}}$ and $\text{LB}^2{}_{\text{EventCatalog}}$

for *Event Catalog*, $LB^1_{Reservation}$ and $LB^2_{Reservation}$ for *Reservation*, $LB^1_{Ticket\ Service}$ and $LB^2_{Ticket\ Service}$ for *Ticket Service*. Moreover, the CDs corresponding to the participants interacting with these services, i.e., $CD_{Customer}$, $CD_{TicketPortal}$ and $CD_{Ticketing}$, are equipped with client-side load balancers leading to a hybrid load balancing. In this configuration of hybrid load balancing, a server-side load balancer is in charge of managing only the traffic generated by premium users. This allows prioritizing premium users according to the related SLA. The hybrid approach is also employed for the microservices: *Ticket Portal Manager*, *Checkout*, and *Ticketing*. Differently from the previous case, the related microservices instances are balanced by a server-side load balancer per microservice type connected with the client-side load balancers of the CDs: $CD_{Customer}$, $CD_{Checkout}$ and $CD_{Ticketing}$. This configuration of the hybrid load balancing routes the traffic of premium users to a dedicated subset of the microservices instances. In fact, the involved microservices handle a light workload, hence it is enough to provide only a dedicated subset of instances to comply with the SLA of premium users. The instances of the microservices *Customer*, *Cart*, and *Print* are balanced only by the client-side load balancers of the CDs they interact with, i.e., cLB_{Portal}, $cLB_{Checkout}$ and $cLB_{Ticketing}$. The hybrid load balancing is not required for *Customer*, *Cart*, and *Print*. Indeed, due to their nature, they manage simpler and fewer requests with respect to the other microservices, and hence their interactions can be properly balanced through client-side load balancers. Finally, the services: *Blacklist Service*, *Payment*, and *Shipping* are represented as single-instances since they are third-party existing services.

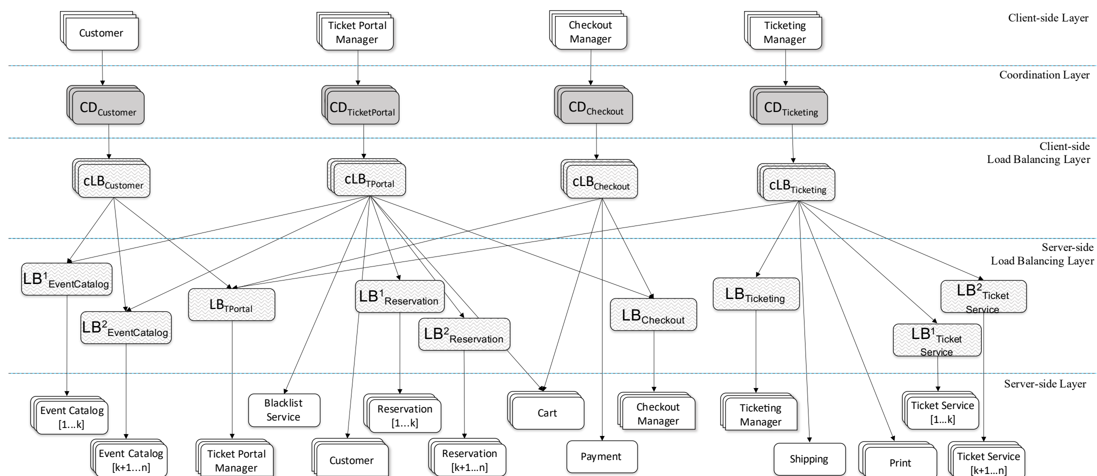


Figure 5.3: Online ticketing system architecture

5.3 Evaluation

The realization of a scalable choreography-based microservices system is obtained through the combination of coordination and load-balancing mechanisms.

The *Coordination Layer* has been experimentally evaluated concerning the system performances in Chapter 3, without detecting issues to consider it a bottleneck. On the other hand, as already discussed, the overall scalability of the system resulted to be weak due to the poor scalability of the involved services. In fact, as shown in Section 3.6, with high demand rates, the execution times of the involved services significantly grow, causing the degradation of user-perceived performances. The same results have been obtained in previous work concerning the approach for the coordination of service choreographies [8].

Thus, the evaluation conducted in this chapter focuses specifically on the load balancing layers, i.e., client-side and server-side load balancing layers, with the aim to evaluate their impact on the scalability of the system and, as a consequence, on the user-perceived performances. Moreover, the evaluation also focuses on the support that the possible design alternatives for load balancing have on some of the dependability attributes of the system.

Evaluations goals: the evaluation concerns the following three aspects of the architectural style and answers the related Evaluation-Questions (EQs):

- **Scalability support:**
  EQ1: How is scalability supported by the architectural style?
- **Dependability support:**
  EQ2: How is dependability supported by the architectural style?
- **User-perceived Performances impact:**
  EQ3: What is the impact of the load-balancing layers on the User-perceived Performances?

Evaluations method: the evaluation is carried out both as an *Argumentation* and a *Technical Experiment* [73]. In particular, the *Argumentation* is used to answer **EQ1** by arguing the support of the design alternative with respect to the following architectural properties: Load balancing scalability (LB Scalability) and Server-side scalability (SS Scalability); whereas **EQ2** is answered by arguing about the architectural properties: User-perceived Performances (UP Performances), Evolvability, Reliability, and Security. Moreover, to quantify the

impact of the load-balancing layers on the UP Performances, and hence to answer **EQ3**, a *Technical Experiment* concerning the implementation of the online ticketing system (Section 5.1) has been conducted.

5.3.1 Evaluation of design alternatives

Table 5.1 summarizes the results of the evaluation of the design alternatives with respect to the considered architectural properties. In particular, from the several architectural configurations enabled by the proposed architectural style (Section 5.2), we derived the following design alternatives: *Client-Side Load Balancing*, *Server-Side Load Balancing* and *Hybrid Load Balancing.* For each of these design alternatives, the table reports if a specific architectural property is supported in a poor (*), moderate (**), or good (***) way.

Table 5.1: Design alternatives evaluation

Design Alternative	UP Performances	LB Scalability	SS Scalability	Evolvability	Reliability	Security
Client-Side	***	***	*	*	***	*
Server-Side	*	*	**	***	*	***
Hybrid	**	**	***	**	**	***

Good: ***, Moderate: **, Poor: *

User-perceived performances Concerning User-perceived performances, the client-side load balancer uses a direct connection with the server without introducing proxy extra hops. Moreover, it does not represent a bottleneck because it handles the traffic of its local microservice instances in a distributed way [80]. Thus, the client-side load balancer supports the User-perceived performance in a good way. Conversely, the server-side load balancer represents a bottleneck and it introduces an extra hop [72, 97], hence, it supports the User-perceived performance in a poor way. The hybrid load balancing mitigates these disadvantages. In fact, the negative effects of the extra hop and the bottleneck represented by the server-side load balancer are reduced if the system resources and the number

of service instances per server-side load balancer are properly determined. Thus, the hybrid load balancing supports user-perceived performance in a moderate way.

Load balancing scalability Regarding scalability, we consider two dimensions: load balancing scalability (LB Scalability) and server-side scalability (SS Scalability). The former refers to the ability of the load balancer to scale with respect to the number of interactions, whereas the latter involves the ability of the load balancer to support the scaling of the microservice instances. Load balancing scalability is supported in a good way by the client-side load balancer since it can add new load balancing capacity to the system each time a new microservice instance appears. Server-side load balancer supports load balancing scalability in a poor way because it can handle incoming requests by applying vertical scaling up to the availability of the computational resources. The hybrid load balancer allows a flexible form of scaling by adding or removing both service instances and load balancer instances, and hence it supports load balancing scalability in a moderate way.

Server-side scalability Server-side scalability is supported in a poor way by the client-side load balancer since it does not manage the number of microservice instances it can use. In fact, a client-side load balancer can't understand whether to add or remove microservice instances, it just distributes its requests among the list of available instances and it can possibly lead to an uneven load distribution [80]. Contrariwise, a server-side load balancer has a continuous awareness of the workload of each microservice instance and hence, according to the traffic, it can easily figure out whenever microservice instances should be added or removed. Moreover, the knowledge of the traffic on the microservice instances allows the server-side load balancer to implement more accurate balancing algorithms. Thus, the server-side load balancer supports server-side scalability in a better way with respect to the client-side. The hybrid load balancer employs both the client-side and server-side load balancer, and hence it allows distributing requests to suitably partitioned microservice instances complying with resources availability, costs, or service-level agreements (SLAs). Therefore, the hybrid load balancer supports server-side scalability in a good way.

