UNIVERSIDADE ESTADUAL DO CEARÁ
CENTRO DE CIÊNCIAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

DAVI MONTEIRO BARBOSA

BEETHOVEN: AN EVENT-DRIVEN LIGHTWEIGHT PLATFORM FOR MICROSERVICE ORCHESTRATION

FORTALEZA – CEARÁ
2018

DAVI MONTEIRO BARBOSA

BEETHOVEN: AN EVENT-DRIVEN LIGHTWEIGHT PLATFORM FOR MICROSERVICE ORCHESTRATION

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Paulo Henrique Mendes Maia

FORTALEZA – CEARÁ

2018

Dados Internacionais de Catalogação na Publicação

Universidade Estadual do Ceará

Sistema de Bibliotecas

Monteiro Barbosa, Davi.
   Beethoven: An Event-Driven Lightweight Platform for Microservice Orchestration [recurso eletrônico] / Davi Monteiro Barbosa. - 2018.
   1 CD-ROM: il.; 4 ¾ pol.

   CD-ROM contendo o arquivo no formato PDF do trabalho acadêmico com 129 folhas, acondicionado em caixa de DVD Slim (19 x 14 cm x 7 mm).

   Dissertação (mestrado acadêmico) - Universidade Estadual do Ceará, Centro de Ciências e Tecnologia, Mestrado Acadêmico em Ciência da Computação, Fortaleza, 2018.
   Orientação: Prof. Dr. Paulo Henrique Mendes Maia.

   1. Arquitetura Orientada a Eventos. 2. Arquitetura de Referência. 3. Composição de Microservice. 4. Orquestração. I. Título.

DAVI MONTEIRO BARBOSA

BEETHOVEN: AN EVENT-DRIVEN LIGHTWEIGHT PLATFORM FOR MICROSERVICE ORCHESTRATION

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Aprovada em: 9 de Julho de 2018

BANCA EXAMINADORA

This dissertation is dedicated to my mother Már-
cia Stela Andrade Monteiro with love.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Professor Dr. Paulo Henrique Mendes Maia, my research supervisors, for his patient guidance, enthusiastic encouragement, valuable support, and useful critiques of this research work.

I would also like to thank Dr. Nabor das Chagas Mendonça and Dr. Lincoln Souza Rocha, for their advice, constructive recommendations, and assistance during the progress of this dissertation. My grateful thanks are also extended to Rômulo Gadelha de Moura Lima for his help during the publication of the results of this dissertation and to Italo Yeltsin Medeiros Bruno for his support during the conduction of the quasi-experiment.

I would also like to extend my thanks to the State University of Ceará, its faculty, direction, and administration.

Finally, I wish to thank my parents Márcia Stela Andrade Monteiro and Antônio Luiz Carlos Barbosa, my sister Danielle Monteiro Barbosa, and my aunt Maria Valdenia de Sousa for their support and encouragement throughout my study. Words are not enough to express how much I love and admire them. This dissertation is dedicated to them.

“Great things are done by a series of small things brought together.”

(Vincent Van Gogh)

RESUMO

A arquitetura monolítica tradicional é construída como uma unidade lógica única que agrega vários serviços para fornecer funcionalidades de negócios. No entanto, a arquitetura monolítica pode apresentar as seguintes desvantagens: (i) dificuldade de compreender e modificar ao longo do tempo; (ii) dimensionamento ineficiente dos recursos computacionais; e (iii) dificuldade em aplicar pequenas modificações. Neste domínio, a arquitetura de *microservices* propõe uma solução para dimensionar recursos computacionais de forma eficiente e resolver outros problemas presentes na arquitetura monolítica. Embora a arquitetura de *microservices* ofereça inúmeros benefícios, há custos associados à sua adoção, como desafios para executar processos de negócios distribuídos entre diferentes *microservices*. Neste contexto, apesar de existir abordagens recentes para a composição de *microservices*, como Medley e Microflows, essas soluções possuem limitações em lidar com a localização dinâmica de *microservices*, pois exigem um registro prévio dos *microservices* necessários para realizar composições. Além disso, essas soluções não estão disponíveis tanto para a indústria quanto para a academia. Para preencher essa lacuna, esta dissertação propõe Beethoven, uma plataforma leve para composição de *microservices* que é composta de uma arquitetura de referência e uma DSL de orquestração baseada em processos de negócios declarativos. A arquitetura de referência segue uma abordagem orientada a eventos e foi instanciada usando o modelo de atores e o ecossistema fornecidos pelo Spring Cloud Netflix. Para demonstrar a viabilidade da plataforma de Beethoven, foram desenvolvidas duas aplicações de exemplo. Além disso, para investigar a avaliação de desempenho da plataforma, um quasi-experimento controlado foi conduzido. Todos os artefatos produzidos como parte dessa dissertação estão disponíveis no GitHub.

Palavras-chave: Arquitetura Orientada a Eventos. Arquitetura de Referência. Composição de *Microservice*. Orquestração.

ABSTRACT

The traditional monolithic architecture is constructed as a single logic unit that aggregates several services to provide business functionalities. However, monolithic applications may have the following drawbacks: (i) difficult to understand and modify over time; (ii) inefficient dimensioning of computational resources; and (iii) difficulty in applying small modifications. In this realm, the microservices architecture proposes a solution for efficiently scaling computational resources and solving other problems present in the monolithic architecture. Although the microservices architecture offers numerous benefits, there are costs associated with its adoption such as challenges to execute business processes distributed across different microservices. In this context, although there exist some recent approaches for microservice composition, such as Medley and Microflows, these solutions have limitations in dealing with the dynamic location of microservices since they require a prior registration of the required microservices to perform compositions. Besides, these solutions are not available for both industry and academic communities. To fill that gap, this dissertation proposes Beethoven, a lightweight platform for microservice composition that is composed of a reference architecture and an orchestration DSL based on declarative business processes. The reference architecture follows an event-driven design approach and has been instantiated by using the actor model and the ecosystem provided by Spring Cloud Netflix. In order to demonstrate the feasibility of the Beethoven platform, two example applications have been developed. In addition, to investigate the performance assessment of the Beethoven platform, a controlled quasi-experiment has been conducted. All artifacts produced as part of this dissertation are available on GitHub.

Keywords: Event-driven Architecture. Reference Architecture. Microservice Composition. Orchestration.

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCE CODES

LISTA DE ABREVIATURAS E SIGLAS

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
BDI	Belief-Desire-Intention
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
DDD	Domain-Driven Design
DDM	Data-Driven Multithreading
DSL	Domain-Specific Language
ECA	Event-Condition-Action
ESB	Enterprise Service Bus
HOCL	Higher-Order Chemical Language
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
LTL	Linear Temporal Logic
M2M	Machine-to-Machine
MSA	Microservices Architecture
ProSA-RA	Process based on Software Architecture - Reference Architecture
REST	Representational State Transfer
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration
URI	Uniform Resource Identifier
WS-BPEL	Web Services Business Process Execution Language
WS-CDL	Web Services Choreography Description Language
WSCI	Web Service Choreography Interface
WSDL	Web Services Description Language
XML	eXtensible Markup Language
YAML	YAML Ain’t Markup Language

CONTENTS

1 INTRODUCTION

The traditional monolithic architecture is constructed as a single logic unit that aggregates several services, which share the same computational resources (e.g., memory space, CPU processing, and database), in order to provide business functionalities. Applications based on the monolithic architecture may have a large number of cross-cutting concerns such as logging, security features, and exception handling. Those cross-cutting concerns may be addressed by aspect-oriented programming (KICZALES *et al.*, 1997) in order to increase the software modularity. In particular, when cross-cutting concerns are running through the same application, it is straightforward to apply the concepts of aspect-oriented programming in order to isolate them. In the monolithic architecture, there are also performance advantages in terms of service communication, which is a simple local method invocation, since all services share the same computational resources.

However, three major drawbacks can be mentioned when using the monolithic architecture: (i) difficult to understand and modify over time; (ii) inefficient dimensioning of computational resources; and (iii) difficulty in applying small modifications. In particular, monolithic applications may become difficult to understand and modify over time, due to the increase in number and complexity of their services. Another complicating factor is how to scale efficiently monolithic applications. For instance, if a service in a monolithic application requires more computational resources, it will be necessary to scale the entire application including all services, causing an unneeded allocation of computational resources. Moreover, in monolithic applications, even small modifications impose rebuilding and redeploying the complete application.

In this realm, the Microservices Architecture (MSA) arises as a novel architectural style to develop a single application as a collection of independent, well-defined, and inter-communicating services (NADAREISHVILI *et al.*, 2016). Microservices are autonomous and communicate to each other through lightweight mechanisms, often HTTP resource APIs (LEWIS; FOWLER, 2014). In addition, the microservice architecture proposes a solution for efficiently scaling computational resources and solving other problems, which are mentioned in the previous paragraph, present in the monolithic architecture. Since microservices can be individually scaled, they provide an efficient manner to allocate computational resources, enabling flexible horizontal scaling in cloud environments. For this reason, the microservices architecture style has been used as an alternative to the traditional monolithic architecture in companies such as Amazon, Netflix, and LinkedIn (VILLAMIZAR *et al.*, 2016).

In addition to scalability, the adherence of the microservice architecture style provides the following benefits: free choice of technologies, easy replacement, resiliency, continuous delivery, and facilitated deployment (NEUMAN, 2015; WOLFF, 2016). In particular, microservices offer free technological choice because the communication among them occurs through lightweight protocols. Thus, there is no demand for two different microservices to adopt the same set of technologies in order to establish a communication. As result, different programming languages, database paradigms, and other technology stacks can be used to develop a collection of microservices minimizing the component intercommunication risks for a MSA-based application (WOLFF, 2016). In another perspective, even if the replacement of a specific microservice caused a temporary failure, a MSA-based application can continue to operate and recover the previous microservice version, thus reducing the risk associated with inadequate microservice updates. Furthermore, since microservices can be deployed independently, they are also advantageous for continuous delivery. Thereby, it is easy to ensure a secure deploy in the microservices architecture style, for example, by running different versions in parallel (WOLFF, 2016).

1.1 PROBLEM

Although the MSA offers numerous benefits over the monolithic architecture, there are costs associated with its adoption such as challenges to execute business processes that are distributed across different microservices. In particular, accessing a microservice through its API using a communication protocol (e.g., HTTP) is straightforward. However, as the software complexity increases, there are issues to manage business processes that extend beyond the boundaries of an individual microservice (NEWMAN, 2015). Therefore, the microservice architectural style faces challenges such as microservices cooperation in order to provide complex and elaborated business processes.

In order to illustrate the problems related to microservice composition and explain its importance for MSA-based applications, consider the following scenario: a MSA-based application, responsible for controlling book lending, that is composed of a set of microservices in which each microservice is responsible for running only one business functionality. In this application, there are distinct microservices to perform book catalog management, employee management, and library user management. However, the main functionality of this application, which is the business process for performing book lending, requires a collaboration of different microservices in order to be completed.

To address this issue, there are two approaches that may be used for (micro) services composition: orchestration and choreography. The former refers to a centralized business process that coordinates a series of service invocations, like the conductor in an orchestra, while the latter represents decentralized and cooperative service coordination, like dancers finding their way and reacting to others around them in a ballet (NEWMAN, 2015).

In order to utilize choreography as a service composition strategy, a microservice responsible for performing a task of the business process must process and publish events related to its execution. In addition, it is required to aggregate a broken to the MSA-based application for transmitting the events generated by microservices. This approach is recommended for relatively simple event processing flow (RICHARDS, 2015b). However, for a complex business process that is composed of multiple steps and requires some level of management to process complex data flows, the orchestration strategy is recommended (RICHARDS, 2015b).

Additionally, it is also possible to accomplish service composition using a specific type of service present in SOA-based applications named *composed service*. By definition, a *composed service* is a special type of service that accesses and depends on other services to provide business functionalities (JOSUTTIS, 2007). However, in the context of MSA, the concept of *composed service* violates one of the main principles proposed by the microservices architectural style (LEWIS; FOWLER, 2014): componentization via services. Componentization in MSA is implemented at the level of services that must have only a single responsibility (LEWIS; FOWLER, 2014). For this reason, in MSA-based applications, it is recommended to use service composition strategies (orchestration or choreography) to execute business processes that require the collaboration of different microservices.

In the early 2000s, different approaches were proposed to address web services composition based on Service-Oriented Architecture (SOA) using either orchestration or choreography (MILANOVIC; MALEK, 2004). However, SOA-based approaches for web services composition fail in the context of microservices (YAHIA *et al.*, 2016) because they require components such as an Enterprise Service Bus (ESB) and Web Services Description Language (WSDL) in order to compose services (XIAO; WIJEGUNARATNE; QIANG, 2016). Nonetheless, different from SOA-based approaches, heavyweight middleware and service description are not part of applications that follow the principles of the microservice architectural style (XIAO; WIJEGUNARATNE; QIANG, 2016).

Although there exist some recent approaches for microservice orchestration, such as

Medley (YAHIA *et al.*, 2016) and Microflows (OBERHAUSER, 2016), those solutions have limitations in dealing with the microservices dynamic location since they require a previous registration of the microservices. For instance, in order to orchestrate microservices using Medley or Microflows, it is required to previously describe and register each microservice that will be part of the orchestration. As consequence, new microservices that have not been described and registered previously cannot be used during the microservice composition. In other words, those solutions compromise the scalability of MSA-based applications in case of new microservices need to be composed at runtime. Besides, those solutions are not available for both industry and academic communities, which may compromise their reusability, extensibility, and experimental reproducibility (MUNAFÒ *et al.*, 2017).

Another complicating factor is how to express microservice composition. In order to address this concern, it is necessary mechanisms to specify data flow processes. For instance, workflows are remarkable alternatives to properly represent the possible data flows (NURCAN, 2008). However, due to the complexity and dynamicity of MSA-based applications, workflows are often either too simple, thus unabling to handle the variety of situations that occur, or they are too complex, trying to model every imagined possible situation but being hard to maintain. In both cases, they may cause several problems to the microservices applications. To address these limitations, declarative business processes arise as a paradigm shift from traditional workflow approaches (GOEDERTIER; VANTHIENEN; CARON, 2015).

To fill that gap, this dissertation proposes Beethoven, a lightweight platform for microservice composition that eases the creation of complex applications using microservice data flows. The platform is composed of a reference architecture and an orchestration Domain-Specific Language (DSL) based on declarative business processes that enable software engineers to express microservice orchestration. The reference architecture follows an event-driven design approach and has been instantiated by using the actor model and the ecosystem provided by Spring Cloud Netflix. In order to validate and demonstrate the feasibility of the Beethoven platform, two example applications have been developed. In addition, in order to investigate the performance assessment of the execution of microservice orchestration by using the Beethoven platform, a controlled quasi-experiment has been conducted. The source code of the Beethoven platform , the example applications, and the quasi-experiment are available on GitHub[1,2,3,4].

[1] <https://github.com/davimonteiro/beethoven>
[2] <https://github.com/davimonteiro/partitur>
[3] <https://github.com/davimonteiro/crm-msa-example>
[4] <https://github.com/davimonteiro/performance-evaluation>

1.2 OBJECTIVES

The following subsections describe the general and specific objectives of this dissertation.

1.2.1 General objective

The main objective of this dissertation is to propose an event-driven lightweight platform that is responsible for running declarative business processes in order to orchestrate microservices. Thus, as a result of the execution of such processes, the platform must perform microservices composition in order to offer elaborate functionalities for MSA-based applications.

1.2.2 Specific objectives

In order to achieve the general objective, the following specific objectives must be accomplished:

a) Design and specify systematically an extensible and scalable event-driven lightweight reference architecture for microservice orchestration;
b) Design and implement an orchestration DSL based on declarative business processes in order to provide a mechanism for specifying data or event flows;
c) Recognize the viability of the Beethoven platform by instantiating a concrete architecture based on the specification of the reference architecture;
d) Verify the feasibility of the Beethoven platform and its concrete architecture using an example application that has been developed by the author;
e) Confirm the feasibility of the Beethoven platform and its concrete architecture using an example application that has not been developed by the author;
f) Evaluate a possible overhead that the Beethoven platform and its concrete architecture may produce during microservice orchestration.

1.3 CONTRIBUTIONS

The main contributions of this dissertation are: (i) the microservice composition platform, named Beethoven, and an orchestration DSL, named Partitur, that enables application engineers to express microservices orchestration from an abstract level; (ii) a concrete implementation of the reference architecture using the Spring Cloud Netflix ecosystem, called Spring

Cloud Beethoven; (iii) an example application to demonstrate how the proposed platform can be used in a real example; (iv) a second example application that is an orchestrated version of existing reference application for microservices-based applications; and (v) a controlled experiment that has been conducted in order to evaluate the possible overhead produced by the proposed platform. All artifacts of this dissertation are available to the scientific community on GitHub.

1.4 OUTLINE

The remaining text of this dissertation is organized as follows:

a) Chapter 2 presents the theoretical framework necessary for the understanding of this dissertation. To this end, the chapter presents an overview and a comparison between SOA and the microservice architectural style. Then, it describes the service composition strategies (orchestration and choreography) that can be used in both SOA and MSA. After that, it introduces an overview of declarative business processes;
b) Chapter 3 discusses the main related work in terms of composition languages, web-services composition, and microservice composition. At the end of this Chapter, it is summarized and highlighted a comparison between each related work in terms of contribution type, composition strategy, architectural style, open source, and evaluation type;
c) Chapter 4 presents an event-driven platform named Beethoven for microservices orchestration using declarative business processes. The Beethoven platform is composed of a reference architecture, a concrete architecture and an orchestration DSL;
d) Chapter 5 presents the instantiation of the reference architecture that has been specified in Chapter 4;
e) Chapter 6 presents the evaluations that have been conducted in order to provide empirical evidence for confirming the benefits offered by the Beethoven platform
f) Chapter 7 summarizes the dissertation and restates its contributions. It outlines future work and concludes with a discussion on longer-term perspective and further research directions.

2 BACKGROUND

This chapter presents the theoretical framework necessary for the understanding of this dissertation. To this end, Section 2.1 presents the concepts, architectural principles, and service classification present in SOA. Next, Section 2.2 introduces the state of the art, principles, best practices, and patterns related to the microservices architecture style. After that, Section 2.3 compares the two architectural styles, SOA and MSA, and highlights their main similarities and differences. Then, Section 2.4 describes the service composition strategies (orchestration and choreography) that can be used in both SOA and MSA. Finally, Section 2.5 introduces a global overview of declarative business processes.

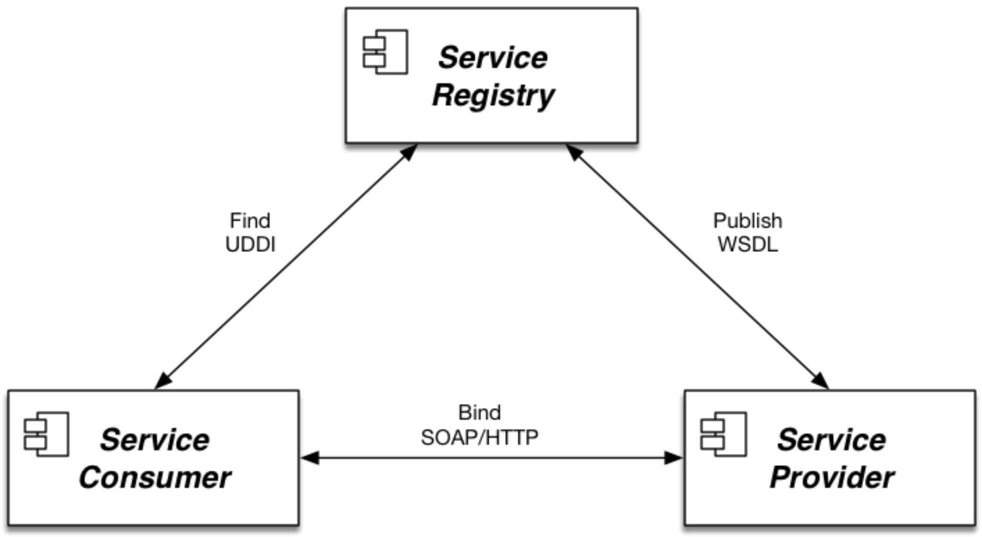
2.1 SERVICE-ORIENTED ARCHITECTURE

SOA is a paradigm and an architectural style for building software applications that are organized as a set of capabilities often distributed across a network and possibly under the control of different ownership domains (LASKEY; LASKEY, 2009; DUGGAN, 2012). By using organized capabilities, it is possible to provide reusable solutions modeled as services to business problems that can be encountered by an individual or organization. Therefore, SOA promotes loose coupling between software components (services) so that they can be reused. In addition, the concept of SOA includes a set of desirable design characteristics for promoting interoperability, reusability, and organizational agility as well as a service-oriented business process modeling paradigm (CHANNABASAVAIAH; HOLLEY; TUGGLE, 2003). The SOA paradigm can be used to realize and maintain business processes developed as large distributed systems that are usually heterogeneous to provide high interoperability (SANDERS; HAMILTON JR.; MACDONALD, 2008).

2.1.1 Architectural components

Software architecture defines the structure of a computing system and comprises software elements, the externally visible properties of those elements, and the relationships among them (BASS; CLEMENTS; KAZMAN, 2012; SHAW; GARLAN, 1996). In SOA, the software architecture of a system is decomposed into well-defined, self-contained, and independent services that can be consumed by clients in different applications or business processes (PAPAZOGLOU; HEUVEL, 2007). A service can be understood as an abstraction of a business capability that provides interfaces to access one or more functionalities. Therefore, in the development of service-oriented applications, all functionalities are provided as services (PAPAZOGLOU; HEUVEL, 2007). The basis for SOA applications is formed by three architectural components: service provider, service consumer, and service registry (HUHNS; SINGH, 2005; PAPAZOGLOU; HEUVEL, 2007). Each SOA component is described below and the interaction between them is represented in Figure 1.

Figure 1 – The basic of Service Oriented Architecture



Source: Papazoglou and Heuvel (2007).

Service provider: Providers are software entities that create services and register them in the service register. Service providers are responsible for publishing a description of the created services using a standardized protocol such as WSDL in public and centralized registries. Thus, by registering services, service clients will be able to find the descriptions of the services in order to use the functionalities provided by the service provider.

Service requester or service consumer: A service consumer represents the software entity that uses the services created by a service provider. In order to locate and use the services provided by a service provider, a service consumer must access the service register to find and retrieve the necessary information for interacting with services. In practice, a service consumer uses Universal Description, Discovery and Integration (UDDI) in order to locate the service provider.

Service broker, service registry or service repository: A service register represents the software entity that both service provider and service consumer interact in order to establish a connection using exchanging data protocol such as Simple Object Access Protocol

(SOAP). The main functionality of a service register is to provide the information about the available services to any potential consumer. If the requested service is available, the service register provides the consumer a contract (service description) and an endpoint address to the consumer service.

2.1.2 Service taxonomy

In the context of SOA-based applications, a service may have different attributes, proposes, and perform different functions. As consequence, there are different manners to classify SOA-based services. Josuttis (2007) classifies them into the following categories:

Single, basic, or atomic services: Atomic services represent a fundamental business operation and should provide a minimal business functionality. In addition, these services should be autonomous and self-contained. For this end, atomic services must not depend on other services in order to provide a business functionality. There are two subcategories of atomic services: data and logic. The former is responsible for reading or writing data from or to databases. The latter is responsible for implementing fundamental business rules, processing input data, and returning correspondent results.

Composite or composed services: Composed services are typically services that access and depended on multiple services to provide business functionalities. For this reason, composite services operate in a level higher than atomic services since they required a collection of services, atomic or composed, in order to execute business processes.

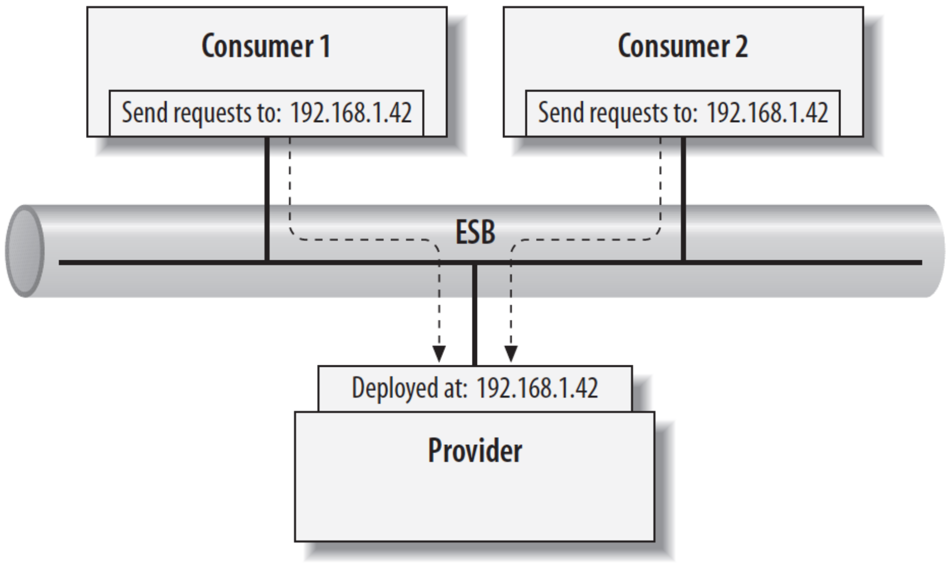
Process services: These services represent business processes that execute in a long period of time and involve some type of human interaction. Thus, process services can be understood as a type of composite service. Although there are similarities in relation to composite services, a process service usually stores state that remains stable over multiple calls. In other words, process services are stateful.

2.1.3 SOA instantiation

In order to instantiate the concepts present in SOA, there are two central elements that may be used: web services and ESB. A web service is defined as a standardized way of integrating web-based applications using interoperable standards Machine-to-Machine (M2M) such as eXtensible Markup Language (XML), SOAP, WSDL, and UDDI over an Internet protocol backbone (ALONSO *et al.*, 2004). These standards provide a common approach for defining, publishing, and using web services. For instance: (i) XML is used to represent data; (ii) SOAP is used to transfer data; (iii) WSDL is used for describing the services available; (iv) UDDI is used for listing available services.

ESB is a software infrastructure that implements communication patterns over different transport protocols in order to support interaction between distributed services in a SOA-based application (PAPAZOGLOU, 2003; JOSUTTIS, 2007). Its purpose is to provide interoperability, connectivity, data transformation, and intelligent routing combined with some additional services such as security, monitoring and logging, service management and so on (JOSUTTIS, 2007). Figure 2 illustrates an example of ESB that acts as a mediator layer between service providers and service consumers. The main rule of an ESB is to provide interoperability between different platforms and programming languages. In this case, for web services, SOAP is used as a standard format to which all platforms, services, and programming languages must use.

Figure 2 – An ESB providing point-to-point connections


Consumer 1
Send requests to: 192.168.1.42
Consumer 2
Send requests to: 192.168.1.42
ESB
Deployed at: 192.168.1.42
Provider

Source**:** Josuttis (2007).

2.2 MICROSERVICES

MSA is an architectural style for developing a single application as a collection of small and autonomous services, each running in its own process and communicating through lightweight mechanisms, often an Hypertext Transfer Protocol (HTTP) resource Application Programming Interface (API) (LEWIS; FOWLER, 2014). Microservices are built around business capabilities using a concept from Domain-Driven Design (DDD) (EVANS, 2003) named bounded context in order to delimit their business functionalities and associated data. In addition, a microservice can be understood as a small and a single responsibility application that can be independently deployed, scaled, and tested (THöNES, 2015). By adopting the microservices architecture, developers can engineer applications that are composed of multiple, self-contained, and portable components deployed across numerous distributed servers (FAZIO *et al.*, 2016).

In order to motivate the use of MSA, it is important to compare it to the monolithic style. A monolithic application is a single software unit that aggregates all necessary computational resources to execute all business functionalities in a particular domain. For instance, in a monolithic-based application, if one service requires more computer resources in order to be executed, it will be necessary to scale the entire application. In addition, in monolithic applications, any modifications impose rebuilding and redeploying a new version of the application.

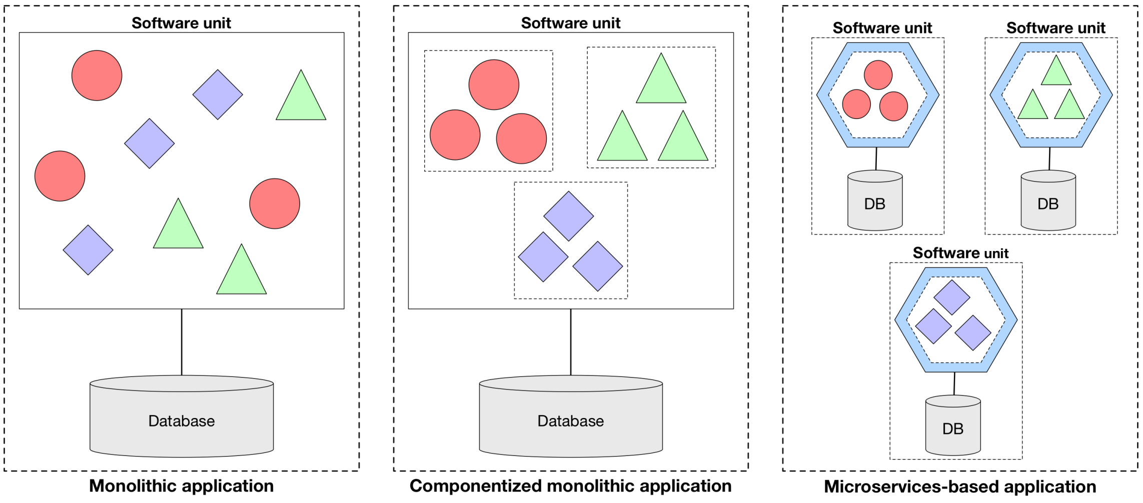
In software engineering, system modularization can be defined as a process of grouping parts of a program and, when appropriate, removing redundancies (SOMMERVILLE, 2010). Although it is possible to build monolithic applications without using software modularization, as shown on the left side in Figure 3, monolithic applications can and should apply the principles and best practices of software engineering such as modularization (HAYWOOD, 2017a; HAYWOOD, 2017b) in order to achieve attributes such as maintainability, low coupling, and high cohesion as shown by the middle image in Figure 3. However, MSA goes one step further by isolating each microservice into an independent application that can be run into virtual containers Docker[1] (BERNSTEIN, 2014) as shown on the right side in Figure 3.

As a consequence of the limitations imposed by the monolithic architecture and the benefits brought by the microservices architecture, companies have adopted MSA as an alternative to the monolithic architecture (MAZLAMI; CITO; LEITNER, 2017). In the scientific community, researchers have focused their efforts on strategies, methodologies, and models for migration from monolithic applications to MSA-based applications (MAZLAMI; CITO; LEITNER, 2017; ESCOBAR *et al.*, 2016; KECSKEMETI; MAROSI; KERTESZ, 2016).

According to Richards (2015a), MSA has a limited service taxonomy that categorizes microservices into two groups: infrastructure and business. The former group encloses more generic and reusable microservices that support nonfunctional tasks such as Discovery Services, API Gateway, Load Balancer, and Configuration Services. The latter group, also known as functional microservices, contains microservices that support specific business operations or

[1] <https://www.docker.com/>

Figure 3 – From monolithic applications to microservices



Source**:** Elaborated by the author.

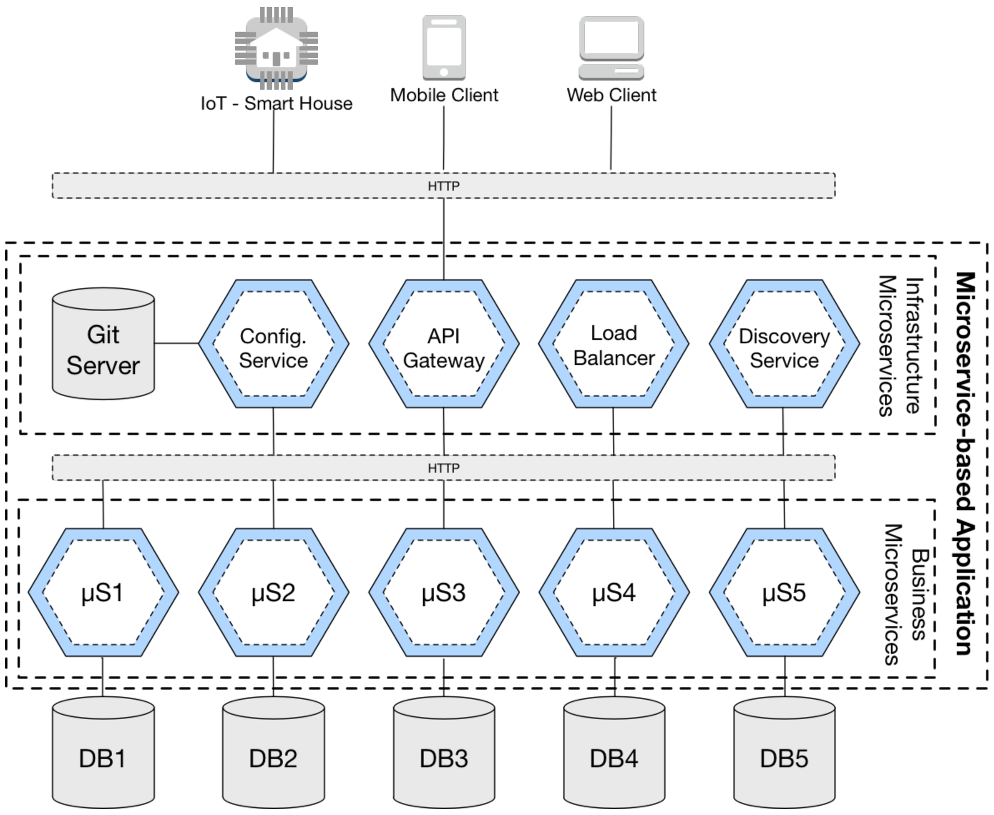
functions. Figure 4 illustrates an example of MSA-based application that follows some of the patterns and best practices that are proposed by both academic literature (MONTESI; WEBER, 2016) and industry. On the top of Figure 4, it is illustrated different clients that access the MSA-based application. In the middle of Figure 4, it is shown the infrastructure microservices responsible for supporting the business microservices. Finally, on the bottom of Figure 4, it is shown the business microservices in which each one accesses its own database.

2.2.1 Principles of the microservice architecture style

For some researchers, MSA is a subset of SOA or a special approach to constrain any SOA-based application to be successful (PAUTASSO *et al.*, 2017a; PAUTASSO *et al.*, 2017b). Although there is no formal definition of the microservices architectural style, Lewis and Fowler (2014) describe a set of principles for this architecture style. Since a software architectural style defines a set of principles that should be used as constraints on a software architecture (PERRY; WOLF, 1992; GARLAN; SHAW, 1994), the following principles should be implemented in MSA-based applications.

Componentization via services - Componentization in software development is a decomposition methodology used to build pieces of software that are easy to execute, test, replace, and deploy (KAUR; MANN, 2010). In microservices application, componentization is performed at the level of services that must have only a single responsibility. Each microservice component encapsulates all the resources required to execute and expose interfaces to other microservices, ensuring loose coupling among the microservices.

Figure 4 – The microservice architecture



Source: Elaborated by the author.

Organised around business capability - Any organization that designs software systems will produce application following its organizational structure (CONWAY, 1968). In other words, application code and teams that are organized around functional areas focus on specialized teams such as user experience teams, server-side logic teams, and database teams. In the context of microservices applications, that organizational structure can impact negatively on the development of new features or requirements changes. For this reason, Lewis and Fowler (2014) argue that the development of a microservice must be performed by cross-functional teams that include user-interface, persistent storage, and any external collaborations.

Products not projects - A project can be defined as a temporary endeavor undertaken to create a unique product or service (INSTITUTE, 2013). However, in the microservices realm, Lewis and Fowler (2014) advocate that the project model as a temporary effort should be avoided. Instead, microservice engineers should adopt the philosophy that the microservice team should own a product over its full lifetime. More specifically, the development team must take full responsibility for the software in production.

Smart endpoints and dumb pipes - A microservice must have all the logic and resources required to function independently. Therefore, Lewis and Fowler (2014) advocate that a microservice is a smart endpoint. The communication mechanism among the microservices (smart endpoint) must follow the philosophy of dumb pipes over a lightweight protocol. Dumb pipes mean that communication mechanisms should act as a message router only. Thus, for instance, the logic for the operation of a microservices should not be transferred to its communication mechanism.

Decentralised data management and governance - The flexibility to adopt different technologies for each microservice is mentioned by Wolff (2016) as one of the main advantages related to the microservice architecture. By using decentralization governance, microservice engineers can address issues using the most appropriate technologies since there is no contract to a single technology as in monolithic applications. Unlikely a monolithic application, in which there is a centralized database, data in microservices applications is decentralized and distributed between each microservice. Decentralized data management provides a framework to define boundaries between microservices since each microservice is a solution to a business capability and works with a conceptual data model (SHADIJA; REZAI; HILL, 2017).

2.2.2 Microservice patterns and best practices

MSA arises as an alternative to the monolithic architectural style by providing benefits such as independent deployments and development, small and focused teams, fault isolation, decentralized governance, and decentralized data management (FRANCESCO, 2017; FRANCESCO; MALAVOLTA; LAGO, 2017). However, in order to exploit these benefits, a MSA-based application should implement the following patterns and best practices as an infrastructure microservices layer as presented in Figure 4.

Circuit Breaker Pattern — In MSA, isolated failures of a single microservice may cascade beyond its boundaries and, thereby, bring the entire microservices application down. For instance, on the occasion that a microservice is unavailable, then any microservice that depends on the unavailable microservice may face problems (e.g., long timeouts). If this problem is not addressed and propagates to other microservices, the health of the MSA-based application will be compromised. In order to prevent that problem, known as the cascading problem, the microservice consumer should use the Circuit Breaker Pattern (NYGARD, 2007) when calling the microservice provider. In this pattern, each microservice must use an internal circuit breaker. Thus, if the microservice provider is available, then the circuit will remain closed (closed circuit state). However, when a microservice provider becomes unresponsive after a number of failure responses, the circuit in the microservice consumer assumes that the provider may be unavailable and should stop waiting for it (open circuit state) (MONTESI; WEBER, 2016). After a specific timeout, the circuit will verify if the microservice provider is available and, in case of a successful attempt, the state will be changed to the closed circuit state. Otherwise, the state will be modified to open circuit. Thus, the Circuit Breaker pattern is an effective mechanism to combat long timeouts and cascading failures.

API Gateways Pattern — Microservices application can serve different clients such as web client, mobile client, and Internet of Things (IoT) devices. Each client may have different needs and restrictions. For instance, in a web store application, when requesting details about a product, the mobile client needs less information than a web client due to its connection and processing limitations. To address this problem, the API Gateway pattern provides a single entry point to access the microservices APIs. In addition, an API Gateway provides functionalities for publishing multiple APIs, each one dedicated to a different set of clients (MONTESI; WEBER, 2016).

Service Discovery Pattern — As a consequence of the dynamic aspect of a microservice that can be deployed, replicated or reallocated during the application execution, it is not possible to determine the microservice location (endpoint) at design time. To address the service location problem, the Service Discovery Pattern is used to determine the location of a service instance at runtime by using a service registry, which can be used by other components to retrieve binding information about other components (MONTESI; WEBER, 2016). In the Service Discovery pattern, microservices register themselves in a service registry in order to publish their locations. There are some services that implement the Service Discovery pattern such as Consul[2], Apache ZooKeeper[3], and Netflix Eureka[4].

Externalized Configuration — The MSA typically integrates a variety of loosely coupled infrastructure microservices that must run in multiple environments (development, test, production) without modifying or rebuilding the microservices during the exchange of environments. Due to technology heterogeneity, configuring each microservice to provide the basis for running and multiple environments might be time-consuming. A possible solution to address microservices configuration is to externalize all configuration from a microservices application, including database properties (URL, credentials, tuning parameters), specific microservices

[2] <https://www.consul.io>
[3] <http://zookeeper.apache.org/>
[4] <https://github.com/Netflix/eureka>

configuration, network configuration, and among others [5]. Therefore, during the microservices initialization, all configurations are read from an external source (e.g., OS environment variables or a Git Server). For instance, Spring Boot externalized configuration [6] allows microservices engineers to externalize configuration that can be read from a variety of external sources including properties files, YAML Ain't Markup Language (YAML) [7] files, environment variables and command-line arguments.

Client Side Load Balancer Pattern — A load balancer that is achieved at the client side for routing requests across servers, rather than at the server side of a client-server network. In this model, each client service regularly receives a load balance list of available services and routes requests using a particular load balancing algorithm (e.g., Round Robin or Random). In the event of a service failure or non-responsive service, the unavailable service can be removed from the load balance list and redistribute the load. The client service's list then is updated when the list is received during subsequent access. Netflix Ribbon[8] is an implementation of the Client Side Load Balancer Pattern.