Evolvability We consider evolvability as the degree to which a component implementation can be changed without negatively impacting other components [47].

A change in a local client-side load balancer implementation may impact the balanced client service resulting in its re-deployment [115]. In fact, client-side microservices may be tightly coupled with the implementation of their client load balancers, (e.g., a microservice and its client load balancer are within the same executable artifact). In such cases, they need to be re-deployed. This introduces a high downtime, and hence a client-side load balancer supports evolvability in a poor way. In a server-side load balancer, the downtime is limited because the change affects only the load balancer [115]. This allows server-side load balancing to support evolvability in a good way. The hybrid load balancer supports evolvability in a moderate way because it permits to mitigate the change impact. This can be obtained by applying the change only on the server-side load balancer whenever possible, restricting the implementation of the change on the client-side load balancer only when strictly required.

Reliability Reliability from an architectural point of view can be defined as the degree to which an architecture is susceptible to failure at the system level in the presence of partial failures within components, connectors, or data [47]. Client-side load balancer due to its fully distributed nature does not represent a single point of failure, and hence it supports reliability in a good way. Conversely, the server-side load balancer introduces a single point of failure and it supports reliability in a poor way. Hybrid load balancer ease this disadvantage by employing several server-side load balancers. Thus, it supports reliability in a moderate way.

Security Regarding security, server-side load balancer hides the internal structure of both the application and the network because clients can't contact directly the microservice instances. Thus, server-side load balancer supports security in a good way. On the contrary, client-side load balancer allows clients to directly interact with the microservice instances, and hence it supports security in a poor way. Hybrid load balancer, because of the presence of server-side load balancer, mitigates completely the problems of the client-side load balancer, thus providing good support for security. In fact, in the hybrid setting the client-side load balancer can only interact with the server-side load balancer, hence clients are not aware of the internal network and possible malicious interactions are prevented [7].

5.3.2 Experimentation

As anticipated, we performed an experiment by implementing and running the system presented in Section 5.1.

This experimentation had the purpose of evaluating the benefits of the presence of a load-balancing layer on the user-perceived performances, in particular in those situations in which there is a high traffic load due to the presence of a large number of users that are accessing the system. Thus, as an indication of the user-perceived performances, we measured the response times for the tasks that involve the *Customer* service (i.e., the user): (i) *Get Events*, (ii) *Get Event Info*, (iii) *Select Tickets* (by measuring the time between the request sent by *Customer* for that task and the reply message received for *Tickets Temp Reserved*), and (iv) *Request Checkout* (by measuring the time between the request sent by *Customer* and the reply message received for *Return Tickets and Shipping Info*). We have run the system by executing the choreography with an increasing number of concurrently interacting users and compared the obtained results for two different scenarios: (i) without any load balancing layer (*unbalanced scenario*), and (ii) with the two load balancing layers by configuring the system as described in Section 5.2.2 (*balanced scenario*).

The prototypal implementation of the system used for experimentation is publicly available[3].

Experimentation Setting

The goal of the experimentation is to specifically evaluate the impact of the load balancing layers on the user-perceived performances, by comparing the results obtained in the two different settings. Since the architecture only focuses on coordination and load balancing, we avoid considering aspects such as how to distribute, replicate and scale databases, being independent of our architectural style. Also, we do not address the problem of guaranteeing consistency among the distributed and possible replicated databases. Thus, we excluded the persistence from our tests to avoid that results could be biased and influenced by possible bottlenecks in the data access, since it may impact system performance.

In order to perform tests, *Customer* service has been implemented as an application that simulates the presence of a varying number of users that concurrently interact with the system. For each simulated user, it sends the requests to the system according to the execution flow prescribed by the choreography definition:

[3] https://github.com/sosygroup/microservices-ticketing-system-prototype

it first sends a message to execute *Get Events* task, then a message for *Get Event Info*, then a message for *Select Tickets*, and at the end a message for *Request Checkout*. The implementation of *Customer* is realized by leveraging the Locust[4] load testing tool. It allows simulating the behavior of the user by considering also thinking times (randomly selected in a range from 5 to 10 seconds) before sending a message after receiving the response for the previous task. Tasks are executed according to the behavior specified in the choreography, so to simulate the correct sequence of interactions. In this way, we assume that *Customer* is always behaving properly, i.e., it does not attempt to perform undesired interactions as explained in Section 5.1, since the effectiveness of the coordination has been already evaluated in Chapter 3 and in [8] and it is out of the scope of this experimentation.

Data have been collected by the *Customer* service, which locally logs the timestamps of each of the measured interactions listed above, in a way that the collection of data does not interfere with the message exchanges between microservices, CDs, and load balancers. Only when all the instances have been executed, logs are collected and analyzed in order to extract the system response times for each of the operations.

Each of the two scenarios has been tested by running the choreography multiple times with an increasing number of concurrent simulated users, starting from 50 up to 10000, in order to simulate both situations with low traffic and situations with very high traffic.

The experimentation has been performed using eight Virtual Machines (VMs) installed in four distinct Server Machines (SMs). Each SM is equipped with 2CPUs Intel Xeon E5-2650 v3, 2.3 GHz, 64 GM RAM, and 1 Gb/s network. Each VM has 4 CPU cores and 4 GB of RAM. The VMs are equipped with Ubuntu Server 22.4 as operating system. Open Stack is the cloud infrastructure provider. Computational resources (CPU cores, RAM, network) of SMs are equally allocated to VMs. Thus, we deployed all the components by following a round-robin approach, except for the CDs, which have been conveniently put (although not mandatory) into the same VM with the consumer or prosumer services they coordinate. In this way, services are uniformly distributed into the VMs to equally distribute the load among them and to avoid resources being consumed in a uniform way. In general, our approach does not constrain the deployment setting, being all the components free to be deployed according to any strategy. The deployment settings related to the two scenarios mentioned

[4] https://locust.io

above are described in Figure 5.4 (unbalanced scenario) and Figure 5.5 (balanced scenario).

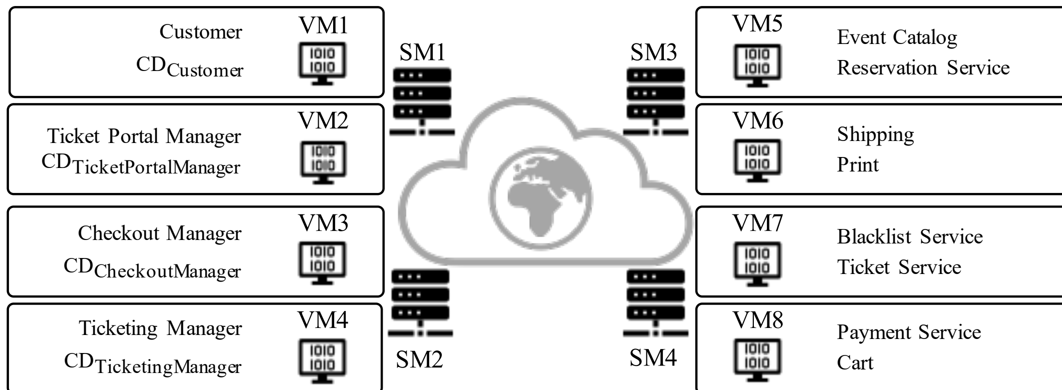


Figure 5.4: Deployment setting for the scenario without load balancing

Experiment Results

The chart in Figure 5.6 shows the results of the experimental runs. Dotted lines describe the trend of mean response times of the four considered interactions in the unbalanced scenario. Continuous lines, instead, describe the trend in the balanced scenario.

The first thing that leaps into the eye is that in the unbalanced scenario, when the number of concurrent users increases over 1000, the response times raise in a significant way. In particular, the mean response time for the *Checkout* interaction raises from 33.7 ms when there are 1000 concurrent users (medium load) to 65 ms when there are 10000 concurrent users (very high load), hence doubling the value. The same holds for the interactions *Get Events* (response time raising from 3 to 9 ms) and *Ticket Selection* (raising from 11 to 24 ms). *Get Event Info* showed a response time raising from (3.2 to 4.9 ms). Despite the fact that the increase in response times is sub-linear with respect to the increase in the number of concurrent users (response times double when the number of concurrent users has a ten-times growth), in this experimental setting a degradation of user-perceived performance can be observed. On the contrary, in the balanced scenario, the raise of the mean response times is reduced significantly (from 34.6 to 37 ms for *Checkout* interaction). Thus, the load balancing resulted to be able to effectively reduce the loss of user-perceived performances when the system undergoes high demand rates.