2.3 SOA VS. MICROSERVICES

SOA and MSA are similar concepts since they both are architectural styles for building applications that are decomposed into services available over a network and integrated across heterogeneous platforms. Furthermore, in both approaches, the system software architectures share the same purpose: decomposing large systems into well-defined services (componentization via service). Thereby, the difference between SOA and MSA is a controversial subject among the industrial and scientific communities since both styles use services as architectural components in order to implement business functionalities (RICHARDS, 2015a). However, in order to achieve the common purpose, each architectural style follows a different path.

In terms of communication mechanisms and governance model in SOA, the application architecture is designed to decompose a large application into simple services, emphasizing service integration with intelligent routing mechanisms (CERNY; DONAHOO; PECHANEC, 2017; CERNY; DONAHOO; TRNKA, 2018). The intelligent routing mechanism provides a centralized governance model that is capable of routing messages, especially dealing with technical aspects, such as load balancing and failover. In addition, intelligent routing, from a

[5] <http://microservices.io/patterns/externalized-configuration.html>
[6] <https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html>
[7] <http://yaml.org/>
[8] <https://github.com/Netflix/ribbon>

business point of view, might also provide different ways of routing messages. For instance, in order to implement business rules, intelligent routing might process messages that are assigned to different priorities or must be processed according to their content (so-called content-based routing). Thereby, intelligent routing mechanisms presented in SOA might result in simplistic endpoints and complex communication channels.

In contrast, since a microservice is a separate codebase that can be deployed independently and is responsible for persisting its own data, MSA implies that the decomposition strategy used is to separate large applications into smart and independent services while considering simple routing mechanisms (BALALAIE; HEYDARNOORI; JAMSHIDI, 2016), with a decentralized governance model. According to Salah *et al.* (2016), the main difference between SOA and MSA is the elimination of the service bus (a central governance entity). As a result, by maintaining dumb pipes to communicate and moving the intelligence and control to the endpoints, MSA leads microservices into higher autonomy and decoupling because they do not need to agree on global level contracts (CERNY; DONAHOO; TRNKA, 2018).

In the context of component sharing, SOA and MSA are inherently different since each architectural style emphasis a different concept. According to Richards (2015a), SOA is based on the principle of reusability that is achieved by the idea of *share-as-much-as-possible* architecture approach. The idea of SOA is to mitigate redundancies in business functionality through the creation of shared software components (services and database) for increasing efficiency and cost saving. MSA, on the other hand, is based on idea of *share-as-little-as-possible*. In particular, MSA are built on the concept from DDD named bounded context. In terms of microservices, a bounded context delimits the functionalities and associated data of a particular service in order to model single closed entity with minimal dependencies. The different goals and philosophies in component sharing leads to different focuses when designing and implement services.

The implementation of systems based on SOA or MSA can support different communication protocols and data-interchange formats. In practice, the same communication protocols or data-interchange formats used by SOA-based applications can be applied in MSA-based applications and vice versa. However, according to Josuttis (2007), SOA has used the following web services standards to realize the web services approach: (i) XML for describing data formats and data types; (ii) HTTP as communication protocol; (iii) WSDL for defining service interfaces; (iv) SOAP for defining the web service protocol; (v) UDDI for registering and finding services. In contrast, according to Xiao, Wijegunaratne, and Qiang (2016), MSA has used: (i) Representational State Transfer (REST) or Advanced Message Queuing Protocol (AMQP) as a SOAP alternative; (ii) HTTP as communication protocol; (iii) JavaScript Object Notation (JSON) for representing data.

The implementation of SOA concepts centralizes various infrastructure services (e.g., communication, security, composition, management) into a single architectural component named ESB. Although ESB is one of the most important components of SOA, it is a single point of failure that can bring down all the infrastructure service in a SOA-based application. Moreover, several ESB implementations (e.g., IBM WebSphere ESB[9], Oracle Service Bus[10], and SAP Process Integration[11]) use proprietary protocols to provide communication between web services. Consequently, such ESB providers create a vendor lock-in in their solutions preventing the reuse of web services on other providers.

In contrast, MSA decomposes centralized services presented in an ESB into several independent microservices where each microservice is bounded by its responsibility. For instance, as shown in Section 2.2.2, there are infrastructure microservices responsible for API Gateway, Service Discovery, and Configuration Server. Solutions for problems encountered in SOA applications may not apply in REST-based microservices. Therefore, instead of using an existing ESB, those infrastructure microservices have been implemented in order to provide lightweight and REST-based solutions that are designed for microservices. Table 2 aims to summarize the comparison between SOA and MSA described in this section.

2.4 SERVICE COMPOSITION STRATEGIES

SOA and MSA represent two architectural styles for building service-based applications that are composed of multiples distributed services. Considering the principles presented in both architectural styles, services must be well-defined, self-contained, and, specifically in the terms of microservices, small. In this form, in order to perform complex business processes, service collaboration is required. For instance, considering a service-based application (SOA or MSA) for hospital management, which each service is responsible for a business capability, it is necessary to perform service interactions to execute business processes such as patient transfer, hospitalization, operation recovery, among others.

Service collaboration can be performed by a specific service known as composed

[9] <https://www-01.ibm.com/software/br/info/middleware/applications/>
[10] <http://www.oracle.com/technetwork/middleware/service-bus/overview/index.html>
[11] <https://wiki.scn.sap.com/wiki/pages/viewpage.action?pageId=16263>

Table 2 – Comparing SOA and MSA

Concern	**SOA**	**MSA**
Architectural component	Services as architectural components	Services as architectural components
Component sharing	Built on the idea of "share-as-much-as-possible" architecture approach	Built on the idea of "share-as-little-as-possible" architecture approach
Philosophy	More importance on business functionality reuse	More importance on the concept of bounded context
Governance model	Centralised data management and governance	Decentralised data management and governance
Communication mechanisms	ESB for communication (too smart pipes)	Less elaborate and simple messaging systems (dump pipes)
Supported protocols	Uses protocols such as WSDL, SOAP, and UDDI	Uses lightweight protocols such as HTTP/REST and AMQP

Source**:** Elaborated by the author.

service. As mentioned in Section 2.1, composed services are services that access and depended on multiple services in order to provide business functionalities. Although the concept of composed service can be applied in service-based applications, composed services infringe one of the main principles proposed by the microservices architectural style: componentization via services. In particular, componentization in MSA is implemented at the level of services that must have only a single responsibility and be self-contained (LEWIS; FOWLER, 2014). For this reason, in MSA-based applications, instead of using composed services, it is recommended to utilize service composition strategies to execute business processes that require microservice collaboration.

In order to address the problem of services collaboration, service composition is adopted by many companies as a flexible solution for building service-based applications. Service composition includes a sequence of tasks or activities and relationships between them to execute a business process. In this context, service orchestration and choreography represent two perspectives on how a service composition can be executed (PELTZ, 2003). The former refers to a centralized business process that coordinates a series of service invocations, like the conductor in an orchestra, while the latter represents decentralized and cooperative service coordination, like dancers all finding their way and reacting to others around them in a ballet (NEWMAN, 2015).

Each service composition approach (orchestration or choreography) has advantages and disadvantages since they address different problems. For instance, according to Richards (2015b), service orchestration is based on the mediator topology that is commonly used for coordinating multiple steps through a central mediator. In addition, service orchestration is recommended when it some level of management at a higher level to process complex data flows is required. Service choreography, on the other hand, is based on the broker topology that is used to chain events together without the use of a central mediator. It is recommended for a relatively simple event processing flow.

2.4.1 Service orchestration

Service orchestration refers to a service composition strategy in which services are controlled in a centralized manner (NANDA; CHANDRA; SARKAR, 2004). A centralized entity, known as orchestrator or conductor, is responsible for managing and coordinating the entire service orchestration. In order to specify service orchestration, domain-specific languages for orchestration are used to describe service interactions by identifying messages, modeling business logic, and invocation sequences (BARKER; WALTON; ROBERTSON, 2009). Those languages are executed on an orchestration engine. In the SOA domain, Web Services Business Process Execution Language (WS-BPEL) is a standard executable language for specifying service orchestration within business processes. An orchestrated business process is also called executable processes since they are intended to be executed by an orchestration engine (BARROS; DUMAS; OAKS, 2005).

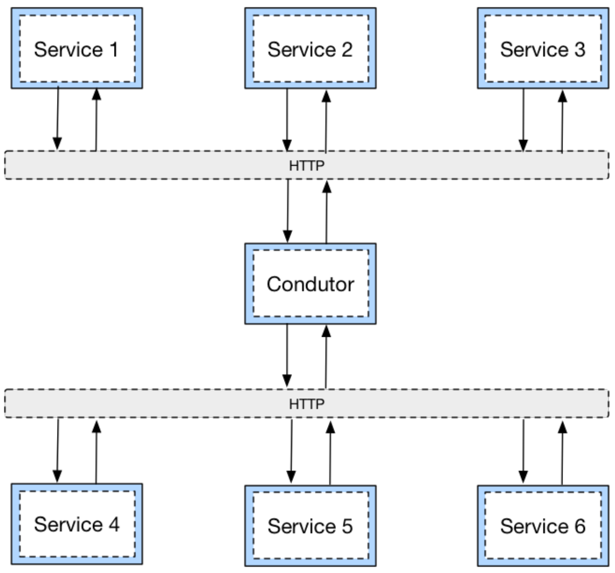
Figure 5 represents the topology found in service orchestration that is composed of a set of services and a single centralized entity. The centralized entity represented in Figure 5 is responsible for coordinating each service in a given order to perform a process flow. To this end, the coordinator performs requests and receives responses for completing a business process.

2.4.2 Service choreography

Service choreography describes a collaboration among a collection of services to achieve a common goal focusing on message exchange (BARROS; DUMAS; OAKS, 2005; BARKER; WALTON; ROBERTSON, 2009). Service choreography may be expressed in languages such as Web Service Choreography Interface (WSCI)[12], Web Services Choreography Description Language (WS-CDL)[13], and Business Process Model and Notation (BPMN) in

[12] <https://www.w3.org/TR/2002/NOTE-wsci-20020808/>
[13] <https://www.w3.org/TR/ws-cdl-10/>

Figure 5 – Service orchestration



Source: Elaborated by the author.

version 2.0 [14]. Different from service orchestration, in the choreography model there is no central element responsible for coordinating services. In this model, each service is responsible for listening, processing, and publishing events. Although there is no central element responsible for coordinating services, a lightweight message broker (e.g., ActiveMQ, HornetQ, or ZeroMQ) is required to distribute events that are triggered by the collection of services. This event distribution activity is performed by a broker or event channel using the publish-subscribe pattern (EUGSTER *et al.*, 2003).

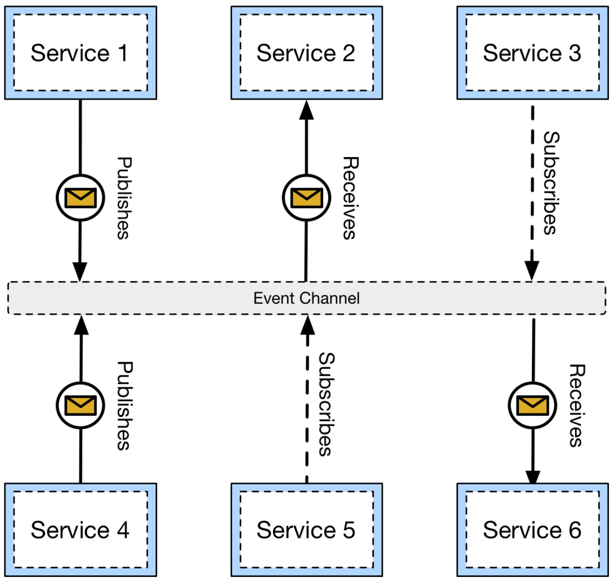
The choreography model is exhibited in Figure 6 and is composed of a set of services and one message broker. The control in choreography is decentralized since each service involved in this composition must know which events should be listened and processed and which events should be created and published in the message broker.

2.5 DECLARATIVE BUSINESS PROCESS

A business process consists of a set of activities that are performed in coordination to accomplish a business goal (AALST; ROSA; SANTORO, 2016). Each business activity may require inputs to process its tasks and produce outputs after finishing its processing. In particular, a business process can be represented as a model in order to provide an abstraction that acts as a blueprint for a set of business process instances. For instance, a workflow can be defined as a business process model that is represented graphically in order to facilitate the visualization of

[14] <http://www.omg.org/spec/BPMN/2.0/>

Figure 6 – Service choreography



Source: Elaborated by the author.

a specific business process. In addition, a workflow presents all the necessary information to automate its execution (SILVA *et al.*, 2013). In this context, there are two modeling paradigms to specify business processes: declarative and imperative.

An imperative business process is a modeling approach that provides a precise definition of the control-flow in the business process (GOEDERTIER; VANTHIENEN; CARON, 2015). In this approach, all the execution alternative flows must be explicitly represented in the process model. Well-designed imperative process models based on formal semantics and precise specifications can be executed efficiently and effectively (GOEDERTIER; VANTHIENEN; CARON, 2015). A dynamic business process, however, may result in maintainability issues (FAHLAND *et al.*, 2009) because there may be new flows that were not mapped during the modeling process. Therefore, the imperative approach is a viable option for cases in which the business process is well known and stable.

In a different manner, a declarative business process, also known as dynamic business process (GASEVIC; GROSSMANN; HALLE, 2009), is a modeling approach that focuses on what should be done in order to achieve business goals, without specifying all possible alternative flows and how to reach the final state in the process model (GOEDERTIER; VANTHIENEN; CARON, 2015). That approach can be defined by Event-Condition-Action (ECA) rules. According to Alferes, Banti, and Brogi (2006), ECA rules are an intuitive and powerful paradigm for programming reactive systems and its fundamental construct forms the following structure: on *Event* if *Condition* do *Action*.

In this context, a declarative business process specifies sets of event conditions, constraints or business rules, and business activities. Event condition is a type of event that should be listened. A constraint is a business rule that must be respected during the process execution. A business activity is an action that manages a business resource. As a consequence of the declarative approach, all alternative flows are implicitly specified and defined as the flow that does not violate the business rules. In addition, declarative business processes can be combined with formalisms such as Linear Temporal Logic (LTL) to ensure the correctness of the process models (SILVA *et al.*, 2013).

In order to illustrate declarative process modeling, a simplified credit approval process is described by Goedertier, Vanthienen, and Caron (2015) . The declarative process represents a set of business activities and a set of business rules. This process can be declaratively specified:

- **business activities**: handleCreditApplication, applyForCredit, checkDebt, checkIncome, reviewCredit, rejectCredit, makeProposal, rejectProposal, acceptProposal, reviewCollateral, changeApllication, collectInformation, completeContract, and closeApplication.
- **business rule 1**: when the customer applies for credit the bank should either make a credit proposal or reject the credit application within 10 days.
- **business rule 2**: a credit review and a collateral review are required before either a proposal is made or the credit is rejected.
- **business rule 3**: income and debt information are a prerequisite to reviewing a credit.
- **business rule 4**: the worker who reviews the credit cannot be the applicant of the credit application.
- **business rule 5**: when the bank makes a credit proposal, the bank is committed to complete the contract when the customer accepts the proposal within 5 days.
- **business rule 6**: the activities check income and check debt may only be performed after a customer has applied for credit.

3 RELATED WORK

The proposal of this dissertation involves the challenge of performing microservices orchestration using a DSL based on declarative business processes. There are research papers that propose DSLs to specify web-services or microservices composition. Other papers present proposals to perform web-services composition applying approaches inspired by chemical reactions. Other studies address microservices composition using orchestration as main service composition strategy. In this context, the related work presented in this chapter address at least one of the mentioned dimensions, which may have different approaches or use other technologies.

3.1 COMPOSITION LANGUAGES

Jaradat, Dearle, and Barker (2013) have proposed a DSL for web services orchestration that provides abstractions for defining service identifiers, workflow interfaces, and coordination elements. In order to address scalability challenges presented in centralized orchestration (e.g., the consumption of network bandwidth, degradation of performance, and single points of failure), the proposed language is used to specify decentralized orchestration using service-oriented workflows that can be partitioned into smaller fragments to be executed in different distributed orchestration services. These orchestration services collaborate in order to execute each part of the workflow specification until it has been completed.

The workflow specification is based on WSDL in order to identify and locate web services. As result, the proposed DSL cannot be applied to orchestrate microservices that are based on the RESTful architectural style. Finally, Jaradat, Dearle, and Barker (2013) have conducted performance evaluations in order to verify the execution time between decentralised and centralised orchestration using the proposed DSL.

Safina *et al.* (2016) have extended the Jolie programming language[1] in order to support microservice orchestration using data-driven workflows. Formerly, Jolie is a programming language proposed by Montesi, Guidi, and Zavattaro (2014) that can be used to orchestrate web-services based on WSDL. Jolie is an imperative DSL that is used to define two types of instructions: deployment and behavior. The former is composed of web service interfaces, message types, and communication ports. The latter represents a sequence of instructions that define imperatively the service orchestration.

In the extended version of Jolie, the authors have introduced new data types and

[1] <http://www.jolie-lang.org/>

operations in order to support regular expressions and data-driven operators. For evaluation, the authors have presented two example applications that are used to illustrate how to apply service orchestration using the original and the extended version of Jolie.

Despite proposing a solution for microservice orchestration, however, Safina *et al.* (2016) have not adopted the principles of the microservice architecture style discussed in Section 2.2.1. In addition, instead of applying the microservice patterns and best practices discussed in Section 2.2 to support microservice composition, the extended version of the Jolie language inherits SOA standards, such as WSDL and ESB, to identify and orchestrate microservices. However, such SOA standards, as discussed in Section 2.3, have not been applied in microservice-based applications. Finally, the Jolie language uses an imperative approach to define data flows in which each step must be specified such as microservices location, communication protocol, inputs and outputs, and statements conditions (e.g., if and else statements).

3.2 WEB-SERVICES COMPOSITION

Wang and Pazat (2013) have proposed a chemistry-inspired middleware for web service composition using orchestration or choreography by providing different sets of rules. Depending on the rules, application engineers are able to specify their preferred execution models to run service compositions, in either centralized or collaborative way. The middleware is implemented as a distributed chemical system in which a collection of web services are modeled as a series of chemical reactions. Furthermore, the middleware is implemented using Higher-Order Chemical Language (HOCL) (BANâTRE; FRADET; RADENAC, 2006), a programming language that implements the chemical programming model, and running over distributed infrastructures (Grid'5000[2]).

The chemical programming model describes computation in terms of a chemical solution in which molecules (representing data) interact freely according to reaction rules (BANâTRE; FRADET; RADENAC,). In the proposed middleware, to perform the composition of web services, each service must be modeled as a chemical web service and connected to an existing web service. In this manner, the collaboration between chemical web services using orchestration or choreography is propagated to the real-world web services. Lastly, the authors have conducted a number of experiments in order to evaluate the efficiency and complexity of different models using the chemical middleware.

[2] <www.grid5000.fr>

Another similar approach inspired by chemistry has been proposed by Fernández, Tedeschi, and Priol (2016). The authors have proposed a chemistry-inspired middleware for executing web service composition in a decentralized manner. In order to define service composition, the HOCL language is used to express workflow definition according to the chemical paradigm. Consequently, it is required to establish a connection between the chemical abstraction of a web service and an existing web service. Lastly, a proof of concept has been realized by the authors, through the deployment of a software prototype, in order to demonstrate the viability of the proposed solution for service composition.

Both aforementioned approaches, however, bring complexity and overhead during the service composition since it is necessary to establish connections between the chemical web service and exiting web services. In addition, the adoption of those approaches may face difficulties and resistance in lightweight architecture such as microservices since it is required a chemical middleware in order to execute service composition defined in HOCL. In addition, solutions based on SOA cannot be applied for REST-based microservice composition because those solutions require description languages (e.g., Business Process Execution Language (BPEL) and WSDL) and a middleware (e.g., ESB) to composite web services.

3.3 MICROSERVICES COMPOSITION

Yahia *et al.* (2016) have proposed an event-driven lightweight platform for microservice composition, named Medley, using orchestration as main composition strategy. In the Medley platform, a particular DSL is used to describe microservice orchestration as compositions. Before defining a composition in Medley DSL, it is required to register the information (e.g., endpoints, operations, data types) about each microservice that will be used during the orchestration. The information about the registered microservice is stored in the platform's process repository for later use. For instance, after registering the information about microservices, a user can use the registered microservices to define which operations will be performed during the orchestration. In addition, Medley DSL follows an event-driven approach for expressing microservice orchestration according to the events that may occur during the microservice composition. Lastly, in order to validate the proposed platform, the authors have conducted a performance evaluation to analyze platform scalability criteria.