Moreover, we can observe that with very low to medium loads (50 to 500 concurrent users), the mean response times for all the considered interactions in the

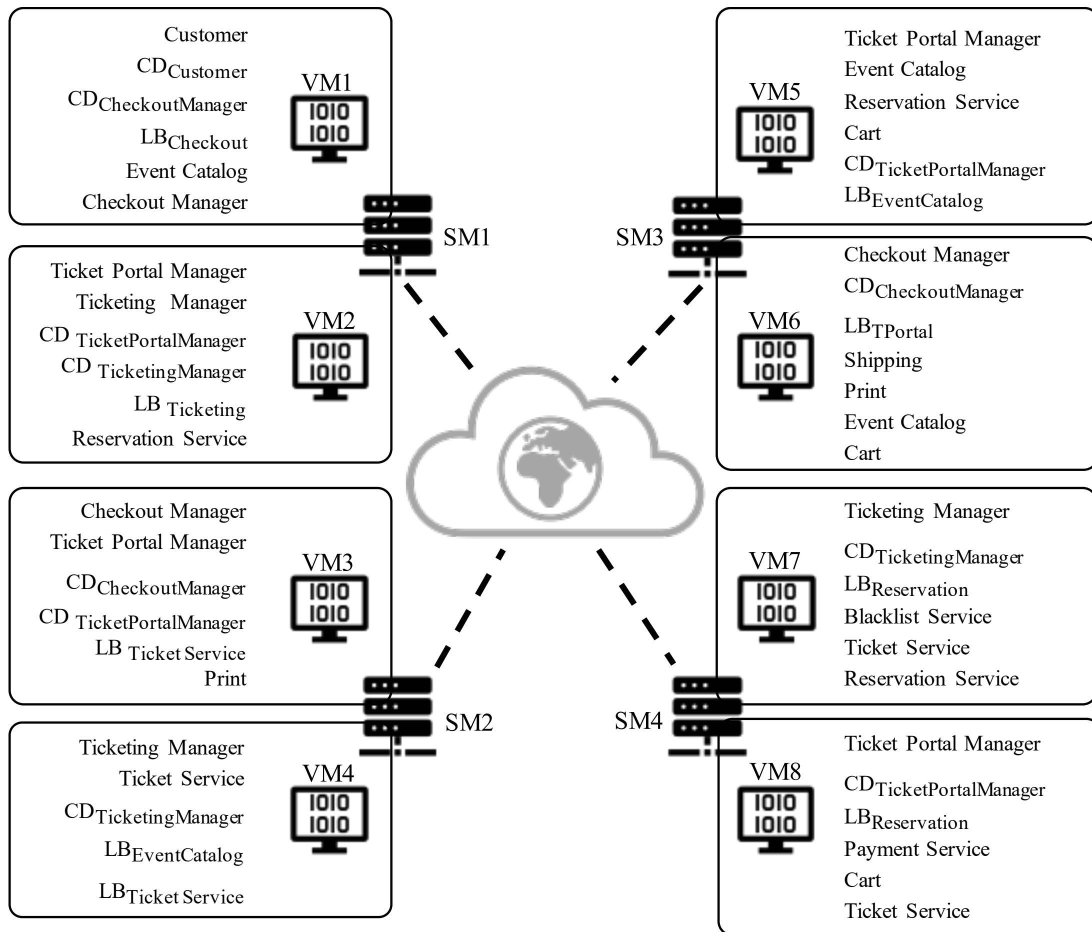


Figure 5.5: Deployment setting for the scenario with load balancing layers (CDs are equipped with local client-side load balancer)

balanced scenario are higher with respect to the unbalanced. This is likely due to the presence of server-side load balancers, which require an extra-hop in the message passing between the microservices involved in the interactions. However, the difference between the mean response times in the two configurations is negligible (4.6 ms at most), while the load balanced setting significantly outperforms the unbalanced one in supporting the system's scalability and, hence, avoiding the degradation of user-perceived performances.

5.4 Discussion

The evaluation of the proposed architecture highlighted how the load balancing layers, in support of the composition of microservices and the distribution of their workload, are able to enhance a number of non-functional properties (user-perceived performances, scalability, evolvability, reliability, and security) of the

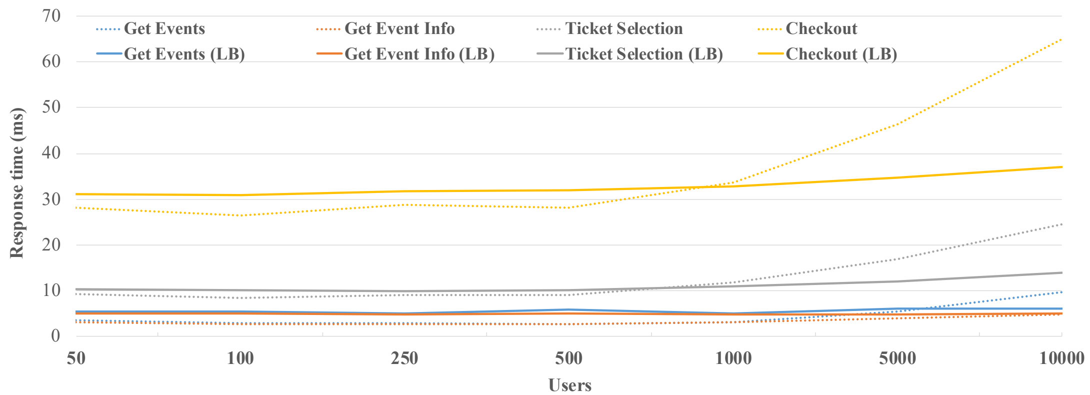


Figure 5.6: Experimentation results

system. The flexible nature of the architecture permits different design alternatives, thus supporting these properties at different levels.

Importantly, the introduction of the load balancing layer in support of the scalability of the system enabled a sensible reduction of the degradation of the user-perceived performances. The experimental results highlighted that thanks to the replication of the services that are more impacted by heavy request loads, the system's response time suffers very marginally from the increase in the number of users in the system. Hence, the scalability problem suffered by choreographed systems, as highlighted in Section 3.7, can be solved by (decomposing monoliths to microservices and) replicating the most affected services and also introducing a load balancing layer as proposed in this chapter.

5.4.1 Threats to validity

Some aspects may threaten the validity of both the presented approach and the evaluation results. In the following, we discuss the most important ones.

Internal validity

The experimentation results were obtained by using a simple round-robin as the load-balancing algorithm implemented into the load balancers. Also, the number of selectable microservice instances was fixed at the deployment of the application, without exploiting any dynamic creation or deletion of instances. In this more dynamic setting, the obtained results might have been different. In fact, a more complex load balancing algorithm may lead to a higher overhead, introduced by the dynamic selection of the service instances and by the more intrinsic complexity of the architecture (that would have required a service registry to keep track of

the available instances). On the other side, the balancing of the workload could have led to better results when the number of users was higher.

Moreover, the experimentation was performed only on the use case system: although it focuses specifically on the load balancing and highlights the benefits arising from its usage, the obtained data may be biased by some factors. For instance, the measured performances may have been affected by the deployment setting (although it is realized through a round-robin deployment on the available VMs).

External validity

The architectural style does not account for the other software components required to realize the more complex balancing scenarios mentioned before. In fact, it does not consider the use, e.g., of service registries, service discoveries, or health checkers that allow the dynamic creation of instances and for their selection. However, it is worth noting that these aspect does not influence the architectural style, since the services implementing the discovery and the registry may be introduced in the architecture without affecting its main concerns (realizing coordination and load balancing in a decoupled way).

Chapter 6

Related work

This chapter reports an overview of works related to the topics covered in this thesis. In particular, we report related approaches at the state-of-the-art concerning: (i) enforcement of choreography realizability, (ii) support for the realization of context-aware systems, (iii) approaches to the decomposition of monoliths to microservices, (iv) approaches to the composition of microservices, and (v) architectures for load balancing.

6.1 Choreography realizability and enforcement

Concerning the automated synthesis of choreographies and the choreography enforcement, our work leverages on the approaches presented in previous works [5, 6, 9], in which CDs are introduced and formally defined, and the automated synthesis process is described and shown at work.