Oberhauser (2016) has proposed a lightweight approach, named Microflows, for microservices orchestration using a declarative approach with cognitive agents. In order to specify microservice orchestration using Microflows, it is required to perform three activities: (i) describing information (e.g., endpoints, supported operations, inputs and outputs) about a microservice using a JSON-based service description; (ii) specifying goals for a particular microservice composition; (iii) defining constraints that must be obeyed during the microservice composition. After defining the microservice orchestration, Microflows utilizes a graph-based database in order to store each service description as a node and microservice dependencies and constraints as edges. In this manner, based on the stored microservice orchestrations (service descriptions, goals, and constraints), the Microflows approach uses Belief-Desire-Intention (BDI) agents (RAO; GEORGEFF *et al.*, 1995) to execute workflows. The author has conducted a performance evaluation for validating the Microflows approach.

Although the aforementioned research papers propose approaches for composite microservices, Medley and Microflows are similar to SOA-based approaches for describing services in order to composite them. For instance, in order to orchestrate microservices using Medley or Microflows, it is required to previously describe and register each microservice that will be part of the orchestration. As consequence, new microservices that have not been described and registered previously cannot be used during the microservice composition. Therefore, Medley and Microflows have limitations in dealing with the microservices dynamic location. Moreover, these solutions are not available for both industry and academic communities, difficulting reusability, extensibility, and experimental reproducibility.

3.4 COMPARISON BETWEEN APPROACHES

Table 3 summarizes and highlights a comparison between each related work and the Beethoven platform in terms of contribution type, composition strategy, architectural style, open source, and evaluation type. The first column represents the study identifier that is cited in the table notes. The second column exhibits the main contribution type that has been proposed (e.g., DSL, middleware, or platform). The third column describes which composition strategy has been addressed by the related work. The fourth column expresses which architectural style is addressed during the service composition (e.g., orchestration or choreography). The sixth column indicates if the proposed solution is open source or not. The last column depicts the evaluation type that has been conducted to evaluate the proposed solution by the related work.

Table 3 – Comparison between approaches in the related work

Ref.	Contribution	Composition strategy	Architectural style	Solution approach	Open source	Evaluation type
S1[1]	DSL for web-service orchestration	Decentralized orchestration	SOA	Imperative approach	No	Performance evaluation
S2[2]	DSL for microservice orchestration	Orchestration	SOA and Microservice	Imperative approach	Yes	Example application
S3[3]	Middleware for web service composition	Orchestration or choreography	SOA	Declarative approach	No	Experiment
S4[4]	Middleware for web service composition	Decentralized orchestration	SOA	Rule-based programming	No	Example application
S5[5]	Platform for microservice composition	Orchestration	Microservice	Event-driven approach	No	Performance evaluation
S6[6]	Platform for microservice composition	Orchestration	Microservice	Declarative approach with BDI agents	No	Performance evaluation
S7[7]	DSL and platform for microservice orchestration	Orchestration	Microservice	Event-driven architecture and declarative business processes	Yes	Example applications and controlled experiment

[1] Jaradat, Dearle, and Barker (2013)
[2] Safina *et al.* (2016)
[3] Wang and Pazat (2013)
[4] Fernández, Tedeschi, and Priol (2016)
[5] Yahia *et al.* (2016)
[6] Oberhauser (2016)
[6] The Beethoven platform

Source**:** Elaborated by the author.

4 BEETHOVEN

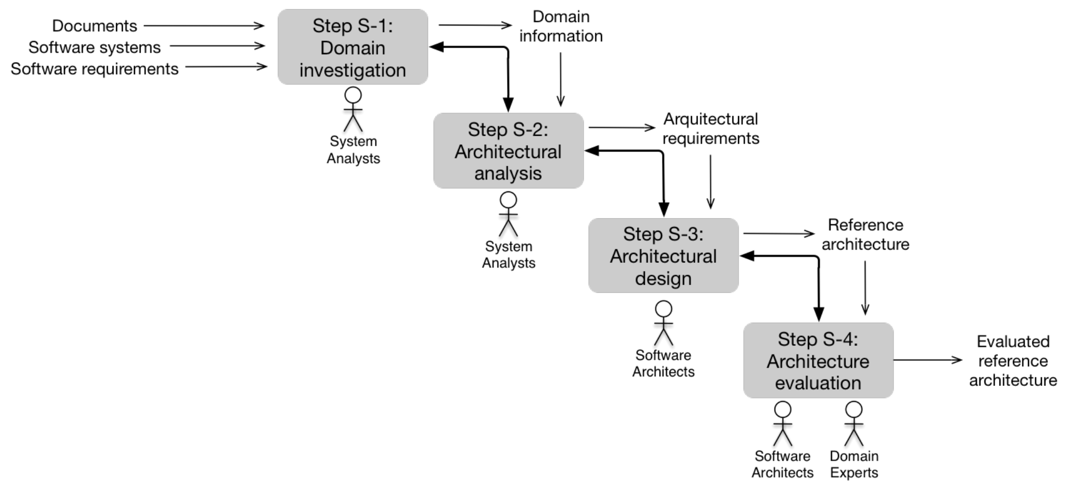
This chapter presents an event-driven platform, named Beethoven, for microservices orchestration that facilitates the creation of complex MSA-based applications using microservice data flows. The Beethoven platform is composed of: (i) a reference architecture, described systematically using a particular methodology in Section 4.1; and (ii) an orchestration DSL based on declarative business processes, detailed in Section 4.2.

4.1 BEETHOVEN'S REFERENCE ARCHITECTURE

A software reference architecture is a special type of architecture that provides abstractions and guidelines for the specification of concrete architectures in a certain domain (ANGELOV; GREFEN; GREEFHORST, ; ANGELOV; GREFEN; GREEFHORST, 2012). According to Martínez-Fernández *et al.* (2017), reference architectures provide the following benefits: (i) standardization for concrete software architectures by using software reference architectures as blueprints for designing applications fulfilling such a standardized design; (ii) facilitation of the design of concrete software architectures by providing guidelines to application developers; (iii) systematic reuse of common functionalities and configurations throughout the creation of applications; (iv) risk reduction by using architectural elements that have been validated previously during the design of the software reference architecture. In order to specify systematically a reference architecture, guidelines, processes, and methodologies have been proposed (ANGELOV; GREFEN; GREEFHORST, 2012; GALSTER; AVGERIOU, 2011; NAKAGAWA, 2006).

In order to establish Beethoven's reference architecture, a process to specify software reference architectures named Process based on Software Architecture - Reference Architecture (ProSA-RA) (NAKAGAWA, 2006) is used. Figure 7 illustrates ProSA-RA which is composed of the following steps: (Step S-1) domain investigation for identifying software requirements to design a reference architecture; (Step S-2) architectural analysis for establishing a set of architectural requirements; (Step S-3) architectural design for representing a reference architecture using different architectural models; (Step S-4) reference architecture evaluation for validating the specification in reference architectures. Following, each step in presented in details.

Figure 7 – Outline structure of ProSA-RA


Documents
Software systems
Software requirements
Step S-1: Domain investigation
System Analysts
Domain information
Step S-2: Architectural analysis
System Analysts
Arquitectural requirements
Step S-3: Architectural design
Software Architects
Reference architecture
Step S-4: Architecture evaluation
Software Architects
Domain Experts
Evaluated reference architecture

Source**:** Adapted from Nakagawa (2006).

4.1.1 Step S-1: Domain investigation

Based on the ProSA-RA process, the domain investigation step has been performed in order to investigate the declarative business process and service composition domains. The purpose of the domain investigation is to identify and consolidate software requirements to design the proposed reference architecture. In order to perform the domain investigation step, the following steps have performed: (i) identification of domain sources; (ii) identification of functionalities; (iii) identification of domain concepts.

In order to accomplish the aforementioned step, it is necessary to consider several sources of information such as domain tools, domain reference architectures, and domain processes. However, to the best of our knowledge, despite the importance of microservices composition and declarative business processes, there are no tool, reference architecture, or domain process that address both microservices composition and declarative business processes. For this reason, sources of information that have been investigated are divided into three sets: (i) performance characteristics of declarative business processes; (ii) requirements for modeling and execution of declarative business processes; (iii) Service orchestration mechanisms. Next, we describe each set used in the domain investigation.

Set 1 - Performance characteristics of declarative business processes. Goedertier, Vanthienen, and Caron (2015) define a set of performance characteristics for declarative business processes that are used by different process modeling approaches. This set is composed of two performance criteria (effectiveness and efficiency) that can directly impact on the business and financial results of an organization. Beyond the performance criteria, there are additional characteristics for ensuring compliance, flexibility, expressibility, and comprehensibility. Each characteristic of declarative business processes is described as follow:

- **Process flexibility** refers to the organization's ability to apply modifications to their business processes during at both design and run-time. This feature is important since that organizations can have flexibility and adapt to accommodate new market situations (FINK; NEUMANN, 2009; AUSTIN; DEVIN, 2009);
- **Process compliance** refers to a process in which is in correspondence with business rules and business regulation. In this context, business rules are all the internally defined business constraints that are provided by the organization. In contrast, business regulation are all the externally imposed business constraints that are imposed by external regulatory entities;
- **Process effectiveness** refers to a measure of a business process in producing the desired business goals that can be qualitatively evaluated;
- **Process efficiency** refers to a business process that is capable to mitigate utilised/wasted business resources in order to achieve its business goals. Process efficiency may result in cost-efficient, when there is no better manner to organize the work that results in a better cost (e.g., in terms of the total cost of ownership), or time-efficient, when there is no better manner to organize the work that results in a better timer (e.g., in terms of average or variability in lead time);
- **Process expressibility** refers to the ability to express a business process with its specific process elements such as control-flow, data, execution and temporal information (LU; SADIQ, 2007);
- **Process comprehensibility** refers to the ability to express and represent business processes that can be easily understandable among various stakeholders (e.g., application developers, business analysts, and business owners) (FAHLAND *et al.*, 2009).

Set 2 - Requirements for modeling and execution of declarative business process. In the paper written by Vasilecas, Kalibatiene, and Lavbic (2016), the authors have conducted an analysis of five process management systems (IBM Web- sphere (v.7.0 2014)[1], Simprocess (v 2015)[2], Simul8[3], AccuProcess [4] and ARIS 9.7 5[5]) that address declarative business processes in order to identify research opportunities. As results, the authors have identified

[1] <http://www-03.ibm.com/software/products/en/modeler-advanced>
[2] <http://simprocess.com>
[3] <http://www.simul8.com>
[4] <http://bpmgeek.com/accuprocess-business-process-modeler>
[5] <http://www.softwareag.com/corporate/products/new_releases/aris9/more_capabilities/default.asp>

that the analyzed tools have no support for both the modeling and execution in declarative business processes. In this manner, the authors have proposed a set of software requirements for modeling and execution of declarative business processes that present a dynamic mechanism to represent business rules and activities based on internal and external context. According to the authors, an external context is a set of variables and context rules that define a particular state of the environment. In contrast, an internal context is a current state of system resources. The set of requirements that are proposed by the authors are presented as follows:

- **REQ-1**: A declarative business process should not have a predefined sequence of activities.
  - **REQ-1.1**: Every subsequent activity should be selected according to predefined rules and a context. Therefore, every subsequent declarative business process instance may differ from the previous instance of the same declarative business process.
  - **REQ-1.2**: If there is no activity for further execution at a declarative business process runtime, it should be possible to do the following: (i) to terminate the execution of a declarative business process instance; (ii) to define a new activity and concerning rules for a declarative business process instance execution.
- **REQ-2**: Context-based dynamicity.
  - **REQ-2.1**: It should be possible to define an external and internal context.
  - **REQ-2.2**: A declarative business process should react to the change in a context.
- **REQ-3**: Rule-based dynamicity.
  - **REQ-3.1**: It should be possible to define new business rules and to change or delete existing business rules at runtime.
  - **REQ-3.2**: A declarative business process should react to the changes in business rules at process instance runtime.
  - **REQ-3.3**: Every next activity in a declarative business process should be selected according to the predefined business rules.
- **REQ-4**: Any role involved in declarative business process execution should support declarative business process instance change, i.e. change of activities or their sequence, according to the context and rules at runtime with possibly low latency.
- **REQ-5**: Before selecting the next activity, the historical data of instances execution of the same declarative business process should be analysed and the selected next activity should not cause execution of an unacceptable sequence of activities, regarding to early gained experience.
  - **REQ-5.1**: The historical data of each declarative business process instance execution

should be stored in a log file.

  - **REQ-5.2**: It should be possible to define executed instances of a declarative business process as a "good practice” and a "bad practice".
  - **REQ-5.3**: It should be possible to select a suitable instance, labelled as a "good practice", from the historical data for repeated execution.
  - **REQ-5.4**: Time, cost, etc. values should be calculated and stored for each executed declarative business process instance.
  - **REQ-5.5**: Instances named "bad instance” should not be executed.

- **REQ-6**: A declarative business process execution and selection of the next activity at runtime should be based on goal-oriented approach.

Set 3 - Service orchestration mechanisms. Murguzur *et al.* (2014) have conducted a systematic literature review and evaluated seventeen service orchestration approaches after applying the inclusion and exclusion criteria. The systematic literature review objective was to analyze service orchestration approaches from a service orchestration and process flexibility perspectives. The service orchestration perspective is composed of the following properties: language expressiveness, composition model, and execution environment. The process flexibility perspective is formed by the following properties: variability, support for large collections of process modifications, adaptation, need for changes during runtime, evolution, need for structural changes during runtime, and looseness, need for loosely-specified models. Each service orchestration property and process flexibility perspective is detailed as follows.
Service orchestration properties:

- **Language expressiveness:** languages that are used to describe services that enable services to be described semantically and define tasks or goals to perform the service orchestration.
- **Composition model:** languages that are used to define service orchestration and support non-functional requirements.
- **Execution environment:** communication protocol that is used (e.g., SOAP or REST) and technologies that are used for service discovery (e.g., UDDI).

Process flexibility abilities:

- **Variability:** ability to execute a process depending on a particular context. Processes that have the same structure can perform different activities according to contextual information. Thus, the behavior of a process depends on which inputs are assigned to the activities performed.

- **Adaptation:** ability to adapt the behavior of the process and its structure through adaptations. There are two types of devices that can trigger an adaptation in the process: exceptions and unexpected situations. In addition, adaptations may have the following models: planned adaptation and unplanned adaptation.
- **Evolution:** ability of process instances to change as the corresponding process structure evolves. Since processes may change over time, it is necessary to provide mechanisms to support and apply evolutionary changes in new and running instances.
- **Looseness:** ability to specify only part of the process during design-time in order to deal with unpredictability, non-repeatability and emergence. In other words, this capability allows to specify fragments of a process that during its execution will be completed from the execution of its activities.

In addition, Murguzur *et al.* (2014) have identified and characterized the following service orchestration techniques: (i) workflow-based techniques that aim to create a workflow for later execution; (ii) artificial intelligence planning techniques that aim to resolve service composition by applying algorithms for solving a planning problem to determine which action should be performed in order to achieve a final goal.

4.1.2 Step S-2: Architectural analysis

Based on information identified in the previous step, a set of architectural requirements for the Beethoven's reference architecture has been established. The set of architecture requirements is classified into two subsets: source and concept. The subset source refers to the information used in the previous step (Set 1: performance characteristics of declarative business processes; Set 2: requirements for modeling and execution of a declarative business process; and Set 3: Service orchestration mechanisms). The subset concept refers to requirements that are related to declarative business process (DBP), requirements related to event-driven architectures (EDA), and requirements related to Quality of Service properties (QoS).

Table 4 illustrates the Beethoven reference architecture requirements. The columns refer to, from left to right, the requirement identification, the requirement description, the source of information related to requirements, and the concepts related to the requirements.

Table 4 – Beethoven reference architecture requirements

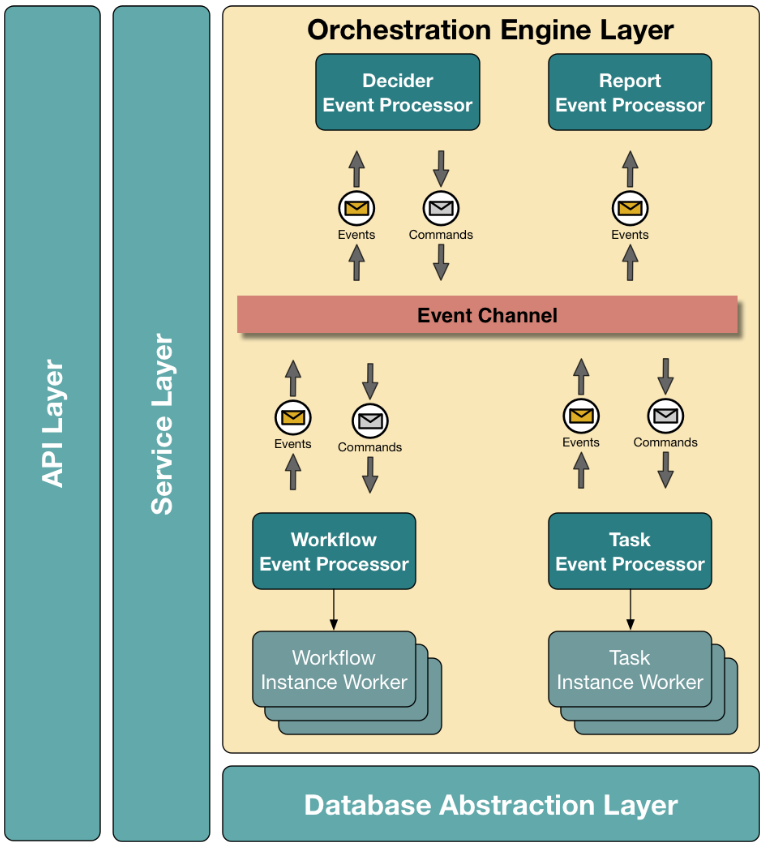
ID	Requirement	Source	Concept
REQ-1	The reference architecture must provide mechanisms to measure process efficiency and effectiveness that can be qualitatively evaluated.	Set 1	QoS
REQ-2	The reference architecture must support metrics related to non-functional requirements.	Set 3	QoS
REQ-3	The reference architecture must provide architectural elements to produce, detect, consume, and react to events that may occur during a declarative business process execution.	Set 1, Set 2, Set 3	EDA
REQ-4	The reference architecture must maintain data consistency across multiple instances of declarative business processes without using distributed transactions.	Set 1, Set 2, Set 3	EDA
REQ-5	The reference architecture must guarantee process compliance during an execution of business processes.	Set 1	DBP
REQ-6	The reference architecture must provide process flexibility in order to apply modifications to an specification of declarative business process during both at design-time and at run-time.	Set 1, Set 2	DBP
REQ-7	The reference architecture must have a contextual framework in order provide context information for instance of declarative business processes.	Set 2	DBP
REQ-8	The reference architecture must support communication protocols that are commonly used in microservices-based application.	Set 3	EDA

Source**:** Elaborated by the author.

4.1.3 Step S-3: Architectural design

In order to design a reference architecture, different architectural models and representation techniques should be used for representing different views of an architecture (NAKAGAWA, 2006). Based on the concepts proposed in the ProSA-RA process, the following resources have been chosen to represent Beethoven's reference architecture: (i) types of knowledge/elements: lifecycle of a declarative business process and interaction between architectural components; (ii) technique of representation: block diagram, layers diagrams, and UML diagrams. Figure 8 depicts the general representation of Beethoven's reference architecture using a block diagram (modules). This architecture is a 4-layer architecture composed by API, Service, Database Abstraction, and Orchestration Engine layers, which are described as follows.

Figure 8 – Beethoven's reference architecture



Source: Elaborated by the author.

4.1.3.1 API Layer

This layer provides a uniform interface and endpoints to standardize the Beethoven's Service Layer access. Such interfaces and endpoints are specified following the RESTful standards and conventions, granting a technology-agnostic access. In order to provide process flexibility, applying modifications to a specification of a declarative business process during both design-time and run-time, the API layer offers RESTful endpoints to maintain workflow specifications. For instance, to add a new event handler to an existing workflow, an HTTP POST request must be performed to the URL `"api/workflows/workflowName/handlers"` using as a request body an event handler definition. In this example, the expression `"{workflowName}"` represents a variable URL for the workflow name. The endpoints offered by the Layer API are described in Appendix B, according to their respective HTTP methods, URLs, request parameters and request body.