In the literature, many approaches have been proposed to deal with the foundational problems of checking choreography realizability, verifying conformance, and enforcing realizability [17, 19, 35, 55, 56, 76]. They propose formal means to address these fundamental aspects of choreographies. However, they are based on different interpretations of the choreography interaction semantics, in terms of both the subset of considered choreography constructs, and the used formal notations. Rather than addressing realizability and conformance of choreography, the goal of this work and of the CHOReVOLUTION project before of this, is to practically relize a choreography development process that comprizes all the development activities, from the modeling to the runtime support, by supporting the reuse of third-party services. In the following, we report the main approaches and tools concerning the synthesis and enforcement of choreographies and the composition of services.

In the work of Gudemann et al. [56], choreography realizability is enforced through the generation of monitors, which act as local controllers interacting with their peers and the rest of the system in order to make peers respect the choreography specification. The notion of monitor is similar to our notion of CDs, since they coordinate the choreography by interacting with the other choreography participants.

Farah et al. [46] address the realizability problem through an a-priori verification techniques using refinement and proof-based formal methods. Their approach is different from ours in that it defines and realizes a notion of peer projection that is correct by construction, avoiding the coordination from the outside and the introduction of additional software entities needed for enforcing realizability such as our CDs. Our approach, however, focuses on realizing a choreography by reusing third-party (and black-box) services rather than generating from scratch the correct peers. Only the application-specific components, together with the CDs, are generated.

Nguyen et al. [94] introduce data support for analysing interaction-based service choreographies. This model uses data-aware interactions as the basic event. The authors propose a top-down development process for data-aware service choreographies by extracting a behavioral skeleton for each choreography participant through projection. In this way, developers have only to complete the skeletons with the business logic needed to realize the distributed application according to the choreography specification. Differently from our approach, which employs BPMN2 choreographies, their focus is on interaction-based service choreographies.

The ASTRO toolset [118] supports the automated composition of services. It aims to compose a service out of a business requirement and the description of available external services. A planner component automatically synthesizes the code of a centralized process that achieves the business requirement by interacting with the available external services. Unlike our approach, ASTRO deals with orchestration-based processes rather than decentralized choreography-based ones.

6.2 Context-aware systems

The concept of variability is widely used in association with Software Product Lines [11, 16, 123]. Several approaches have been introduced to facilitate the development of variable service architectures in the context of service-oriented

SPLs [25, 86, 104, 113, 116]. However, they mainly focus on the architectural and development aspects of variability, without concerning the context-aware runtime adaptation of the service composition. In the following, we report the main approaches and frameworks that deal with the realization of dynamically adaptable context-aware SOA systems.

Yu et al. [125] presented MoDAR, an approach to the development of dynamically adaptive service-based systems. The approach considers the development of two system models: a simplified business process (base model) and a variable model, which consists of a set of rules that drive the dynamic behavior of the system. A model-driven platform supports the semi-automatic transformation of the code into an executable system. Our approach is similar in that their base model is an underspecified model of the system, while (a part of) our variable part is defined through variants, and selection functions drive the dynamic behavior. However, we are more focussed on the context-awareness adaptation and we explicitly consider the context by defining its model.

Cubo et al. [38] proposed an extension of DAMASCO, a framework for the service discovery and composition, by managing the variability of services and context during the service composition at runtime. They describe the service interfaces through a context profile that describes also the context information, and they use *feature models* in order to specify the service variability. The composition consider the context features in the model to find matching services with the adequate features, thus realizing adaptation. Differently from our approach, Cubo et al. consider the context changes during the composition through discovery means, while our adaptation is entirely performed at runtime with dynamic service selection according to the runtime conditions. Moreover, while they assume that context information is inferred by the client requests, we make use of acquisition functions in order to get context information also from outside the system.

Murguzur et al. [90] introduce LateVa. In this framework, runtime variability is achieved through the definition of a base process model and process fragments. A fragment describes a single variant realization for each of the variation point defined into the base model. The latter is also annotated with endpoints from which context data is gathered. A variability model defines the variants and their context data mappings as feature models. At runtime, the fragment selector service searches for an available fragment in a model repository that can realize the required features. This approach tackles the adaptation challenge in a similar way with respect to our solution for the task-flow-level adaptation – through the

definition of variants that are associated to variation points. However, we also explicitly consider the message and participant-level adaptation.

Bucchiarone et al.

De Sanctis et al. [40] presented an approach for designing and allowing the dynamic context-aware adaptation of service-based applications. Here, the adaptation since the design time and allows dividing the adaptation and the application logic in two separate models. This work leverages their previous works [30, 31], in which abstract activities are defined at design time and are then refined at runtime when adaptation is performed through the continuous integration of new services. Unlike their approach, we avoid to model the adaptation process since the adaptation is performed by choosing adaptable entities among a well-defined set of candidates.

Riccobene et al. [105] show SCA-ASM, a formal framework for modeling and executing service-oriented applications that are able to monitor and react to environmental changes. The approach exploits multi-agents that execute a distributed MAPE-K loop for monitoring the context, planning and execution adaptation both at architectural and at behavioral level. MAPE-K loops are used also in the work presented by Mongiello et al. [87], in which a service-oriented architecture is built at runtime by defining a metamodel based on knowledge graphs to model information about the environment around the user. Our approach differs from these since our adaptation strategy evaluates the context on demand without executing a control loop.

Calinescu and Rafiq [34] describe a method to automatically synthesize intelligent service proxies. They enable self-adaptation of a service-based system through the runtime selection of suitable participants in a workflow. Differently from our approach, they focus the selection to keep satisfying the high-level requirements of the system, dynamically selecting the services by means of online learning. In contrast, we focus the service selection on functional aspects rather than on non-functional requirements satisfaction.

De Prado et al. [52] provide a scalable event-driven context-aware SOA architecture that exploits data obtained from IoT devices and offers context-aware REST services to the user. The approach described by the authors exploit an enterprise service bus that incorporate data coming from IoT devices and allows the communication between all involved agents. The context is managed by a context broker and a context DB, which keep the context information updated and provide the context information to the REST services. This approach focuses on the collection of context data by also using complex event processing, while we

gather context information from business messages and exploit them on demand at runtime when selecting services.

6.3 Decomposition into microservices

The body of knowledge on this topic is wide and covers several aspects. In this section, we only report those that are more similar and closely related to the approach presented in this thesis, by considering the main common characteristics: inputs required by the approaches, usage of static code analysis, graph-based clustering, data-driven decomposition, and the leveraging of layered architectures.

Discussing the *input requirements*, some researchers using a top-down approach have involved the users to manually provide high-level domain information, such as domain models, data-flow diagrams, or use cases [36, 59, 81]. Similarly, Levcovitz et al. [78] manually categorized the database tables, established a dependency graph of the code, and extracted the microservices from the bottom up [49]. In contrast, our approach is not dependent on any manual input effort, system models, and/or user's system knowledge as it requires the monolith's code only.

Several researchers applied SCA to exploit the semantic information for clustering using proximity measures or topic modeling [13, 28, 68, 84, 102, 108, 117]. In contrast, we perform SCA to understand the pre-existing system's architecture to identify the structural elements (i.e., classes, methods, etc.) and the relationships between them (i.e., method calls, class inheritance, etc.), like, e.g., [45, 71, 74, 82, 98, 111, 117].

Representing the structural elements of the original software system in a processable format, such as trees or graphs, is a widely exploited strategy, as *graph-based clustering* may lead toward the identification of microservice candidates [82]. Having a graph-based representation, some approaches [28, 82, 117] applied the Louvain algorithm [24] to find candidates to increase modularity value. Gysel et al. [59] in their visualization tool of Service Cutter applied Girvan–Newman's algorithm [53] to find communities. Mazlami et al. [84] designed a graph-cutting algorithm to generate microservices. Kamimura et al. [68] perform graph clustering to find features based on program groups and data. In [103] four self-developed graph-based algorithms are used to generate microservices. Tyszberowicz et al. [119] represent a monolithic system in a bipartite graph and then use the shortest path to find nodes closer to each other to find density-based clusters. In contrast to the aforementioned approaches, we do not rely solely

on graph clustering to find candidates. We apply the Louvain algorithm on two different graph-based representations of the system to find highly cohesive communities of nodes and clusters of domain entities. Instead of considering those clusters as candidate microservices, they are used as input in the optimization phase to obtain high-cohesive and low-coupled microservices.