4.1.3.2 Service Layer

This layer provides a controlled access to the other layers in the Beethoven's reference architecture (i.e. Database Abstraction and Orchestration layers). This layer implements all services that are consumed by the API layer (e.g., create a new event handler, delete a particular task, and so on). In addition, the services offered by this layer can be used to build applications for managing, monitoring, and visualizing the workflow execution.

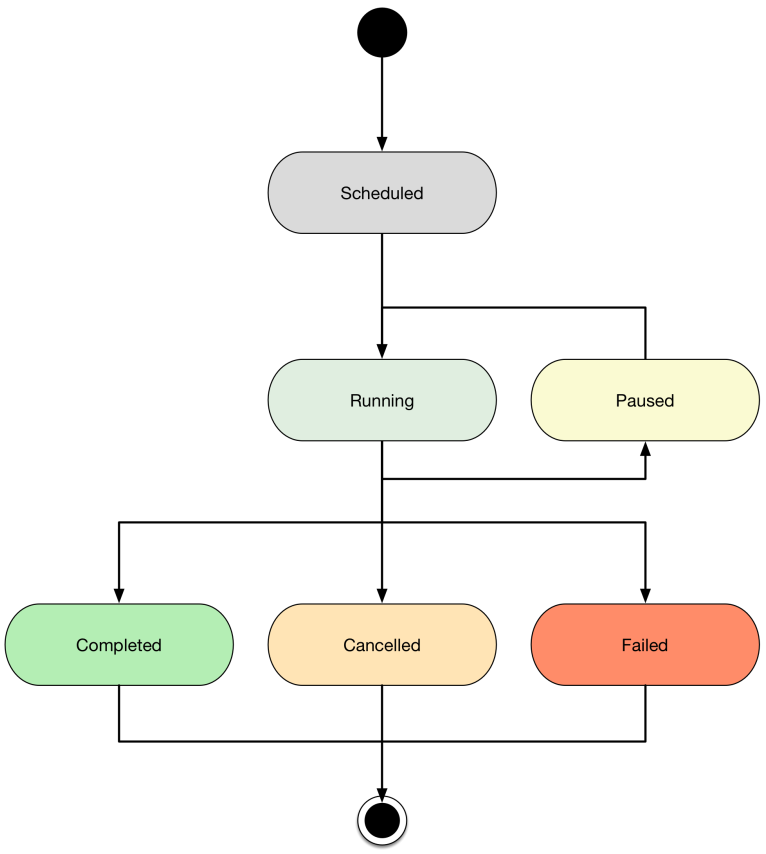
The Service Layer provides a set of services to handle the execution of workflow instances. For example, it is possible to start a workflow instance; later, pause the running workflow instance; then, after a period of time, start the workflow instance from the state in which it was paused. These commands affect the life cycle of a workflow instance. The workflow instance is created after scheduling a workflow definition.

Figure 9 presents the states that compose the life cycle of a workflow instance. The first state that a workflow instance can assume is *scheduled*. Once scheduled, the Beethoven platform should start the execution of the workflow instance (the *running* state). During the execution of the workflow instance, it is possible to pause (the *paused* state) and restart the workflow instance. After performing all tasks successfully, the workflow instance is completed (the *completed* state). If one task fails and this fault is not handled, then the execution of the workflow instance will fail (the *failure* state). Another possibility to interrupt the execution of the workflow instance is through the cancel command (the *canceled* state).

4.1.3.3 Database Abstraction Layer

This layer is used to store the workflow, task, and event handler definitions. All information concerning the workflow must be stored in order to create workflow instances. Runtime changes in the workflow definition will result in the creation of new workflow instances that have the new version of the workflow definition. The Database Abstraction Layer is also responsible for tracking and recording information about different workflows execution, such as resource utilization, throughput, and execution time. Such information can be used in a further analysis to identify, for instance, the existence of bottleneck or failures when performing certain tasks.

Figure 9 – Life cycle of a workflow instance



Source**:** Elaborated by the author.

4.1.3.4 Orchestration Engine Layer

This is the architecture core layer and follows the event-driven architectural style to provide a workflow execution mechanism in a decoupled and scalable manner. It is composed of three main architectural components: Event Channel, Event Processor (Decider, Report, Workflow, and Task), and Instance Work (Task and Workflow). The Event Channel component is used as an event bus to exchange messages between Event Processor components and can be implemented as message queues, message topics, or a combination of both. The Event Processor component is responsible for processing a specific type of event and notifying a successful or failure execution by publishing another event in the Event Channel. An Event Processor component can be bound to a set of Instance Work component instances. The Instance Work component is responsible for performing a specific activity (e.g., decision, reporting, workflow, or task) demanded by the Event Processor component to which it is bound.

The Orchestration Engine, as shown in Figure 8, is composed by an event channel, used to exchange event messages between event processors, which can be of four types: Decider Event Processor, Report Event Processor, Workflow Event Processor, and Task Event Processor.

Specifically, in the event process context, there are two types of events: an *event* (occurrence of a particular action) and a *command* (an action to be performed). In the Orchestration Engine, each event processor can send or receive events and commands to or from another event processor using the Event Channel. In Table 5 and 6, all events and commands are described by their respective identifiers, sources, targets, and descriptions.

The Workflow Event Processor can send events during execution of workflow instances, such as *WorkflowScheduledEvent*, *WorkflowStartedEvent*, and *WorkflowCompletedEvent*, that represent events for scheduling, starting and completing a workflow, respectively. In addition, it can also receive commands to control the execution of a workflow instance, such as *StartWorkflowCommand*, *StopWorkflowCommand*, and *CancelWorkflowCommand*, for starting, stopping and canceling a workflow, respectively. After receiving the command to start a workflow, the Workflow Event Processor creates a workflow instance and a worker (Workflow Instance Worker), which is responsible for handling all events and commands related to the workflow instance that is created by the event processor. In this way, multiple workflow instances can be executed in parallel isolatedly.

The Task Event Processor sends events during running tasks (*TaskStartedEvent*, *TaskCompletedEvent*, *TaskFailedEvent*, and *TaskTimeoutEvent*) and receives commands to execute tasks (*StartTaskCommand*). In summary, the Workflow Event Processor and Task Event Processor delegate the activity of managing the events and commands to their respective workers, providing scalability to the proposed architecture.

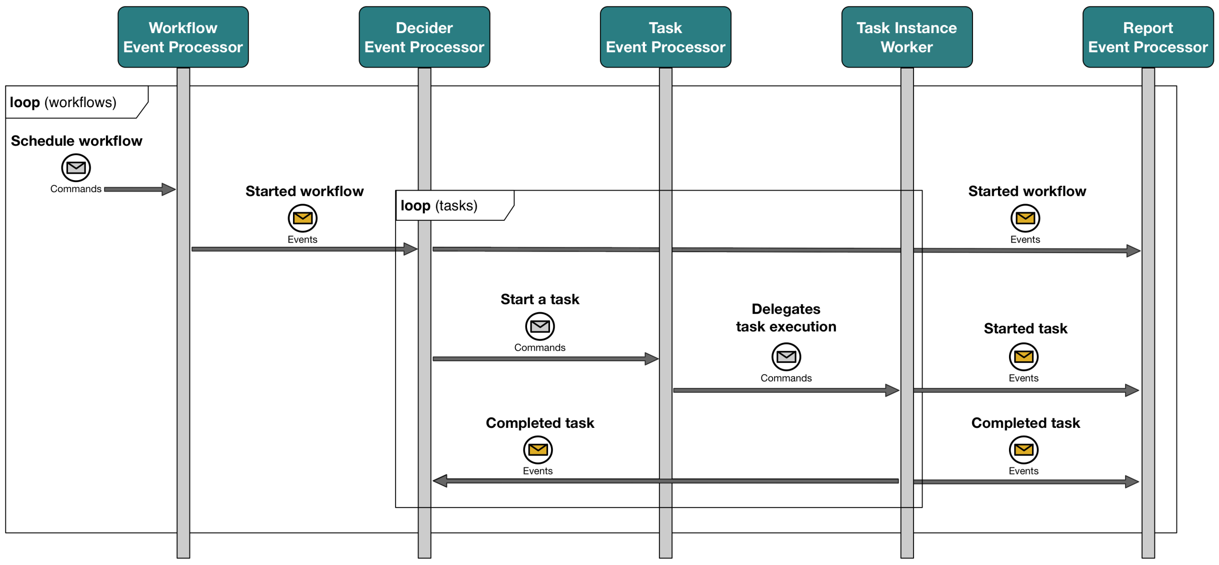
The Decider Event Processor receives events generated by the Task Event Processor and Workflow Event Processor (e.g., *WorkflowCompletedEvent* or *TaskCompletedEvent*) to decide which action (command) should be performed. To this end, the Decider Event Processor uses event handlers definitions in order to evaluate ECA rules. For instance, if a specific event occurs during the execution of a workflow instance, and an evaluated condition is true, then a particular command or a set of command must be performed.

The Report Event Processor receives all events that are generated by running workflow instances with their respective tasks to record metrics (e.g, execution time) on each instance. Furthermore, additional information about failures or timeouts in the task execution are also written from the events received by the processor. This way, information about the execution time of each task or the entire workflow can be consulted later using the API and Service layers.

The sequence diagram, shown in Figure 10, illustrates the interaction between the Orchestration Engine components in the sequential order that those interactions may occur. For the sake of simplicity, some events that are triggered during the workflow execution and do not affect the understanding of the Orchestration Engine have been omitted.

Firstly, while there are workflows to be executed, the Workflow Event Processor will receive a command to schedule a particular workflow. The scheduled workflow waits until the Workflow Event Processor creates a workflow instance and starts running it, sending an event about the started workflow to the Decider Event Processor and Report Event Processor. If there is a task to be performed, the Decider Event Processor sends a command to start a particular task to the Task Event Process. After that, the Task Event Processor delegates the responsibility for executing the task to the Task Instance Worker, in which sends events about started tasks to the Report Event Processor. When that task is completed, the Decider Event Processor receives an event, evaluates its conditions, and if there are more tasks to be executed that satisfies the specified conditions, it sends a command to start another task. During the execution of each workflow, all events and commands exchanged by the event processors are forwarded to the Report Event Processor.

Figure 10 – Interaction between the Orchestration Engine components



Source**:** Elaborated by the author.

4.1.4 Step S-4: Reference architecture evaluation

In contrast to reference architectures that are designed to facilitate system design and development in multiple projects, concrete software architectures are designed in a specific context and reflect concrete business goals of the stakeholders (ANGELOV; GREFEN;

Table 5 – List of events

Identifier	Source	Description
WorkflowScheduledEvent	Workflow Event Processor	The workflow instance execution was scheduled. It takes the workflow name as input.
WorkflowCompletedEvent	Workflow Event Processor	The workflow instance execution was started. It takes the workflow name and the instance name as input.
WorkflowStartedEvent	Workflow Event Processor	The workflow instance execution was scheduled. It takes the workflow name as input.
WorkflowFailedEvent	Workflow Event Processor	The workflow execution closed due to a failure.
WorkflowCanceledEvent	Workflow Event Processor	The workflow execution was successfully canceled and closed.
TaskStartedEvent	Task Event Processor	The task was dispatched to a worker. It takes the task name, task input, instance name, and workflow name as input.
TaskCompletedEvent	Task Event Processor	The task was successfully completed. It takes the task name, task output, instance name, and workflow name as input.
TaskTimeoutEvent	Task Event Processor	The task timed out. It takes the task name, instance name, and workflow name as input.
TaskFailedEvent	Task Event Processor	The task failed. It takes the task name, task output, instance name, and workflow name as input.

Source**:** Elaborated by the author.

Table 6 – List of commands

Identifier	Target	Description
ScheduleWorkflowCommand	Workflow Event Processor	Schedules a workflow in the queue. It takes the workflow name as input.
StartWorkflowCommand	Workflow Event Processor	Starts a workflow that was stoped. It takes the workflow instance name as input.
StopWorkflowCommand	Workflow Event Processor	Stops a running workflow. It takes the workflow instance name as input.
CancelWorkflowCommand	Workflow Event Processor	Cancels a running workflow. t takes the workflow instance name as input.
StartTaskCommand	Task Event Processor	Starts a task execution. It takes the task name, task input, workflow name, and instance name as input.

Source**:** Elaborated by the author.

GREEFHORST, 2012). Thus, concrete software architectures can be used to validate the specification in reference architectures. Aiming at observing the viability of the Beethoven's reference architecture, as well as its capability to execute declarative business processes, a concrete architecture have been implemented and it presented in Chapter 5.

4.2 ORCHESTRATION DSL

This section introduces a textual DSL named Partitur for microservices orchestration based on declarative business processes. The chapter focuses on the design and specification of the Partitur language, presenting its characteristics and formal definition. By using Partitur, it possible to specify declarative business processes that follow the structure of ECA rules. A business process in Partitur is composed of three definitions: Workflow, Task, and Event Handler. Each definition is described by using syntax diagrams and examples.

4.2.1 Partitur language design

Partitur is a textual DSL for creating declarative business processes in order to compose a collection of microservices. Partitur is built using Xtext[6], a tool based on Eclipse Modeling Framework for the development of programming languages and domain-specific languages. That covers many aspects of a programming language infrastructure including a parser, linker, typechecker, compiler, and sophisticated Eclipse IDE integration. We have chosen to use the Eclipse platform and Xtext for Partitur implementation because they are both open source and mature technologies used extensively in both academia and industry. In the next subsection, we present the main Partitur definitions: Workflow, Task, and Event Handler. The completed grammar definition of Partitur is presented in Appendix A.

4.2.2 Partitur workflow definition

A workflow is an abstraction of a business process that is executed in a distributed manner among different microservices. Different from some imperative approaches presented in Chapter 3, a workflow defined in Partitur follows a declarative approach and is composed of a set of activities that may eventually be executed and a set of constraints that must be obeyed. In this way, the definition of a workflow is designed to ensure that all activities performed during a business process are in accordance with the business constraints that have been specified.

Declarative business processes have a structure composed of two central elements: business activities and business constraints. A constraint is a business rule that must be respected

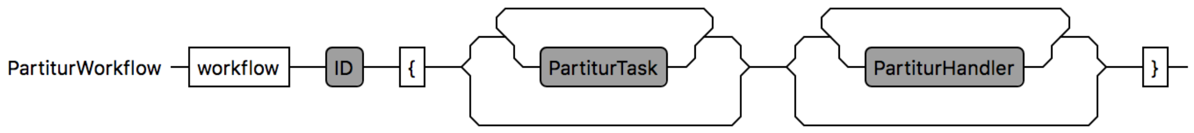
[6] <https://www.eclipse.org/Xtext/>

during the process execution. A business activity is an action that manages a business resource. Since the Partitur language is based on declarative business processes, the Partitur workflow definition provides declarative process elements such as a unique identifier (workflow name), a set of tasks (business activities), and a set of event handlers (business constraints). Each Partitur workflow element is defined as follows:

- A **unique identifier** that represents the workflow name;
- A **set of tasks** that contains the business tasks or activities that may be performed during the execution of a business process;
- A **set of event handlers** that contains all the business constraints.

The Partitur workflow definition begins with the keyword **`workflow`**, followed by a unique identifier, an optional set of task definitions, and an optional set of event handler definitions. Figure 11 represents the syntax diagram of the Partitur workflow definition.

Figure 11 – Partitur workflow definition



Source: Elaborated by the author.

Code 1 provides an incomplete example of workflow definition in Partitur. The workflow is identified by `workflowName` and is composed of two tasks (`taskName1` and `taskName2`) and two event handlers (`h1` and `h2`).

Code 1 – Sample of workflow

```
workflow workflowName {
    task taskName1 {...}
    task taskName2 {...}
    handler h1 {...}
    handler h2 {...}
}
```

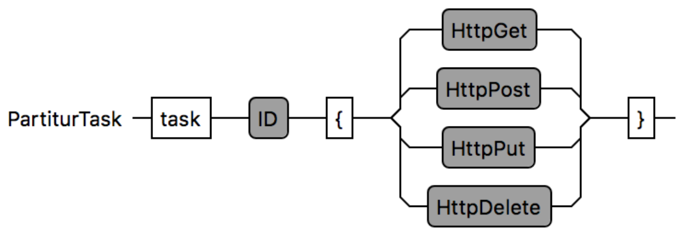
4.2.3 Partitur task definition

In terms of declarative business processes, a business activity is an action that manages a business resource. In Partitur, a task is an atomic and asynchronous operation (business activity) responsible for performing an action that manages a microservice (business resource). A Partitur task is composed of a unique identifier and an HTTP request. Each task element is described as following:

- A **unique identifier** that represents the task name;
- A **HTTP request** that represents one of the following HTTP methods: DELETE, GET, POST, and PUT.

Figure 12 represents the syntax diagram of the Partitur task definition that begins with the keyword **`task`**, followed by a unique identifier, and an HTTP request. Code 2 provides an example of task definition. In this example, the task is identified by `createNewBook` and is composed by one HTTP request that should be performed to a specific Uniform Resource Identifier (URI) resource.

Figure 12 – Partitur task definition



Source**:** Elaborated by the author.

Code 2 – Sample of task

```
task createNewBook {
    post("http://book-service/books")
}
```

4.2.3.1 Contextual input and HTTP utility methods

Partitur provides higher level utility methods that are built in the DSL in order to perform HTTP requests during task executions. The HTTP methods supported by Partitur are: DELETE, GET, POST, and PUT. The Partitur HTTP methods follow a code style convention named *FluentInterface* (FOWLER, 2005). Furthermore, these methods facilitate the invocation of RESTful-based microservices and enforce REST principles (Resources, Representations, Messages, and Stateless) (FIELDING; TAYLOR, 2002).

In addition to DSL methods for invoking HTTP requests, the Partitur language provides a data-driven approach for executing tasks. This approach is model of execution based on the concept of Data-Driven Multithreading (DDM) (KYRIACOU; EVRIPIDOU; TRANCOSO, 2006) and provides a scheduler that executes a task after all of its required data have been produced. As a result, no synchronization or communication latencies are experienced after a task begins its execution. In order to implement the concept of DDM, the concept of contextual inputs is introduced.

A contextual input is a data-driven mechanism for specifying the input of data to a specific task from the result obtained after performing another task. For example, in order to execute a task for the creation of purchase order, a task must receive the result from another task responsible for calculating the shipping value. As a result, contextual inputs can be a powerful mechanism that enables flexibility and non-blocking execution in the definition of data-driven tasks.

Contextual inputs can be used as request body, query params, URI variables, and HTTP headers. In order to use contextual inputs, a user must apply the following command: "${contextualInputIdentifier}". The contextual input command is composed of an expression (${...}), which involves a contextual input identifier. Furthermore, a contextual input is identified dynamically by the Beethoven platform that will provide dynamic access to data from a particular contextual input identifier. Code 3 illustrates samples of contextual inputs.

Code 3 – Sample of contextual inputs

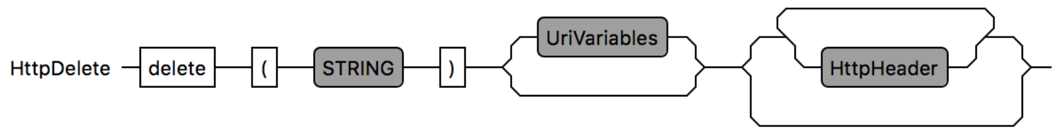
```
body("${calculateShippingFee.response}")
queryParam("fee", "${calculateShippingFee.response}")
uriVariables("${calculateShippingFee.response}")
header("fee", "${calculateShippingFee.response}")
```

As follows, it is described how to specify each Partitur utility method to perform HTTP requests:

Partitur HTTP delete - The Partitur delete method represents an HTTP DELETE request and should be used to delete the specified resource. The delete method definition begins the with the term `delete`, follow by the resource URI, an optional set of URI variables, and an optional set of HTTP headers. Figure 13 represents the syntax diagram of the Partitur delete definition. Code 4 represents two manners to express the Partitur delete method for removing a consumer (resource) with an ID of 6732. In the first definition (Lines 1-2), the resource ID is specified through the URI. In the second definition (Lines 4-6), the resource ID is specified using URI variables. In addition, in both examples, the delete method utilizes an HTTP header for authorization bearer access token that is part of OAuth 2.0 protocol[7].

[7] <https://oauth.net/2/>

Figure 13 – Partitur delete method definition



Source**:** Elaborated by the author.

Code 4 – Example of the Partitur delete method

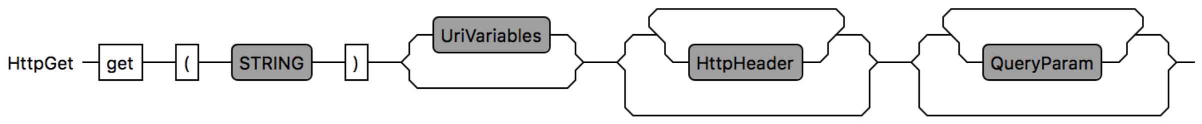
```
delete("http://consumer-service/consumers/6732")
    .header("Authorization", "Bearer token")

delete("http://consumer-service/consumers/")
    .uriVariables("6732")
    .header("Authorization", "Bearer token")
```

Partitur HTTP get - The Partitur get method is an abstraction of an HTTP GET request that is responsible for retrieving information from a resource URI. It is possible to use URI variables. In addition, it is possible to send query strings using the method `queryParams` that receives two arguments: the name of the query param and the value of the query param. The Partitur get method also supports HTTP headers. The Partitur get definition is composed of the term `get`, followed by the resource URI, an optional set of URI variables, an option set of HTTP headers, and an optional set of query params. Figure 14 represents the syntax diagram of the Partitur get definition.

Code 5 represents an instance of the get method that is used to retrieve a consumer by sending the following URL parameters: the parameter name with "George" as value; the parameter birthday with “04-28-1988" as value; the parameter status, which is a contextual input, with "${checkStatus.response}" as value. In addition, the header for authorization bearer access token is used in the example.

Figure 14 – Partitur get method definition



Source**:** Elaborated by the author.

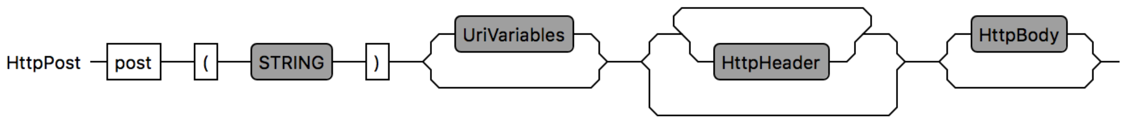
Code 5 – Example of the Partitur get method

```
get("http://consumer-service/consumers")
    .header("Authorization", "Bearer token")
    .queryParams("title", "George")
    .queryParams("birthday", "04-28-1988")
    .queryParams("status", "${checkStatus.response}")
```

Partitur HTTP post - The Partitur post method creates a new resource by submitting a given entity to a resource URI. It represents an HTTP POST method that supports URI variables, HTTP headers, and a request message body. In Partitur post method, it is possible to send a resource by using the `body` method. The definition of the post method is specified by the term `post`, followed by the resource URI, an optional set of URI variables, and an optional body. Figure 15 represents the syntax diagram of the Partitur post definition.

Code 6 is an example of Partitur post method in which a JSON file is sent to the consumer resource in order to create a new consumer. In addition, it uses HTTP headers for content type and authorization token.

Figure 15 – Partitur post method definition



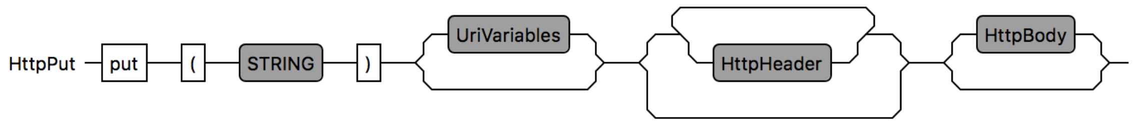
Source: Elaborated by the author.

Code 6 – Example of Partitur post method

```
post("http://consumer-service/consumers/")
    .header("Authorization", "Bearer token")
    .header("Content-Type", "application/json")
    .body(json)
```

Partitur HTTP put - The Partitur put method should be used to update a resource from a given URI. It represents the HTTP PUT method. In the same way, as in the post method, it supports URI variables, HTTP headers, and a request message body. The definition of put method is specified by the term `put`, followed by the resource URI, an optional set of URI variable, and an optional body. Figure 16 represents the syntax diagram of the Partitur put definition. Code 7 is an example of Partitur put method in which a contextual input is used to update an existing consumer. In addition, it uses an HTTP headers for the content type and authorization token.

Figure 16 – HTTP PUT definition



Source: Elaborated by the author.

Code 7 – Sample of HTTP PUT

```
put("http://consumer-service/consumers/")
    .header("Authorization", "Bearer token")
    .header("Content-Type", "application/json")
    .body("${createNewConsumer.response}")
```

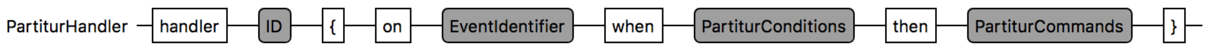
4.2.4 Partitur event handler definition

Partitur event handlers are based on ECA rules for programming business constraints in declarative business processes. As mentioned before in Section 2.5, ECA rules provide an intuitive and powerful paradigm for programming reactive systems and its fundamental construct forms the following structure: on *Event* if *Condition* do *Action*. In other words, ECA rules are composed of the following structure: when *Event* occurs, if *Condition* is verified, then execute *Action*. In order to define a Partitur event handler, the following structure is used: an event handler name, an event listener (*Event*), a set of conditions (*Condition*), and a set of commands (*Action*). Each part of the event handler definition is explained below:

- A **unique identifier** that represents the event handler name;
- **Event listeners** are used to define which events must be listened and captured during the workflow execution;
- **Conditions** define which parameters must be true in order to process an event. Otherwise, events will not be processed. There are the following conditions that may be used: *workflowNameEqualsTo*, *taskNameEqualsTo*, and *taskResponseEqualsTo*;
- **Commands** define which action should be performed on the occurrence of an event that satisfies the specified conditions. For example, the *startTask* command is used to execute a task by its name and an optional input value. There are the following available commands: *startTask*, *scheduleWorkflow*, *startWorkflow*, *stopWorkflow*, and *cancelWorkflow*.

The Partitur event handler definition is specified by the keyword **`handler`**, followed by a unique identifier, the event listener definition, a set of condition definitions, and a set of command definitions. The definition of event listeners, conditions, and commands in Partitur are described in the following subsections. Figure 17 represents the syntax diagram of the Partitur event handler definition.

Figure 17 – Partitur event handler definition



Source**:** Elaborated by the author.

In Code 8, it is illustrated how to declare an event handler in Partitur. In this example, an event handler is declared to listen for the `WORKFLOW_SCHEDULED` event. If this event is caught, the `workflowNameEqualsTo("newConsumerProcess")` condition will be evaluated. If the evaluated condition is true, the `startTask("createNewConsumer")` command will be executed.

Code 8 – Example of event handler

```
handler h1 {
    on WORKFLOW_SCHEDULED
    when workflowNameEqualsTo("newConsumerProcess")
    then startTask("createNewConsumer")
}
```

4.2.4.1 Event listener definition

The definition of an event listener is composed by the keyword **`on`** and an event identifier. Event identifiers are constants that represent all possible events that can be triggered during the execution of a workflow. For instance, after a workflow has started, an event represented by WORKFLOW_STARTED is triggered. In addition, event identifiers are divided into two categories: task events and workflow events. The former represents the events related to executions of tasks, while the latter designates the events associated with executions of workflows. Table 7 describes all the event identifiers.

4.2.4.2 Condition definition

The definition of a condition is composed of the keyword **`when`** and a set of Partitur conditions. In particular, a Partitur condition is a utility method built in the proposed DSL in order to evaluate boolean expressions. The available condition are `taskNameEqualsTo`, `taskResponseEqualsTo`, and `workflowNameEqualsTo`. The description for each Partitur condition is presented in Table 8.

Table 7 – List of Partitur event identifiers

Event identifier	Category	Description
TASK_STARTED	Task Event	It indicates that a task has started.
TASK_COMPLETED	Task Event	It indicates that a task has completed.
TASK_TIMEDOUT	Task Event	It indicates that a task has timed out.
TASK_FAILED	Task Event	It indicates that a task has failed.
WORKFLOW_SCHEDULED	Workflow Event	It indicates that a workflow has been scheduled.
WORKFLOW_STARTED	Workflow Event	It indicates that a workflow has started.
WORKFLOW_COMPLETED	Workflow Event	It indicates that a workflow has completed.

Source**:** Elaborated by the author.

Table 8 – List of Partitur conditions

Condition	Description	Parameters
`taskNameEqualsTo`	Condition responsible for evaluating whether the captured event was triggered by the specified task.	**taskName** - Unique identifier representing the task name that must be verified.
`taskResponseEqualsTo`	Condition responsible for evaluating whether the response of a particular task satisfies a JSONPath expressions.	**jsonPath** - A JSONPath expression. **matcher** - A function to check for conditions.
`workflowNameEqualsTo`	Condition responsible for evaluating whether the captured event was triggered by the specified workflow.	**workflowName** - Unique identifier representing the workflow name that must be verified.

Source**:** Elaborated by the author.

Code 9 illustrates examples of conditions in Patitur. The first condition verifies if the task name is equal to the string "taskName". The second condition verifies if the member of JSON object that is referred by the JsonPath expression "$.consumers" has the size is equal to 100. The third condition verifies if the workflow name is equal to the string "workflowName".

Code 9 – Example of conditions

```
when taskNameEqualsTo("taskName")
when taskResponseEqualsTo("$.consumers", hasSize(100))
when workflowNameEqualsTo("workflowName")
```

In order to understand the `taskResponseEqualsTo` condition, two concept must be introduced: JsonPath expressions and a Partitur matchers. First, JsonPath[8] is a language used to read, query, and manipulate JSON objects using JSON-based expressions. By using JsonPath

[8] <https://github.com/json-path/JsonPath>

expressions, it is possible, for example, to query and select a specific property of a JSON object in order to perform boolean expressions. Second, A Partitur matcher is also an internal method built in the proposed DSL for evaluating boolean expressions. Partitur matchers perform boolean verifications in a JSON object that is queried by JsonPath expressions. For instance, in order to implement a business constraint that needs to check if a smartphone price is less than 500 dollars, the following condition should be specified:

Code 10 – Example of Partitur matcher

```
taskResponseEqualsTo("$.smartphone.price", lessThan(500))
```

In addition, Code 10 illustrates how to use the condition `taskResponseEqualsTo`. This condition is composed of two parameters. The first one refers to a JsonPath expression that selects the property price from a JSON object representing a smartphone, while the second one represents a parameter matcher that is used to verify whether the selected value of a JSON object is less than 500. The available Partitur matchers are described in Table 9.

Table 9 – List of Partitur matchers

Matcher	Description	Parameters
`not`	It is a unary operation that inverts the logic by negation.	**matcher** - the Partitur matcher that should be inverted.
`lessThan`	It verifies if a examined value is less than the specified value.	**value** - the value that should be compared.
`equalTo`	It verifies if a examined value is equal to the specified value.	**value** - value - the value that should be compared.
`greaterThan`	It verifies if a examined value is greater than the specified value.	**value** - value - the value that should be compared.
`nullValue`	It verifies if the examined object is null.	There is no parameters.
`empty`	It verifies if a collection is empty.	There is no parameters.
`hasItem`	It verifies if the item is contained in a examined collection.	**item** - the item to compare against the items of an examined collection.
`hasSize`	It verifies if the size of a collection has the specified size.	**size** - the expected size of an examined collection.

Source: Elaborated by the author.

Partitur conditions, JsonPath expressions, and Partitur matchers provide an effective mechanism to specify business constraints that must restrict the execution of tasks in a declarative business process. Therefore, business constraints can be described at different levels of complexity. For instance, it is possible to define a business process that is constituted of a simple sequence of task invocation by using only the `taskNameEqualsTo` condition. In contrast, it possible to define a business process that is composed of complex business rules using the condition `taskResponseEqualsTo`.

4.2.4.3 Partitur command definition

Partitur commands are the last part of a declarative business process that follows the ECA structure. A Partitur command definition is determined by the keyword **`then`** followed by a set of Partitur commands. Partitur commands are methods build into the language in order to perform actions that must be taken on tasks or workflows. For example, the command `startTask` receives as a parameter the name of the task to be executed. Commands are indivisible operations that can be executed successfully or not. The list of Partitur commands are composed of `startTask`, `startWorkflow`, `stopWorkflow`, `cancelWorkflow`, `finishWorkflow`. Each Partitur command and its parameters are described in Table 10.

Table 10 – List of Partitur commands

Command	Description	Parameters
`startTask`	Command responsible for starting a task.	**taskName** - Unique identifier representing the task name that must be started.
`startWorkflow`	Command responsible for starting a workflow	**workflowName** - Unique identifier representing the workflow name that must be started.
`stopWorkflow`	Command responsible for stopping a workflow.	**workflowName** - Unique identifier representing the workflow name that must be stopped.
`cancelWorkflow`	Command responsible for canceling a workflow.	**workflowName** - Unique identifier representing the workflow name that must be calceled.
`finishWorkflow`	Condition responsible for finishing a workflow.	**workflowName** - Unique identifier representing the workflow name that must be finished.

Source**:** Elaborated by the author.

Code 11 presents the definition of five commands. The first command represents an action to start the task named "taskName". The second command defines an action to start a workflow named "workflowName". The third command specifies an action to stop the execution of the workflow named "workflowName". The fourth command represents a command to cancel the execution of the workflow named "workflowName". Finally, the fifth command represents an action to terminate the execution of a workflow.

Code 11 – Example of commands

```
then startTask("taskName")
then startWorkflow("workflowName")
then stopWorkflow("workflowName")
then cancelWorkflow("workflowName")
then finishWorkflow("workflowName")
```

5 SPRING CLOUD BEETHOVEN

This chapter presents the concrete architecture, named Spring Cloud Beethoven, that has been implemented based on the specification of the Beethoven's reference architecture. The remainder of this chapter presents the Spring Cloud ecosystem, details about the concrete architecture implementation, the basic concepts and common terminology related to the actor model, a formal description of each actor that composed the concrete architecture, and a methodology that describes how to use Spring Cloud Beethoven in MSA-based applications.

5.1 SPRING CLOUD ECOSYSTEM

Spring Cloud[1] is an umbrella project that provides a set of components for software engineers to develop cloud-native applications based on common patterns in distributed systems (e.g., distributed configuration management, service registration and discovery, circuit breakers, intelligent routing, a control bus, leadership election, distributed sessions, cluster state, and so on). Spring Cloud solutions provide integrations for Spring Boot applications by applying autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. As follows, a subset of the main Spring Cloud solutions is described.

- **Spring Cloud Bus**[2] provides a lightweight message broker for connecting nodes of a distributed system. The message broker is implemented using an AMQP broker as the transport mechanism;
- **Spring Cloud Netflix**[3] provides Netflix OSS integrations for MSA-based applications using the best practices and patterns presented in Section 2.2.2;
- **Spring Cloud for Amazon Web Services**[4] provides a module set to consume AWS services and reduce the infrastructure related code in Spring-based applications;
- **Spring Cloud Config**[5] provides server and client-side support for externalized configuration in a distributed system. To this end, it provides a central microservice to manage external configuration properties for applications across all environments (e.g., debug, testing, and production);
- **Spring Cloud Sleuth**[6] provides a distributed tracing solution for distributed applications

[1] <http://projects.spring.io/spring-cloud/>
[2] <https://cloud.spring.io/spring-cloud-bus/>
[3] <https://cloud.spring.io/spring-cloud-netflix/>
[4] <https://cloud.spring.io/spring-cloud-aws/>
[5] <https://cloud.spring.io/spring-cloud-config/>
[6] <https://cloud.spring.io/spring-cloud-sleuth/>

by instrumenting external systems automatically and capturing data in logs or a remote collector service.

Netflix, a pioneer in the microservices domain, has built many tools that are available as open-source software under the Netflix OSS platform. In particular, Spring Cloud solutions have been initially based on the technology stack provided by Netflix OSS in order to provide integration with Spring Boot applications. The Netflix OSS platform is composed of a set of microservices patterns such as a service discovery through Eureka, distributed configuration through Archaius, resilient and intelligent inter-process and client side load balancer through Ribbon, and isolate latency and fault tolerance at runtime through Hystrix. Each Netflix OSS solution addresses a specific problem that MSA-based applications may face.

Although there are solutions for different common problems presented in MSA-based applications, there is no solution to address microservice composition problems. Since the concrete architecture is based on the Spring Cloud Netflix ecosystem, it provides integration for Spring Boot applications using auto-configuration and binding to Spring Cloud Netflix components, such as Spring Cloud Eureka (service discovery), Spring Cloud Ribbon (client-side load balancer), and Spring Cloud Hystrix (circuit breaker). Therefore, in order to address dynamic microservices location, Spring Cloud Beethoven relies on Spring Cloud Eureka for service discovery and Spring Cloud Ribbon for client-side load balancer. As consequence, there is no need to describe previously and register each microservice that will be part of the orchestration. Thus, new microservices that are added to a microservices-based application can be used during the microservice composition.

5.2 CONCRETE ARCHITECTURE

In contrast to reference architectures that are designed to facilitate system design and development in multiple projects, concrete software architectures are designed in a specific context and reflect concrete business goals of the stakeholders (ANGELOV; GREFEN; GREEFHORST, 2012). Thus, concrete software architectures can be used to validate the specification in reference architectures. Aiming at observing the viability of the Beethoven's reference architecture, the concrete architecture, named Spring Cloud Beethoven, has been implemented based on the specification of the Beethoven's reference architecture and the Spring Cloud Netflix ecosystem. Specifically, in order to implement Spring Cloud Beethoven, the following technologies have been used: Java programming language[7], Spring Cloud Netflix[8], and Akka toolkit[9].

The concrete architecture has been implemented to run as an infrastructure microservice using a minimal setup in order to facilitate its integration with microservices-based applications. For instance, in order to initialize Spring Cloud Beethoven, there is a set of Java annotations that must be used in a Spring Boot application for configuring the concrete architecture automatically. A complete step by step guide for use of the concrete architecture is presented in Section 5.5. Since Spring Cloud Beethoven is an infrastructure microservice, it can be accessed directly through its RESTful API. Therefore, in a microservices-based application, any microservice can manage workflows (create, delete, start, pause, cancel) using the Beethoven API.

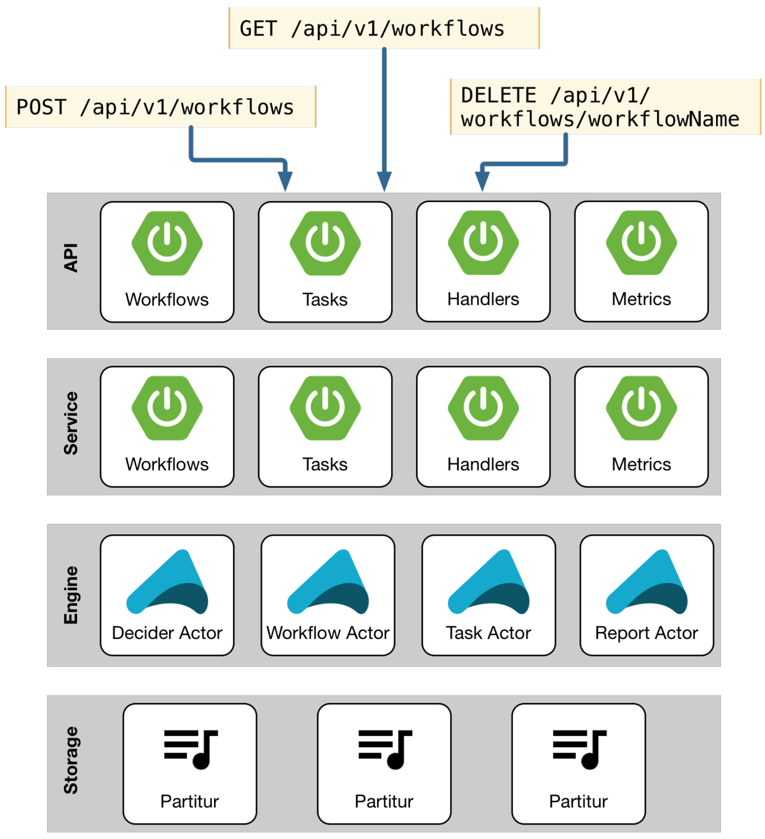
Figure 18 illustrates the Beethoven concrete architecture, which is decomposed into four layers. The top layer contains the software components responsible for providing RESTful APIs. The service layer contains the software components responsible for offering the services to the components in the upper layer. The engine layer comprises the actors responsible for executing microservice orchestrations that are defined in Partitur. Finally, the storage layer comprises the software components responsible for storing the specifications of the orchestration DSL.