The *data-driven* approaches are primarily top-down or hybrid, in which the database is partitioned or the data-flow graphs of business logic are analyzed [65]. Romani et al. [108] have applied semantic analysis on the database schema to find clusters, which are then mapped to topics for microservice identification. They do not form any graph representation of the extracted information from the database schema, however, they relate to us by focusing on the persistence layer and persisted domain entities. Li et al. [81] extended the dataflow-driven semi-automatic decomposition top-down approach of Chen et al. [36]. They map the business logic's code toward the data storage or the persistence layer. They build a virtual data flow from the manual input to apply an algorithm to find clusters or modules as microservices. However, differently from our approach, they require manual intervention and system knowledge to analyze the business logic and use cases and build the data-flow diagrams.

Some approaches leverage systems' layered architectures [106] to identify microservices. Trabelsi et al. [117] extend the work in [2] and classify the source code classes into three layers and identify three service types: entity, application, and utility. After building a weighted graph representation of the system, they apply the Louvain algorithm to group classes of each layer into services according to their static relationships. Then, a fuzzy clustering algorithm is applied to cluster services across different layers into candidate microservices. In [128], authors assign each class to its related layer. By using two different similarity metrics that favor a vertical decomposition, a hierarchical clustering algorithm builds a dendrogram that is then suitably cut into a set of candidate microservices. In contrast, we exploit layers to build the graph representation and assign weights to the graph's edges. Both clustering and optimization are performed on the graph without considering the system layers explicitly, and allow methods from all the layers to be included in each microservice. Vertical decomposition is obtained through the inter-relationships between methods and methods and entities.

Finally, differently from all the approaches reported above, we represent and decompose the system at the method-level resolution, even splitting classes if needed, rather than considering classes as atomic units.

6.4 Microservices composition approaches

Oberhauser presents Microflows [99], a lightweight and automated approach for the orchestration of semantically-annotated microservices through agent-based clients. Microservices are provided with semantic metadata in support for their automated invocations. They are mapped to nodes and then represented in a graph. Clients act as agents and execute a workflow by planning the shortest path into the graph.

Yahia et al. [21] propose Medley, an event-driven microservice composition platform. It is based on a DSL for producing an orchestration description without the need to focus on the communication issues. The code is compiled into a lower-level code and run on a event-based platform. Medley handles the service adaptation according to service availability and supports horizontal scaling among clusters of nodes.

The work by Monteiro et al. [88] aims to overcome the limitations of the approaches proposed by Oberhauser and Yahia due to the dynamic location of microservices. They propose the use of declarative business processes, which focus on what should be done in order to achieve a business goal rather than all possible alternative flows to be followed. The orchestration of microservices is realized through the Beethoven platform. It is realized through a layered architecture, whose core is the orchestration layer composed of an event bus and an event processor that processes messages, handles tasks, and balances the workload. The orchestration is modeled through a specific DSL.

Gutierrez-Fernandes et al. [58] propose the use of process engines as microservice platform instead of using orchestrators when the business logic of microservices involves complex workflows. Business process languages such as BPMN can be used as a high-level language to define the microservices behavior and their message passing. To this extent, Valderas et al.[120] defined a microservice composition approach as a choreography of BPMN fragments. Their approach considers the developement of a BPMN2 diagram representing the “big picture” of the system, which is then splitted into fragments, each representing the behavior of a microservice. A BPMN engine is used in order to deploy and execute the microservices, while a message bus allows the communication among them.

Sun et al. [112] propose an approach to model variable microservice-based system by using VxBPMN4MS, an extension of BPMN for supporting variability. Microservices invocations are explicitly modeled in the business process and are manipulated in the composition. After the modeling phase, the business pro-

cess is automatically transformed in a microservice composition framework called AMS implementing the business logic. The composition framework monitors the microservice execution, handling variability and runtime exceptions.

Our approach leverages on a similar technique (BPMN2 diagrams) for the modeling of the microservices choreography. However, while the reported approaches consider the microservices internal behavior, we only focus on their interaction protocols, thus considering (micro)services as black-box entities that have to be externally coordinated. Moreover, none of these approaches considered explicitly dependability aspects since they mainly focused on development and composition issues. In contrast, our layered architecture leverages on the automated choreography synthesis approach in [9] and provides specific support for scalability in microservice composition through the load balancing layer.

Besides the aforementioned Medley and Beethoven, other approaches for the microservice composition considered the use of a specifically-developed programming language. In particular, the Jolie web-service oriented programming language[1] provides support for microservices development by shifting the focus from the computation and frameworks to the communication and dependencies among microservices [57]. By taking inspiration from WS-BPEL, Jolie allows defining the interfaces and the workflow of each microservice, separating the definition of the behavior of a microservice from its deployment.

6.5 Architectures for microservices load balancing

Many works in the state-of-the-art about load balancing are specifically aimed at analyzing techniques and algorithms for efficiently performing the service selection process and optimizing the load balancing among the system computational resources [3]. These are fundamental aspects since an efficient selection process speeds up the communication among microservices, while an optimal balancing strategy guarantees high performances when the system load increases. However, these aspects are out of the scope of this work, since we focused on the architectural aspect rather than on a specific load balancing algorithm.

In [83], the author briefly discusses different load balancing strategies. Beyond the server-side and the client-side strategies, the author portrays an “external” load balancing strategy. Here, clients can communicate directly with the server, after they have requested to an external load balancer which instance (i.e., which URI) they have to connect to. In this external approach, there are neither extra

[1] https://www.jolie-lang.org

ops nor proxified interactions, while the client complexity is smaller with respect to a client-side approach, and the scalability and performances are improved with respect to a server-side approach. However, the external load balancer still represents a single point of failure, while clients have to still manage some complexity (i.e., the communication with the external load balancer) and they must be trusted.

In [80] Baker Street[2], a framework for realizing client-side load balancing in microservice systems, is presented. By following the architectural style of client-side load balancing, each client service is provided with an instance of a local load balancer that handles its local traffic. In order to manage the information about the availability and location of the microservice instances, each instance has (i) a health checker that registers the microservice to a global discovery service, and (ii) a routing system that owns and updates a local list of the service instances. When a microservice has to send a message, its routing systems proxify the communication by routing the message to one of the available target microservice instances. As argued by the author, while this approach owns all the advantages of a client-side load balancing, it may distribute the load in an uneven way since each load balancer may randomly route the traffic towards the same microservice instances. The use of an hybrid approach, as envisioned in our architecture, would allow the mitigation of this issue.

Niu et al. [97] argue that besides the overhead introduced by load balancers that proxify the interaction among microservices, load balancing techniques ignore the competition for shared microservices among different chains of requests that connect a microservice to another. In order to overcome this issue, they present a chained-oriented load balancing algorithm (COLBA), which balances the microservice instances according to the request chains, isolating microservices across them. Their approach allows steering requests only through microservices that belong to the same chain. They address the challenge by modeling load balancing as a non-cooperative game and employ a convex optimization technique to obtain an approximated optimal solution to the problem. Similarly, in [126] is pointed out that the load on a microservice instance directly depends on the load distributed on its predecessor instances in the request chain. Considering this, the relationships between microservice instances are modeled as a directed graph, and a QoS-aware load balancing problem is formulated. However, obtaining such kinds of load balancing requires the knowledge of all the request chains across the microservices, besides the fact that the load balancing would require a

[2]http://bakerstreet.io/

complete update if some microservice changes and the request chains change as a consequence. In contrast, our load balancing architecture is agnostic with respect to the system's topology, and it is robust with respect to the update, addition, or removal of microservices. Moreover, similar advanced balancing techniques can be still implemented as upgrades of the load balancers' internal logic.

Chapter 7

Conclusions and future work

This thesis reported the work done in the scope of Service Oriented Architectures (SOAs), in particular on the composition of services and the decomposition of monoliths into microservices. The thesis presented a three-fold contribution:

- It enhanced the approach at the state-of-the-art for the composition and coordination of services into choreographies by proposing a solution for the realization of context-aware choreographies;
- It proposed a novel fully-automated approach for the decomposition of monolithic systems into microservices;
- It proposed an architectural style enabling the load balancing of microservices in choreography-based systems.

All the presented approaches mentioned above have been evaluated by also running experiments on use cases and the results obtained were satisfactory.

The short-term future work concerns further experimentations on all the solutions presented in the thesis, in particular by leveraging a use case, possibly from an industrial real-world case study, capable of showing the context-awareness, decomposition, and load-balancing needs together.

Then, we plan to integrate the generation of the new software artifacts supporting context awareness and load-balancing directly within the synthesis process realized in the previous works [4–6, 9, 10].