The architectural requirements and the reference architecture represent a set of specifications for building applications based on an event-driven architecture to execute declarative business processes in parallel using non-blocking operations. Among these specifications, the following can be highlighted: (i) elements to produce, detect, consume, and react to events that may occur during a declarative business process execution; (ii) data consistency across multiple instances of declarative business processes without using distributed transactions; (iii) guarantee process compliance during an execution of business processes.

Based on the specifications of the reference architecture, it was decided to implement the engine layer of the concrete architecture using the actor model since this model satisfies the requirements through properties such as scalability, thread safety, encapsulation, fair scheduling, location transparency, and mobility (see Section 5.3 for more details). In addition, the actor model has a dynamic nature that allows the creation of new actors, modification of actor behavior at runtime and communication between actors through message exchanges. The following

[7] <http://www.oracle.com/technetwork/java/>
[8] <https://cloud.spring.io/spring-cloud-netflix/>
[9] <https://akka.io/>

Figure 18 – Concrete architecture



Source**:** Elaborated by the author.

Section describes each actor presented in the engine layer (e.g., WorkflowActor, TaskActor, DeciderActor, and ReportActor).

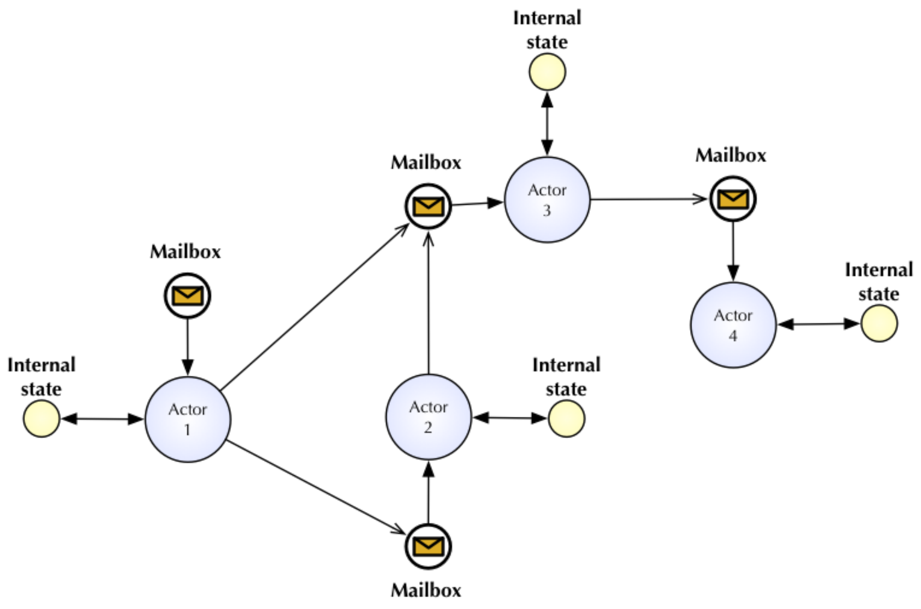
5.3 ACTOR MODEL

The actor model is a mathematical theory that treats the concept of actors as the universal primitives of digital computation (HEWITT; BISHOP; STEIGER, 1973). The model was originally proposed by Carl Hewitt in the 1970s as an object-oriented model to be used in the Artificial Intelligence area for safely exploiting concurrency in distributed systems (KOSTER; CUTSEM; MEUTER, 2016). In the actor model, applications are composed of autonomous entities denominated actors that communicate exclusively through asynchronous message passing. In addition, an actor has its own behavior and mutable internal state that cannot be shared directly with other actors.

An example of an actor system is represented by Figure 19. As displayed in the figure, actors are essentially independent concurrent processes that encapsulate their internal state and behavior, communicating exclusively by exchanging messages (KŘIKAVA; COLLET;

FRANCE, 2012). Messages are sent asynchronously and each actor maintains a mailbox (queue of received messages). Based on its designated behavior, each actor has the ability to respond to incoming messages by sending new messages, creating new actors, or defining a new behavior, which specifies how the message will process the next message. In other words, actors can be understood as reactive, dynamic, scalable, and distributed entities.

Figure 19 – Network of several actors



Source**:** Elaborated by the author.

Two important properties of the actor model are thread safety and scalability. Thread safety in actor model is provided by guaranteeing that message processing is executed by one thread at a time. In addition, access to an actor's mailbox is race conditions free. Therefore, unlike shared memory concurrency models, the actor model provides a thread safety framework for implementing concurrent-based applications. Scalability is accomplished by the ability to create new actors to perform specific tasks. This property allows applications based on this model to scale their logical processing units (actors) to satisfy higher demand. In addition, the actor model has a lower context-switching overhead over the standard shared-memory threads with locks (HALLER; ODERSKY, 2009). Furthermore, there are four important semantic properties of actor systems: encapsulation, fairness, location transparency and mobility (KARMANI; SHALI; AGHA, 2009). Each property is described below:

Encapsulation: The purpose of encapsulation is to provide a manner to organize programs into logical units for logically related software resources (SEBESTA, 2012). In the context of the actor model, there are two important requirements for encapsulation: state encapsulation and safe messaging.

Fair Scheduling: The notion of fairness in the actor model is based on a guarantee that a message is eventually delivered to its destination actor unless the destination actor is permanently unavailable.

Location Transparency: In the actor model, the actor name is agnostic about the actual location of an actor. In other words, although an actor system can be run on the same CPU or distributed on a network, the actor names must be unique in the context of their actor system. Therefore, location transparency provides an infrastructure for a software engineer to build actor-based applications without worrying about the actual physical actor locations.

Mobility: Mobility is defined as the ability of a computation to move across different computational resources. Mobility can be classified as either strong or weak (FUGGETTA; PICCO; VIGNA, 1998). Strong mobility is defined as the ability of a system to support migration of both code and execution state. Weak mobility, on the other hand, only allows migration of code across different computational resources.

Those aforementioned properties enable compositional design and simplify reasoning (AGHA *et al.*, 1997) and improve performance as applications and architectures scale (KIM; AGHA, 1995). Since actors do not use shared memory or any shared system resources to communicate and interact exclusively using asynchronous messages, the actor model prevents common concurrency issues such as low-level data races and deadlocks by design (KOSTER; CUTSEM; MEUTER, 2016).

5.4 ACTORS DESCRIPTIONS

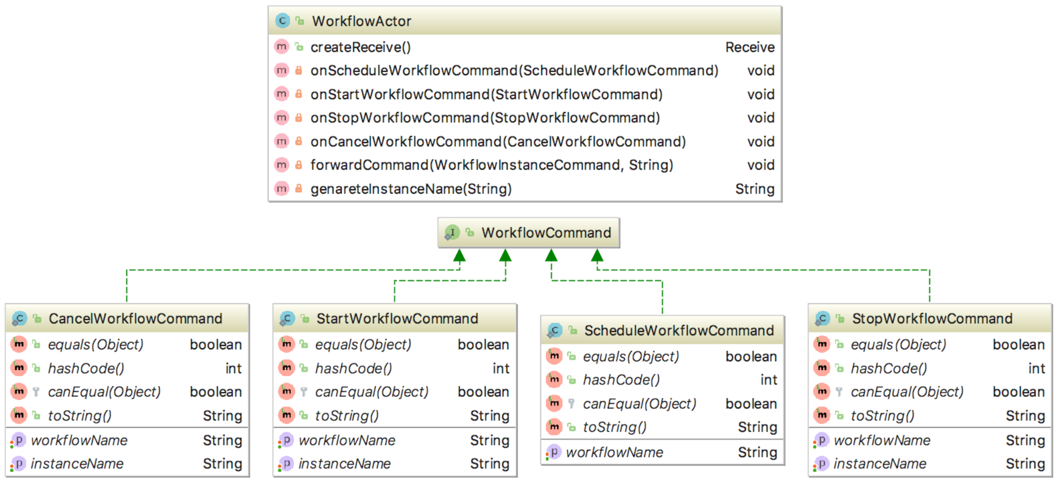
In the actor model, actors are essentially independent of concurrent processes that encapsulate their state and behavior and communicate exclusively by exchanging messages. To implement the orchestration engine of the reference architecture presented in Section 4.1.3.4, each event processor has been instantiated in the concrete architecture as an actor. Next, each actor implemented in the concrete architecture is detailed in terms of internal state, sent or received messages, behavior, and UML class diagram.

5.4.1 WorkflowActor

The *WorkflowActor* has been implemented following the Workflow Event Processor specification. For this reason, it is able to receive and process commands to manage workflow instances. In practice, the *WorkflowActor* receives commands and creates child actors to manage workflow instances. Specifically, the *WorkflowActor* works as a supervisory actor who receives commands to manage a workflow instance and delegates that responsibility to a *WorkflowInstanceActor* (*WorkflowActor*'s child). Each *WorkflowInstanceActor* receives commands from the *WorkflowActor* and updates the state of the workflow instance. As a result, all work that the *WorkflowActor* receives is delegated to a *WorkflowInstanceActor*. This approach provides a mechanism to isolate a workflow execution with no shared data among the concurrent actors.

The UML class diagram, illustrated in Figure 20, represents the *WorkflowActor* class and the commands to manage workflows. The *WorkflowActor* class is composed of methods that are responsible for processing the following command: *ScheduleWorkflowCommand*, *StartWorkflowCommand*, *StopWorkflowCommand*, and *CancelWorkflowCommand*. In order to send these commands to the *WorkflowActor*, it is required to specify their appropriate input. For instance, to schedule the execution of a specific workflow, the *ScheduleWorkflowCommand* should be used as command and a workflow name should be used as command input. The other commands required two inputs: a workflow name and a workflow instance name.

Figure 20 – UML class diagram of WorkflowActor



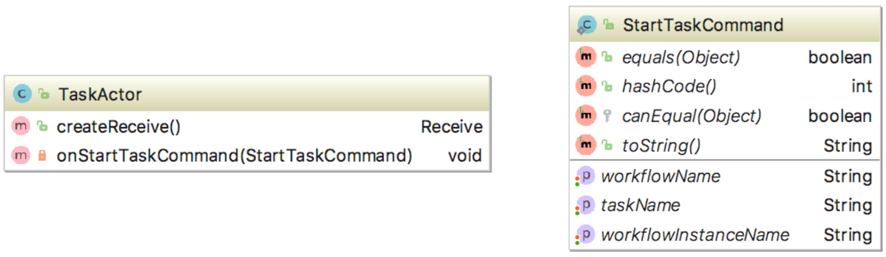
Source**:** Elaborated by the author.

5.4.2 TaskActor

The *TaskActor* receives a command to start tasks and sends success or failure events during the task execution. To execute a task, *TaskActor* uses the services provided by the *TaskService* to perform HTTP requests asynchronously. After receiving a command to start a task, the *TaskActor* performs an HTTP request and registers the request callbacks. For example, when executing an HTTP request, one callback is recorded for a successful response and another callback is recorded for failure response. Each callback triggers an event that is sent to other actors (*DeciderActor* and *ReportActor*). In this way, different tasks can be performed asynchronously in parallel with no external interference.

The UML class diagram, illustrated in Figure 21, represents the *TaskActor* class and the command to start tasks. Since a task is an atomic operation, there is no command to interrupt the task execution. In other words, after a task is started, it cannot be paused or canceled. In order to execute a task, the *StartTaskCommand* should be used with the following command inputs: task name, workflow name, and workflow instance name.

Figure 21 – UML class diagram of TaskActor



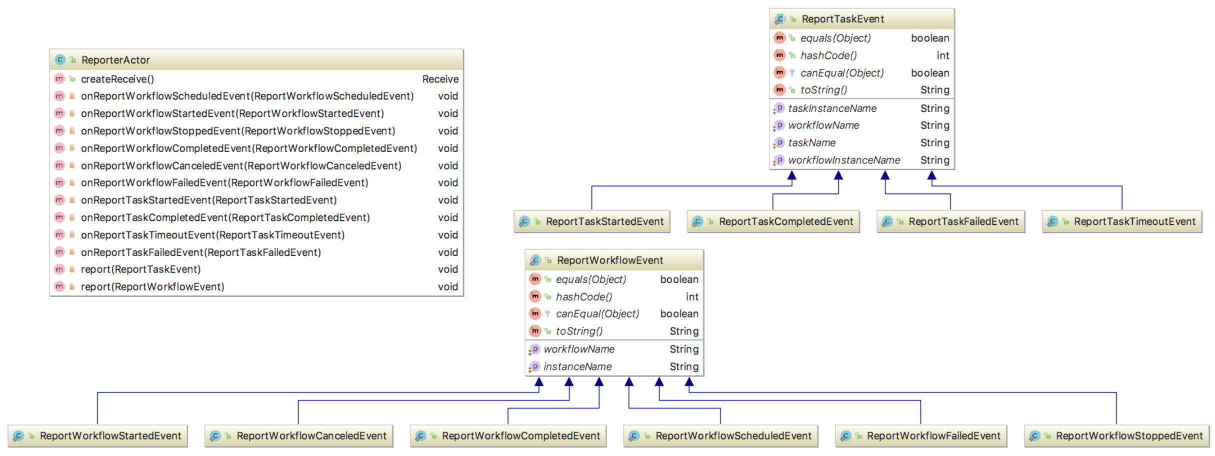
Source: Elaborated by the author.

5.4.3 ReportActor

The *ReportActor* is responsible for listening to all events triggered by other actors (*WorkflowInstanceActor* and *TaskActor*) during the execution of workflow instances. These events are used to record information about the execution of each task in a workflow. As an internal state, the *ReportActor* stores and maintains information about all workflow instances for further analysis of the application engineers. In addition, the information stored by the *ReportActor* can be consumed through the Spring Cloud Beethoven's API. For instance, it is possible to retrieve the elapsed time for a specific task or calculate the execution time of a workflow instance. It is also possible to retrieve all tasks that have failed in order to investigate the possible causes. Therefore, the *ReportActor* provides metrics and tracking mechanisms for software engineers in order to understand how distributed business processes are running in a microservices-based application.

The UML class diagram, illustrated in Figure 22, represents the *ReportActor* class and event that are captured during the execution of workflows and tasks. Since the *ReportActor* listeners to events that are triggered *WorkflowInstanceActor* and *TaskActor*, there are two event categories that may be processed: workflow and task. The workflow events are *ReportWorkflowScheduledEvent*, *ReportWorkflowStartedEvent*, *ReportWorkflowStoppedEvent*, *ReportWorkflowCompletedEvent*, *ReportWorkflowCanceledEvent*, and *ReportWorkflowFailedEvent*. The task events are *ReportTaskStartedEvent*, *ReportTaskTimeoutEvent*, *ReportTaskFailedEvent*, and *ReportTaskCompletedEvent*.

Figure 22 – UML class diagram of ReportActor



ReporterActor
createReceive() Receive
onReportWorkflowScheduledEvent(ReportWorkflowScheduledEvent) void
onReportWorkflowStartedEvent(ReportWorkflowStartedEvent) void
onReportWorkflowStoppedEvent(ReportWorkflowStoppedEvent) void
onReportWorkflowCompletedEvent(ReportWorkflowCompletedEvent) void
onReportWorkflowCanceledEvent(ReportWorkflowCanceledEvent) void
onReportWorkflowFailedEvent(ReportWorkflowFailedEvent) void
onReportTaskStartedEvent(ReportTaskStartedEvent) void
onReportTaskCompletedEvent(ReportTaskCompletedEvent) void
onReportTaskTimeoutEvent(ReportTaskTimeoutEvent) void
onReportTaskFailedEvent(ReportTaskFailedEvent) void
report(ReportTaskEvent) void
report(ReportWorkflowEvent) void
ReportTaskEvent
equals(Object) boolean
hashCode() int
canEqual(Object) boolean
toString() String
taskInstanceName String
workflowName String
taskName String
workflowInstanceName String
ReportTaskStartedEvent
ReportTaskCompletedEvent
ReportTaskFailedEvent
ReportTaskTimeoutEvent
ReportWorkflowEvent
equals(Object) boolean
hashCode() int
canEqual(Object) boolean
toString() String
workflowName String
instanceName String
ReportWorkflowStartedEvent
ReportWorkflowCanceledEvent
ReportWorkflowCompletedEvent
ReportWorkflowScheduledEvent
ReportWorkflowFailedEvent
ReportWorkflowStoppedEvent

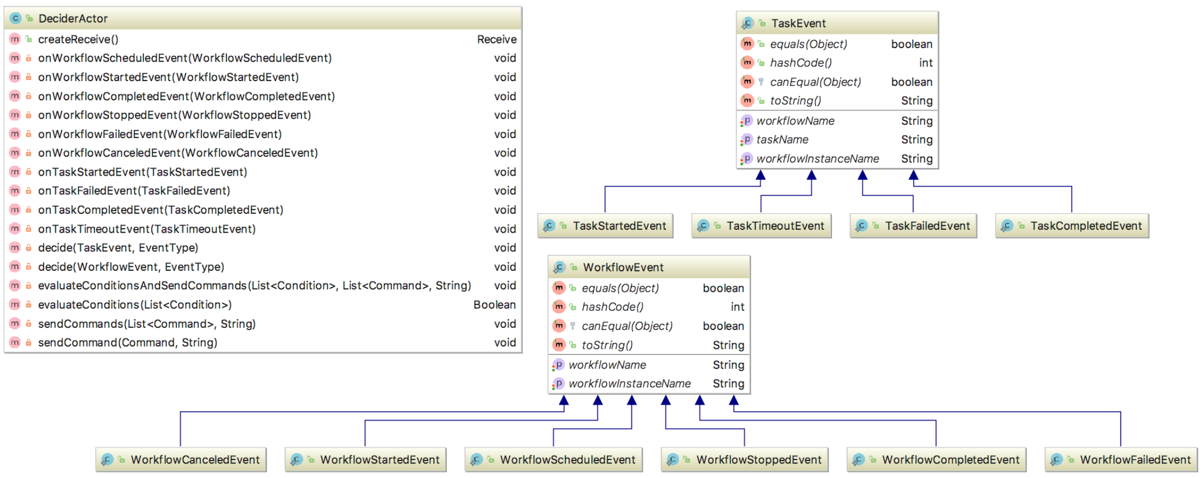
Source**:** Elaborated by the author.

5.4.4 DeciderActor

The *DeciderActor* is responsible for determining which action should be performed during executions of workflow instances. To this end, the *DeciderActor* receives events triggered by the actors *WorkflowInstanceActor* and *TaskActor* and decides, based on the event handlers, which command (e.g., start a workflow or a task) should be sent and to whom. The *DeciderActor* does not store information in its internal state, and its behavior has a reactive nature.

The UML class diagram, illustrated in Figure 23, represents the *DeciderActor* class and event that are captured during the execution of workflows and tasks. Similar to *ReportActor*, the *DeciderActor* listeners to all events that are triggered *WorkflowInstanceActor* and *TaskActor*. However, only the events that satisfy the conditions specified in the event handlers are processed. In addition, as in the *ReportActor*, there are two event categories that may be processed: workflow and task. In the workflow events, there are the following events: *WorkflowScheduledEvent*, *WorkflowStartedEvent*, *WorkflowCompletedEvent*, *WorkflowStoppedEvent*, *WorkflowFailedEvent*, and *WorkflowCanceledEvent*. In the task events, there are the following events: *TaskStartedEvent*, *TaskFailedEvent*, *TaskCompletedEvent*, and *TaskTimeoutEvent*.

Figure 23 – UML class diagram of DeciderActor


DeciderActor
createReceive() Receive
onWorkflowScheduledEvent(WorkflowScheduledEvent) void
onWorkflowStartedEvent(WorkflowStartedEvent) void
onWorkflowCompletedEvent(WorkflowCompletedEvent) void
onWorkflowStoppedEvent(WorkflowStoppedEvent) void
onWorkflowFailedEvent(WorkflowFailedEvent) void
onWorkflowCanceledEvent(WorkflowCanceledEvent) void
onTaskStartedEvent(TaskStartedEvent) void
onTaskFailedEvent(TaskFailedEvent) void
onTaskCompletedEvent(TaskCompletedEvent) void
onTaskTimeoutEvent(TaskTimeoutEvent) void
decide(TaskEvent, EventType) void
decide(WorkflowEvent, EventType) void
evaluateConditionsAndSendCommands(List<Condition>, List<Command>, String) void
evaluateConditions(List<Condition>) Boolean
sendCommands(List<Command>, String) void
sendCommand(Command, String) void
TaskEvent
equals(Object) boolean
hashCode() int
canEqual(Object) boolean
toString() String
workflowName String
taskName String
workflowInstanceName String
TaskStartedEvent
TaskTimeoutEvent
TaskFailedEvent
TaskCompletedEvent
WorkflowEvent
equals(Object) boolean
hashCode() int
canEqual(Object) boolean
toString() String
workflowName String
workflowInstanceName String
WorkflowCanceledEvent
WorkflowStartedEvent
WorkflowScheduledEvent
WorkflowStoppedEvent
WorkflowCompletedEvent
WorkflowFailedEvent

Source: Elaborated by the author.