Finally, we plan to address the limitations of the approach to the decomposion. In particular, we plan to allow experts to provide their inputs during or after the community detection phase as a further refinement step. This will improve the quality of the obtained domain contexts, overcome the limitations concerning the

clustering phase, and take into account possible application-specific requirements. Moreover, we plan to improve the analysis phase by considering dynamic analysis to optimize the overhead of inter-MS communications. This will also call for more accurate metrics to be used as the objective function of the optimization phase. We also plan to extend our tool by using machine learning techniques to classify classes more accurately, especially when the system is large and complex.

List of Publications

G. Filippone, M. Autili, and M. Tivoli. Towards the synthesis of context-aware choreographies. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 197–200. IEEE, 2020.

G. Filippone, M. Autili, F. Rossi and M. Tivoli. Migration of Monoliths through the Synthesis of Microservices using Combinatorial Optimization. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 144-147. IEEE, 2021.

G. Filippone, M. Autili and M. Tivoli. Synthesis of context-aware business-to-business processes for location-based services through choreographies. *Journal of Software: Evolution and Process*, 34(10), 2022.

G. Filippone, C. Pompilio, M.Autili and M.Tivoli. An architectural style for scalable choreography-based microservice-oriented distributed systems. *Computing.* 2022.

G. Filippone, N. Q. Mehmood, M. Autili, F. Rossi, M. Tivoli. From monolithic to microservice architecture: an automated approach based on graph clustering and combinatorial optimization. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, pages 45–57. IEEE, 2023.

References

[1] M. L. Abbott and M. T. Fisher. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Addison-Wesley Professional, Boston, 2015.

[2] M. Abdellatif, R. Tighilt, N. Moha, H. Mili, G. El Boussaidi, J. Privat, and Y.-G. Guéhéneuc. A type-sensitive service identification approach for legacy-to-soa migration. In *International Conference on Service-Oriented Computing*, pages 476–491. Springer, 2020.

[3] S. Afzal and K. Ganesh. Load balancing in cloud computing - a hierarchical taxonomical classification. *Journal of Cloud Computing*, 8, 12 2019.

[4] M. Autili, A. Di Salle, F. Gallo, C. Pompilio, and M. Tivoli. Model-driven adaptation of service choreographies. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1441–1450, New York, NY, USA, 2018. Association for Computing Machinery.

[5] M. Autili, A. Di Salle, F. Gallo, C. Pompilio, and M. Tivoli. Aiding the realization of service-oriented distributed systems. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 1701–1710, New York, NY, USA, 2019. Association for Computing Machinery.

[6] M. Autili, P. Inverardi, and M. Tivoli. Choreography realizability enforcement through the automatic synthesis of distributed coordination delegates. *Science of Computer Programming*, 160:3–29, 2018.

[7] M. Autili, A. Perucci, and L. De Lauretis. *A Hybrid Approach to Microservices Load Balancing*, pages 249–269. Springer International Publishing, Cham, 2020.

[8] M. Autili, A. Perucci, L. Leite, M. Tivoli, F. Kon, and A. Di Salle. Highly collaborative distributed systems: Synthesis and enactment at work. *Concurrency and Computation: Practice and Experience*, 33(6):e6039, 2021.

[9] M. Autili, A. D. Salle, F. Gallo, C. Pompilio, and M. Tivoli. Chorevolution: Service choreography in practice. *Science of Computer Programming*, 197:102498, 2020.

[10] M. Autili and M. Tivoli. Distributed enforcement of service choreographies. In J. Cámara and J. Proença, editors, *Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems, FOCLASA 2014, Rome, Italy, 6th September 2014*, volume 175 of *EPTCS*, pages 18–35, 2014.

[11] M. A. Babar, L. Chen, and F. Shull. Managing variability in software product lines. *IEEE Software*, 27(3):89–91, 94, 2010.

[12] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Migrating to cloud-native architectures using microservices: An experience report. In A. Celesti and P. Leitner, editors, *Advances in Service-Oriented and Cloud Computing*, pages 201–215, Cham, 2016. Springer International Publishing.

[13] L. Baresi, M. Garriga, and A. D. Renzis. Microservices identification through interface analysis. In *European Conference on Service-Oriented and Cloud Computing*, pages 19–33. Springer, 2017.

[14] A. Barker, P. Besana, D. Robertson, and J. B. Weissman. The benefits of service choreography for data-intensive computing. In T. Rauber, G. Rünger, E. Jeannot, and S. Jha, editors, *Proceedings of the 7th international workshop on Challenges of large applications in distributed environments, CLADE@HPDC 2009, Garching near Munich, Germany, June 10, 2009*, pages 1–10, 2009.

[15] E. R. Barnes. An algorithm for partitioning the nodes of a graph. *SIAM Journal on Algebraic Discrete Methods*, 3(4):541–550, 1982.

[16] L. Bastida, F. J. Nieto, and R. Tola. Context-aware service composition: A methodology and a case study. In *Proceedings of the 2nd International Workshop on Systems Development in SOA Environments*, pages 19–24, New York, NY, USA, 2008. Association for Computing Machinery.

[17] S. Basu and T. Bultan. Choreography conformance via synchronizability. In *Proceedings of the 20th international conference on World wide web*, pages 795–804, 2011.

[18] S. Basu and T. Bultan. Automated choreography repair. In *International Conference on Fundamental Approaches to Software Engineering*, pages 13–30. Springer, 2016.

[19] S. Basu, T. Bultan, and M. Ouederni. Deciding choreography realizability. *Acm Sigplan Notices*, 47(1):191–202, 2012.

[20] C. Bauer and A. Novotny. A consolidated view of context for intelligent systems. *J. Ambient Intell. Smart Environ.*, 9(4):377–393, 2017.

[21] E. Ben Hadj Yahia, L. Réveillère, Y.-D. Bromberg, R. Chevalier, and A. Cadot. Medley: An event-driven lightweight platform for service composition. In A. Bozzon, P. Cudre-Maroux, and C. Pautasso, editors, *Web Engineering*, pages 3–20, Cham, 2016. Springer International Publishing.

[22] D. Bertsimas, C.-P. Teo, and R. Vohra. Analysis of lp relaxations for multiway and multicut problems. *Networks*, 34(2):102–114, 1999.

[23] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni. A survey of context modelling and reasoning techniques. *Pervasive Mob. Comput.*, 6(2):161–180, 2010.

[24] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[25] N. Boffoli, M. Cimitile, F. M. Maggi, and G. Visaggio. Managing soa system variation through business process lines and process oriented development. In *Workshop on Service-Oriented Architectures and Software Product Lines (SOAPL)*, pages 61–68, 2009.

[26] A. B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203, 2000.

[27] A. Bouguettaya, M. P. Singh, M. N. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, B. Benatallah, B. Medjahed, M. Ouzzani, F. Casati, X. Liu, H. Wang, D. Georgakopoulos, L. Chen, S. Nepal, Z. Malik, A. Erradi, Y. Wang, M. B. Blake, S. Dustdar, F. Leymann, and M. P. Papazoglou. A service computing manifesto: the next 10 years. *Communication ACM*, 60(4):64–72, 2017.

[28] M. Brito, J. Cunha, and J. Saraiva. Identification of microservices from monolithic applications through topic modelling. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1409–1418, 2021.

[29] A. Bucchiarone, C. Cappiello, E. D. Nitto, B. Pernici, and A. Sandonini. A variable context model for adaptable service-based applications. *Int. J. Adapt. Resilient Auton. Syst.*, 3(3):35–53, 2012.

[30] A. Bucchiarone, M. De Sanctis, A. Marconi, M. Pistore, and P. Traverso. Design for adaptation of distributed service-based systems. In A. Barros, D. Grigori, N. C. Narendra, and H. K. Dam, editors, *Service-Oriented Computing*, pages 383–393, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[31] A. Bucchiarone, M. De Sanctis, A. Marconi, M. Pistore, and P. Traverso. Incremental composition for adaptive by-design service based systems. In S. Reiff-Marganiec, editor, *IEEE International Conference on Web Services, ICWS 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 236–243. IEEE Computer Society, 2016.

[32] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara. From monolithic to microservices: An experience report from the banking domain. *IEEE Software*, 35(3):50–55, 2018.

[33] A. Bucchiarone, K. Soysal, and C. Guidi. A model-driven approach towards automatic migration to microservices. In J.-M. Bruel, M. Mazzara, and B. Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 15–36, Cham, 2020. Springer International Publishing.

[34] R. Calinescu and Y. Rafiq. Using intelligent proxies to develop self-adaptive service-based systems. In *Seventh International Symposium on Theoretical Aspects of Software Engineering, TASE 2013, 1-3 July 2013, Birmingham, UK*, pages 131–134. IEEE Computer Society, 2013.

[35] M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274, 2013.

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

[37] F. Corradini, F. Fornari, A. Polini, B. Re, and F. Tiezzi. A formal approach to modeling and verification of business process collaborations. *Science of Computer Programming*, 166:35–70, 2018.

[38] J. Cubo, N. Gamez, L. Fuentes, and E. Pimentel. Composition and self-adaptation of service-based systems with feature models. In J. Favaro and M. Morisio, editors, *Safe and Secure Software Reuse*, pages 326–342, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[39] K. Dar, A. Taherkordi, R. Rouvoy, and F. Eliassen. Adaptable service composition for very-large-scale internet of things systems. In D. M. Eyers, editor, *Proceedings of the 8th Middleware Doctoral Symposium of the 12th ACM/IFIP/USENIX International Middleware Conference, Lisbon, Portugal, 12 December 2011*, pages 2:1–2:6, 2011.

[40] M. De Sanctis, A. Bucchiarone, and A. Marconi. Dynamic adaptation of service-based applications: a design for adaptation approach. *J. Internet Serv. Appl.*, 11(1):2, 2020.

[41] A. K. Dey. Understanding and using context. *Personal Ubiquitous Computing*, 5(1):4–7, 2001.

[42] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[43] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina. Microservices: How to make your application scale. In A. K. Petrenko and A. Voronkov, editors, *Perspectives of System Informatics*, pages 95–104, Cham, 2018. Springer International Publishing.

[44] J. Erickson and K. Siau. Web services, service-oriented computing, and service-oriented architecture: Separating hype from reality. *J. Database Manag.*, 19(3):42–54, 2008.

[45] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, and R. Casallas. Towards the understanding and evolution of monolithic

applications as microservices. In *2016 XLII Latin American computing conference (CLEI)*, pages 1–11. IEEE, 2016.

[46] Z. Farah, Y. Aït-Ameur, M. Ouederni, and K. Tari. A correct-by-construction model for asynchronously communicating systems. *International Journal on Software Tools for Technology Transfer*, 19(4):465–485, August 2017.

[47] R. T. Fielding. Architectural styles and the design of network-based software architectures. 2000. *PhD Thesis, University of California, Irvine*, 162, 2000.

[48] M. Fowler. Monolith first, 2015. https://martinfowler.com/bliki/MonolithFirst.html (Last accessed on April 2023).

[49] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner. 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*, pages 128–141. Springer, 2018.

[50] T. Furtado, E. Francesquini, N. Lago, and F. Kon. A middleware for reflective web service choreographies on the cloud. In F. M. Costa and A. Andersen, editors, *Proceedings of the 13th Workshop on Adaptive and Reflective Middleware, ARM@Middleware 2014, Bordeaux, France, December 8-12, 2014*, pages 9:1–9:6, 2014.

[51] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic. Obtaining ground-truth software architectures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 901–910, 2013.

[52] A. García De Prado, G. Ortiz, and J. Boubeta-Puig. Cared-soa: A context-aware event-driven service-oriented architecture. *IEEE Access*, 5:4646–4663, 2017.

[53] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[54] A. Gorbenko, V. Kharchenko, and A. Romanovsky. *Using Inherent Service Redundancy and Diversity to Ensure Web Services Dependability*, page 324–341. Springer-Verlag, Berlin, Heidelberg, 2009.

[55] G. Gössler and G. Salaün. Realizability of choreographies for services interacting asynchronously. In *International Workshop on Formal Aspects of Component Software*, pages 151–167. Springer, 2011.

[56] M. Güdemann, G. Salaün, and M. Ouederni. Counterexample guided synthesis of monitors for realizability enforcement. In S. Chakraborty and M. Mukund, editors, *Automated Technology for Verification and Analysis*, pages 238–253, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[57] C. Guidi, I. Lanese, M. Mazzara, and F. Montesi. *Microservices: A Language-Based Approach*, pages 217–225. Springer International Publishing, Cham, 2017.

[58] A. M. Gutiérrez-Fernández, M. Resinas, and A. Ruiz-Cortés. Redefining a process engine as a microservice platform. In M. Dumas and M. Fantinato, editors, *Business Process Management Workshops*, pages 252–263, Cham, 2017. Springer International Publishing.

[59] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann. Service cutter: A systematic approach to service decomposition. In M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, editors, *Service-Oriented and Cloud Computing*, pages 185–200, Cham, 2016. Springer International Publishing.

[60] L. Han, J. P. Salomaa, J. Ma, and K. Yu. Research on context-aware mobile computing. In *22nd International Conference on Advanced Information Networking and Applications, AINA 2008, Workshops Proceedings, GinoWan, Okinawa, Japan, March 25-28, 2008*, pages 24–30. IEEE Computer Society, 2008.

[61] S. Hassan, N. Ali, and R. Bahsoon. Microservice ambients: An architectural meta-modelling approach for microservice granularity. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 1–10. IEEE, 2017.

[62] S. Hassan, R. Bahsoon, and R. Kazman. Microservice transition and its granularity problem: A systematic mapping study. *Software: Practice and Experience*, 50(9):1651–1681, 2020.

[63] A. Homay, M. de Sousa, A. Zoitl, and M. Wollschlaeger. Service granularity in industrial automation and control systems. In *2020 25th IEEE*

International Conference on Emerging Technologies and Factory Automation (ETFA), volume 1, pages 132–139. IEEE, 2020.

[64] R. S. Huergo, P. F. Pires, and F. C. Delicato. Mdcsim: A method and a tool to identify services. *IT Convergence Practice*, 2(4):1–27, 2014.

[65] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng. Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, 47(5):987–1007, 2021.

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

[67] M. Kalske, N. Mäkitalo, and T. Mikkonen. Challenges when moving from monolith to microservice architecture. In I. Garrigós and M. Wimmer, editors, *Current Trends in Web Engineering*, pages 32–47, Cham, 2018. Springer International Publishing.

[68] M. Kamimura, K. Yano, T. Hatano, and A. Matsuo. Extracting candidates of microservices from monolithic application code. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 571–580. IEEE, 2018.

[69] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[70] S. Klock, J. M. E. Van Der Werf, J. P. Guelen, and S. Jansen. Workload-based clustering of coherent feature sets in microservice architectures. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 11–20. IEEE, 2017.

[71] H. Knoche and W. Hasselbring. Using microservices for legacy software modernization. *IEEE Software*, 35(3):44–49, 2018.

[72] M. Kogias, R. Iyer, and E. Bugnion. Bypassing the load balancer without regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 193–207, New York, NY, USA, 2020. Association for Computing Machinery.

[73] M. Konersmann, A. Kaplan, T. Kühn, R. Heinrich, A. Koziolek, R. Reussner, J. Jürjens, M. al Doori, N. Boltz, M. Ehl, D. Fuchs, K. Groser, S. Hahner, J. Keim, M. Lohr, T. Sağlam, S. Schulz, and J.-P. Töberg. Evaluation methods and replicability of software architecture research objects. In

2022 IEEE 19th International Conference on Software Architecture (ICSA), pages 157–168, 2022.

[74] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger. Microservice decomposition via static and dynamic analysis of the monolith. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 9–16. IEEE, 2020.

[75] S. Lane, A. Bucchiarone, and I. Richardson. Soadapt: A process reference model for developing adaptable service-based applications. *Information and Software Technology*, 54(3):299–316, 2012.

[76] I. Lanese, F. Montesi, and G. Zavattaro. *The Evolution of Jolie*, pages 506–521. Springer International Publishing, Cham, 2015.

[77] L. A. F. Leite, G. A. Oliva, G. M. Nogueira, M. A. Gerosa, F. Kon, and D. S. Milojicic. A systematic literature review of service choreography adaptation. *Serv. Oriented Comput. Appl.*, 7(3):199–216, 2013.

[78] A. Levcovitz, R. Terra, and M. T. Valente. Towards a technique for extracting microservices from monolithic enterprise systems. *arXiv preprint arXiv:1605.03175*, 2016.

[79] J. Lewis and M. Fowler. Microservices a definition of this new architectural term, 2014. https://martinfowler.com/articles/microservices.html (Last accessed on April 2023).

[80] R. Li. Baker Street: Avoiding Bottlenecks with a Client-Side Load Balancer for Microservices, 2015. https://thenewstack.io/baker-street-avoiding-bottlenecks-with-a-client-side-load-balancer-for-microservices/ (Last accessed on June 2022).

[81] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, and Z. Shan. A dataflow-driven approach to identifying microservices from monolithic applications. *Journal of Systems and Software*, 157:110380, 2019.

[82] Z. Li, C. Shang, J. Wu, and Y. Li. Microservice extraction based on knowledge graph from monolithic applications. *Information and Software Technology*, 150:106992, 2022.

[83] Malcom. Load-balancing strategies, 2018. https://www.beyondthelines.net/computing/load-balancing-strategies/ (Last accessed on June 2022).

[84] G. Mazlami, J. Cito, and P. Leitner. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531. IEEE, 2017.

[85] M. Mazzara, A. Bucchiarone, N. Dragoni, and V. Rivera. *Size Matters: Microservices Research and Applications*, pages 29–42. Springer International Publishing, Cham, 2020.

[86] B. Mohabbati, M. Hatala, D. Gašević, M. Asadi, and M. Bošković. Development and configuration of service-oriented systems families. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1606–1613. Association for Computing Machinery, 2011.

[87] M. Mongiello, T. D. Noia, F. Nocera, and E. D. Sciascio. Case-based reasoning and knowledge-graph based metamodel for runtime adaptive architectural modeling. In S. Ossowski, editor, *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, pages 1323–1328. ACM, 2016.

[88] D. Monteiro, P. H. M. Maia, L. S. Rocha, and N. C. Mendonça. Building orchestrated microservice systems using declarative business processes. *Service Oriented Computing and Applications*, 14(4):243–268, 2020.

[89] F. Montesi, M. Peressotti, and V. Picotti. Sliceable monolith: Monolith first, microservices later. In *2021 IEEE International Conference on Services Computing (SCC)*, pages 364–366, 2021.

[90] A. Murguzur, S. Trujillo, H.-L. Truong, S. Dustdar, O. Ortiz, and G. Sagardui. Run-time variability for context-aware smart workflows. *IEEE Software*, 32(3):52–60, 2015.

[91] M. E. Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

[92] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[93] S. Newman. *Building microservices.* " O'Reilly Media, Inc.", 2021.

[94] H. N. Nguyen, P. Poizat, and F. Zaïdi. Automatic skeleton generation for data-aware service choreographies. In *2013 IEEE 24th International*

Symposium on Software Reliability Engineering (ISSRE), pages 320–329, 2013.

[95] T. Nguyen, A. Colman, M. Adeel Talib, and J. Han. Managing service variability: State of the art and open issues. pages 165–173, 01 2011.

[96] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3-4):313–341, 2008.

[97] Y. Niu, F. Liu, and Z. Li. Load balancing across microservices. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 198–206, 2018.

[98] L. Nunes, N. Santos, and A. Rito Silva. From a monolith to a microservices architecture: An approach based on transactional contexts. In *European Conference on Software Architecture*, pages 37–52. Springer, 2019.

[99] R. Oberhauser. Microflows: Lightweight automated planning and enactment of workflows comprising semantically-annotated microservices. In *Sixth International Symposium on Business Modeling and Software Design (BMSD 2016), Volume 1*, pages 134–143, 2016.

[100] M. Ozkaya. Monolith First Approach Before Moving to Microservices, 2023. https://medium.com/design-microservices-architecture-with-patterns/monolith-first-approach-before-moving-to-microservices-da969be8bf7c (Last accessed on April 2023).

[101] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[102] I. Pigazzini, F. Arcelli Fontana, and A. Maggioni. Tool support for the migration to microservice architecture: An industrial case study. In T. Bures, L. Duchien, and P. Inverardi, editors, *Software Architecture*, pages 247–263, Cham, 2019. Springer International Publishing.

[103] V. Raj and S. Ravichandra. A service graph based extraction of microservices from monolith services of service-oriented architecture. *Software: Practice and Experience*, 2022.

[104] M. Razavian and R. Khosravi. Modeling variability in the component and connector view of architecture using uml. In *2008 IEEE/ACS International Conference on Computer Systems and Applications*, pages 801–809, 2008.

[105] E. Riccobene and P. Scandurra. Formal modeling self-adaptive service-oriented applications. In R. L. Wainwright, J. M. Corchado, A. Bechini, and J. Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1704–1710. ACM, 2015.

[106] M. Richards. *Software Architecture Patterns*. O'Reilly Media, Inc., 2015.

[107] C. Richardson. *Microservices Patterns*. Manning Publications, 2018.

[108] Y. Romani, O. Tibermacine, and C. Tibermacine. Towards migrating legacy software systems to microservice-based architectures: a data-centric process for microservice identification. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 15–19. IEEE, 2022.

[109] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Wirel. Commun.*, 8(4):10–17, 2001.

[110] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *1994 First Workshop on Mobile Computing Systems and Applications*, pages 85–90, 1994.

[111] A. Selmadji, A.-D. Seriai, H. L. Bouziane, R. O. Mahamane, P. Zaragoza, and C. Dony. From monolithic architecture style to microservice one based on a semi-automatic approach. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 157–168. IEEE, 2020.

[112] C.-a. Sun, J. Wang, Z. Liu, and Y. Han. A variability-enabling and model-driven approach to adaptive microservice-based systems. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 968–973, 2021.

[113] H. Sun, R. R. Lutz, and S. Basu. Product-line-based requirements customization for web service compositions. In *Proceedings of the 13th International Software Product Line Conference*, pages 141–150, USA, 2009. Carnegie Mellon University.

[114] D. Taibi, V. Lenarduzzi, and C. Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[115] D. Taibi, V. Lenarduzzi, and C. Pahl. Architectural patterns for microservices: a systematic mapping study. In *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*, pages 221–232, 2018.

[116] Y. Topaloglu and R. Capilla. Modeling the variability of web services from a pattern point of view. In *Web Services*, pages 128–138, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[117] I. Trabelsi, M. Abdellatif, A. Abubaker, N. Moha, S. Mosser, S. Ebrahimi-Kahou, and Y.-G. Guéhéneuc. From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis. *Journal of Software: Evolution and Process*, 2022.

[118] M. Trainotti, M. Pistore, G. Calabrese, G. Zacco, G. Lucchese, F. Barbon, P. Bertoli, and P. Traverso. Astro: Supporting composition and execution of web services. In B. Benatallah, F. Casati, and P. Traverso, editors, *Service-Oriented Computing - ICSOC 2005*, pages 495–501. Springer Berlin Heidelberg, 2005.

[119] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu. Identifying microservices using functional decomposition. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pages 50–65. Springer, 2018.

[120] P. Valderas, V. Torres, and V. Pelechano. A microservice composition approach based on the choreography of bpmn fragments. *Information and Software Technology*, 127:106370, 2020.

[121] H. Vincent, V. Issarny, N. Georgantas, E. Francesquini, A. Goldman, and F. Kon. Choreos: scaling choreographies for the internet of the future. In *Middleware'10 Posters and Demos Track*, pages 1–3. 2010.

[122] J. Wade. Brownfield vs. greenfield development: What's the difference in software?, 2018. https://synoptek.com/insights/it-blogs/greenfield-vs-brownfield-software-development/ (Last accessed on April 2023).

[123] D. L. Webber and H. Gomaa. Modeling variability in software product lines with the variation point model. *Science of Computer Programming*, 53(3):305–331, 2004.

[124] Y. Yoon, C. Ye, and H. Jacobsen. A distributed framework for reliable and efficient service choreographies. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 785–794, 2011.

[125] J. Yu, Q. Z. Sheng, and J. K. Y. Swee. Model-driven development of adaptive service-based systems with aspects and rules. In L. Chen, P. Triantafillou, and T. Suel, editors, *Web Information Systems Engineering – WISE 2010*, pages 548–563, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[126] R. Yu, V. T. Kilari, G. Xue, and D. Yang. Load balancing for interdependent iot microservices. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 298–306, 2019.

[127] Z. Zainol and K. Nakata. Generic context ontology modelling: A review and framework. In *2010 2nd International Conference on Computer Technology and Development*, pages 126–130, 2010.

[128] P. Zaragoza, A.-D. Seriai, A. Seriai, A. Shatnawi, and M. Derras. Leveraging the layered architecture for microservice recovery. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 135–145, 2022.

[129] Y. Zhang, B. Liu, L. Dai, K. Chen, and X. Cao. Automated microservice identification in legacy systems with functional and non-functional metrics. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 135–145, 2020.