5.5 STEP BY STEP

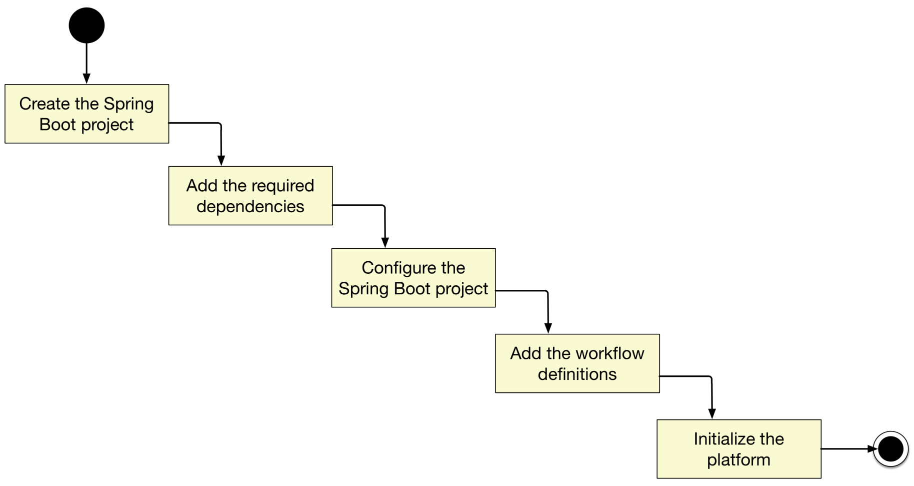
The Spring Cloud Beethoven platform has been designed to run as an infrastructure microservice using a minimal setup. Therefore, there are few steps (illustrated in Figure 24) required to add the proposed platform to a microservices-based application. To this end, a software engineer should follow these steps: (i) create Spring Boot project using a build automation (e.g., Maven or Gradle); (ii) add the required dependencies; (iii) configure the Spring Boot project; (iv) add the workflow definitions; and (v) initialize the Spring Cloud Beethoven platform. As follows, each step is described in details.

5.5.1 Step S-1

The first step consists of creating a Spring Boot project that is used as basis for the Spring Cloud Beethoven platform. For this purpose, Spring Initializr[10] may be used to generate a Spring Boot project with its dependencies. In Spring Initializr, illustrated in Figure 25, a developer needs to specify the build automation tool of project (Maven or Gradle), the programming language (e.g., Java, Kotlin, or Groovy), the version of Spring Boot, the project metadata (group ID and artifact ID), and a optional set of project dependencies.

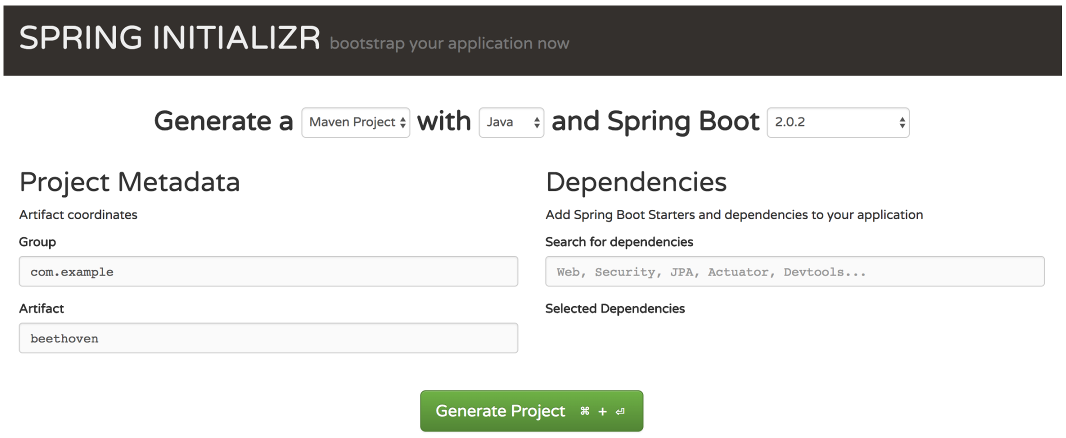
[10] <https://start.spring.io/>

Figure 24 – Step by step guide


Create the Spring Boot project
Add the required dependencies
Configure the Spring Boot project
Add the workflow definitions
Initialize the platform

Source**:** Elaborated by the author.

Figure 25 – Spring Initializr main screen



SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 2.0.2

Project Metadata

Artifact coordinates

Group

com.example

Artifact

beethoven

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Source**:** Elaborated by the author.

5.5.2 Step S-2

After creating a Spring Boot project, the software engineer should add the following dependencies: (i) Spring Cloud Netflix Eureka; and (ii) Spring Cloud Beethoven. Listing 12 and Listing 13 demonstrate how to add the required dependencies in a Maven[11] pom file or a Gradle[12] build file respectively.

[11] <https://maven.apache.org/>
[12] <https://gradle.org/>

Code 12 – Maven dependencies

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>io.beethoven</groupId>
    <artifactId>beethoven-starter</artifactId>
    <version>0.0.1</version>
</dependency>
```

Code 13 – Gradle dependencies

```
compile("org.springframework.cloud:spring-cloud-starter-netflix-eureka-client")
compile("io.beethoven:beethoven-starter:0.0.1")
```

5.5.3 Step S-3

In order to configure the Spring Boot project, the software engineer needs to configure Eureka by using property configuration files. By default, Spring Boot searches for the property file *application.properties* in the classpath. The minimal configuration file, illustrated in Listing 21, is composed of the application name, the server HTTP port, the name of the application to be registered with Eureka, and the Eureka service URL.

Code 14 – Minimal configuration

```
# Application configuration
spring.application.name=beethoven-service
server.port=9090

# Eureka configuration
eureka.instance.appname=beethoven-service
eureka.client.service-url.default-zone=http://localhost:8761/eureka/
```

5.5.4 Step S-4

The workflows defined in Partitur must be located in a folder named workflows in the *classpath*. Thus, Spring Cloud Beethoven searches for the workflows definitions and load them in order to create workflow instances when needed.

5.5.5 Step S-5

In order to initialize the Spring Cloud Beethoven platform, the software engineer must create a class and add following Java annotations: *@EnableBeethoven* and *@SpringBootApplication*. The first annotation enables the Beethoven platform to search for workflow specifications and the second annotation enables autoconfiguration in a Spring-based application. The created class needs to implement the main method that runs a Spring Boot application and initialize the Beethoven platform using the initialize method as shown in Listing 15.

Code 15 – Spring Cloud Beethoven initialization

```java
@EnableBeethoven
@SpringBootApplication
public class BeethovenApplication {
    public static void main(String... args) throws Exception {
        ApplicationContext context = SpringApplication.run(BeethovenApplication.
    ↪ class, args);
        ActorSystem actorSystem = context.getBean(ActorSystem.class);
        Beethoven.initialize(actorSystem);
    }
}
```

6 EVALUATION

This chapter presents the evaluations that have been conducted in order to provide empirical evidence for analyzing the benefits and trade-offs provided by the Beethoven platform. In order to achieve this particular aim, two example applications have been developed and one controlled experiment has been conducted. In Section 6.1, the first example application, which is an implementation of an MSA-based application, is presented. In Section 6.2, the second example application, which is an orchestrated version of reference application for microservices-based applications, is exhibited. Finally, Section 6.3 presents the controlled experiment that has been conducted in order to evaluate the possible overhead produced by the proposed platform after the modification of the reference application.

6.1 EXAMPLE APPLICATION 1

In order to demonstrate the applicability and feasibility of the Beethoven platform, an example application has been developed. An example application is a validation method used to describe and validate a new type of tool, framework, methodology, model, or process in software engineering that may be applied to evaluate phenomena, methods, techniques (SHAW, 2003). The developed example application is a microservices-based application (available on GitHub[1]) that implements a Customer Relationship Management (CRM) system for an investment bank. The example application is composed of the following microservices:

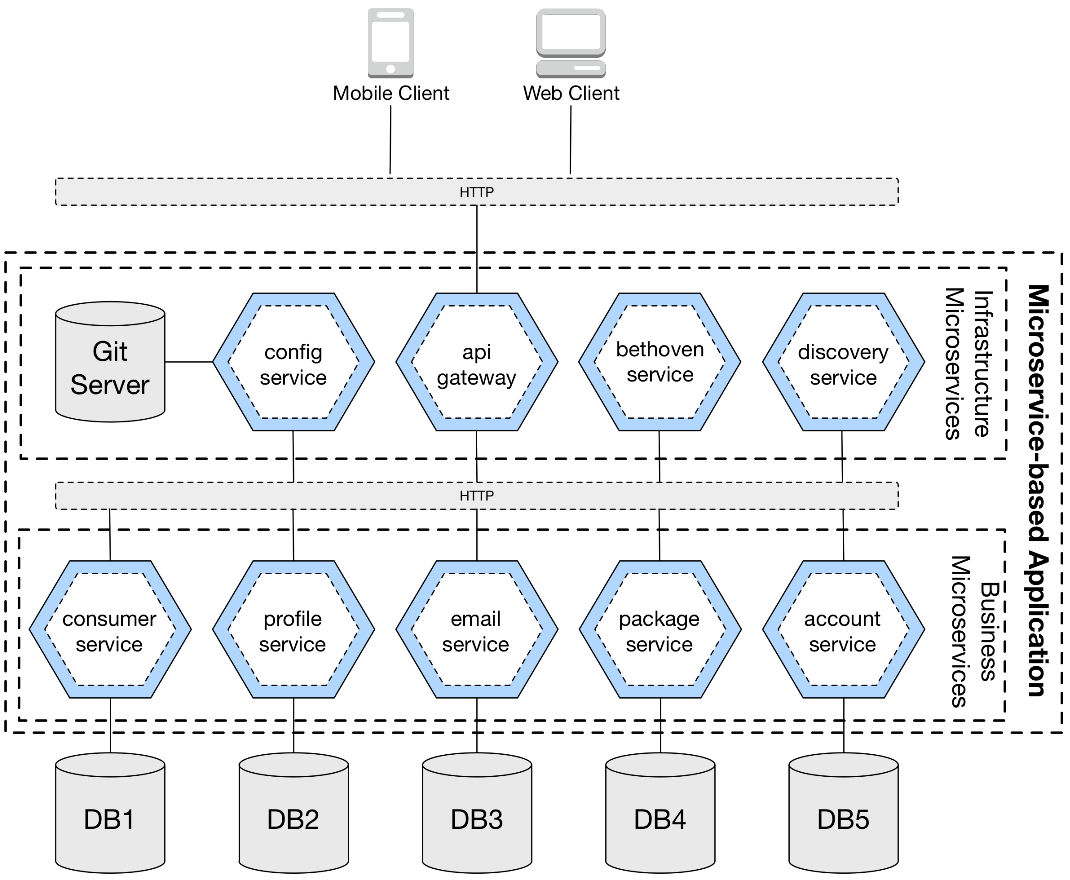
costumer-service: manages the costumer registry;
profile-service: analyzes costumers' history to define retention strategies;
email-service: sends welcome, promotion, and informational emails;
package-service: sends personalized packs;
account-service: sends credit/debit card, and letter within card password;
discovery-service: maintains a registry of service information;
api-gateway: servers as entry point for different clients;
config-service: manages and distributes the configuration for each microservice;
beethoven-service: manages and performs Partitur workflows.

The graphical representation of the example application architecture is illustrated in Figure 26. As shown in Section 2.2, a microservices-based application is comprised of infrastructure and business services. In the infrastructure layer, the example application consists

[1] <https://github.com/davimonteiro/crm-msa-example>

of the following microservices: *discovery-service*, *api-gateway*, *config-service*, and *beethoven-service*. In the business layer, the example application is formed of the following microservices: *costumer-service*, *profile-service*, *email-service*, *package-service*, and *ccount-service*. Each business microservice possesses its own database and communicates exclusively through HTTP requests.

Figure 26 – CRM application's architecture



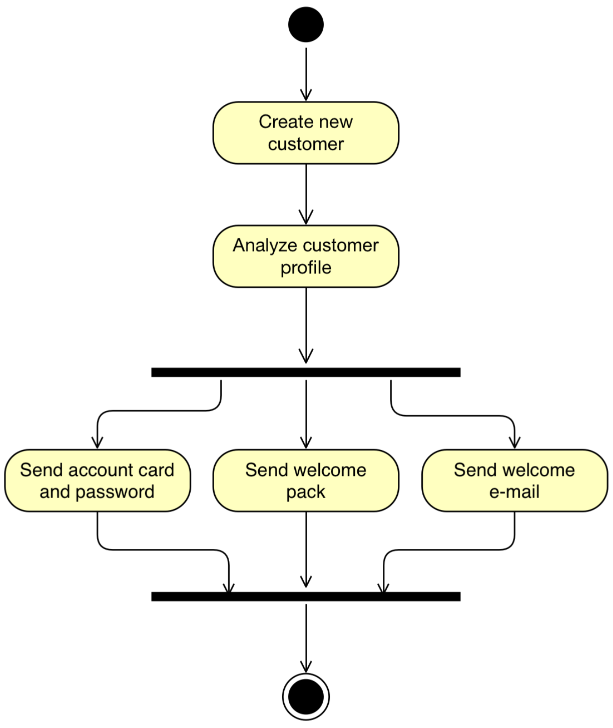
Source**:** Elaborated by the author.

6.1.1 New customer process

In the CRM application, each microservice provides a business capability. For instance, *profile-service* is responsible for investigating the costumers' history in order to define retention strategies. However, *profile-service* does not provide any additional functionality in order to perform the operations that are required to instantiate the retention strategies. These operations are performed by other microservices. For instance, in order to send a promotional e-mail, which is a type of retention strategies, *email-service* should be invoked.

In this context, there is a distributed business process for new customers that is started after registering a new customer using the *costumer-service*. Next, the *profile-service* analyzes the new customer profile to define which retention strategies should be used. In this process, new customers should receive a welcome email (*email-service*), a personalized package (*account-service*), and a letter that contains the card's password and the account's card (*account-service*). Each business process step is performed by a specific microservice. The graphical representation of the new customer process is depicted in Figure 27 using the UML activity diagram notation.

Figure 27 – New customer process



Source: Elaborated by the author.

In order to model the new customer process as a Partitur workflow specification, the methodology described in Section 5.5 has been used. As result, Code 16 has been produced and registered in *beethoven-service*. Code 16 depicts a Partitur specification for the new customer process. Line 1 declares a Partitur workflow definition using the keyword `workflow` and *newCustomerProcess* as workflow name. Lines 2-26 define a set of tasks for creating a new customer, analyzing the customer profile, sending the account card and password, sending a welcome package, and sending a welcome e-mail. In other words, each declared task is responsible for performing one step defined in the new customer process shown in Figure 27. For instance, the step for creating a new customer is executed by the task named `newCustomerProcess`.

Lines 27-43 define three Partitur event handlers. The first one defines when the workflow named *newCustomerProcess* is scheduled, then the task named *createNewCustomer* must be started. The second one defines when the task named *createNewCustomer* is finished, then the task named *analyzeCustomerProfile* must be started. Finally, the third one defines when the task named *analyzeCustomerProfile* is terminated, then the following tasks should be started: *sendWelcomeEmail*, *sendWelcomePackage*, *sendAccountCardAndPassword*.

Code 16 – Partitur specification for the new customer process

```
workflow newCustomerProcess {
    task createNewCustomer {
        post("http://customer-service/customers")
            .header("Content-Type", "application/json")
            .body("${createNewCustomer.input}")
    }
    task analyzeCustomerProfile {
        post("http://profile-service/profiles/analyze")
            .header("Content-Type", "application/json")
            .body("${createNewCustomer.response}")
    }
    task sendWelcomeEmail {
        post("http://email-service/welcome")
            .header("Content-Type", "application/json")
            .body("${analyzeCustomerProfile.response}")
    }
    task sendWelcomePackage {
        post("http://package-service/welcome")
            .header("Content-Type", "application/json")
            .body("${analyzeCustomerProfile.response}")
    }
    task sendAccountCardAndPassword {
        post("http://account-service/cards-passwords")
            .header("Content-Type", "application/json")
            .body("${analyzeCustomerProfile.response}")
    }
    handler h1 {
        on WORKFLOW_SCHEDULED
        when workflowNameEqualsTo("newCustomerProcess")
        then startTask("createNewCustomer")
    }
    handler h2 {
        on TASK_COMPLETED
        when taskNameEqualsTo("createNewCustomer")
        then startTask("analyzeCustomerProfile")
    }
    handler h3 {
        on TASK_COMPLETED
        when taskNameEqualsTo("analyzeCustomerProfile")
        then startTask("sendWelcomeEmail"),
             startTask("sendWelcomePackage"),
             startTask("sendAccountCardAndPassword")
    }
}
```

6.1.2 New versions of the new customer process

In order to illustrate how to apply modifications in the initial steps of the new customer process, consider the following scenario: the regulatory agencies from the federal government have imposed new requirements for the new customers in investment banks. These requirements are related to the acquisition of information about the annual income of customers that should be forwarded to the regulatory agencies using their web services. In order to implement these requirements, a new step must be added to the new customer process for sending customer information to the regulatory agencies. This modification can be done by adding a new task in the Partitur specification (off-line modifications) or by modifying the business process at runtime using the Beethoven's API (online modifications).

In order to apply off-line modification on the new costumer process, the following activities must be performed: (i) interrupt *beethoven-service*; (ii) modify the Partitur specification of the new costumer process by adding a new task to send the income information of the new customers and adding an event handler to start the new task; and (iii) restart *beethoven-service* in order to utilize the new Partitur specification of the new costumer process for creating new workflow instances. After the modification has been applied and *beethoven-service* has been restarted, new workflow instances will use the new specification during their execution.

In order to apply online modification on the new customer process, it is possible to use the Beethoven's API for adding a new task to send the income information of the new customers and update an existing event handler to start the new task. To this end, the HTTP request presented in Code 17 is used to add a new task named *sendCostumerIncomeData* to the new customer process and the HTTP request presented in Code 18 is used to update the event handler *h3* for starting *sendCostumerIncomeData* after *analyzeCustomerProfile* has been completed. Thus, by using the Beethoven's API, there is no need to interrupt *beethoven-service* in order to apply modifications in the Partitur specification of the new customer process. After the modification has been applied, new workflow instances will use the new specification during their execution.

Code 17 – Adding a new task

```
POST /api/workflows/newCustomerProcess/tasks HTTP/1.1
Content-Type: application/json

{"name":"sendCostumerIncomeData",
    "httpRequest":{
        "url":"http://income-tax.gov/new-custumers",
        "method":"POST",
        "headers":[{"name":"Content-Type","value":"application/json"}]
    }
}
```

Code 18 – Updating an event handler

```
PUT /api/workflows/newCustomerProcess/handlers/h3 HTTP/1.1
Content-Type: application/json

{
    "name":"h3",
    "eventType":"TASK_COMPLETED",
    "conditions":[{"conditionFunction":{"taskName":"analyzeCustomerProfile"}}],
    "commands":[
        {"operation":"START_TASK","taskName":"sendWelcomeEmail"},
        {"operation":"START_TASK","taskName":"sendWelcomePackage"},
        {"operation":"START_TASK","taskName":"sendAccountCardAndPassword"},
        {"operation":"START_TASK","taskName":"sendCostumerIncomeData"}
    ]
}
```

6.2 EXAMPLE APPLICATION 2

In order to reduce the research bias during evaluation of the proposed platform, a second example application has been developed. However, instead of implementing a new microservices-based application to demonstrate the feasibility of the Beethoven platform, an existing microservices-based application has been adapted in order to use the Beethoven platform for performing microservice composition in distributed business processes. The second example application is based on a reference application (available on GitHub[2]) for an online web store that has been developed to demonstrate the best practices (see Section 2.2.2) for building microservices using Spring Boot[3] and Spring Cloud[4]. The reference application is composed of the following microservices:

account-service: responsible for managing the account information of users;

[2] <https://github.com/kbastani/spring-cloud-event-sourcing-example>
[3] <https://projects.spring.io/spring-boot>
[4] <https://cloud.spring.io>

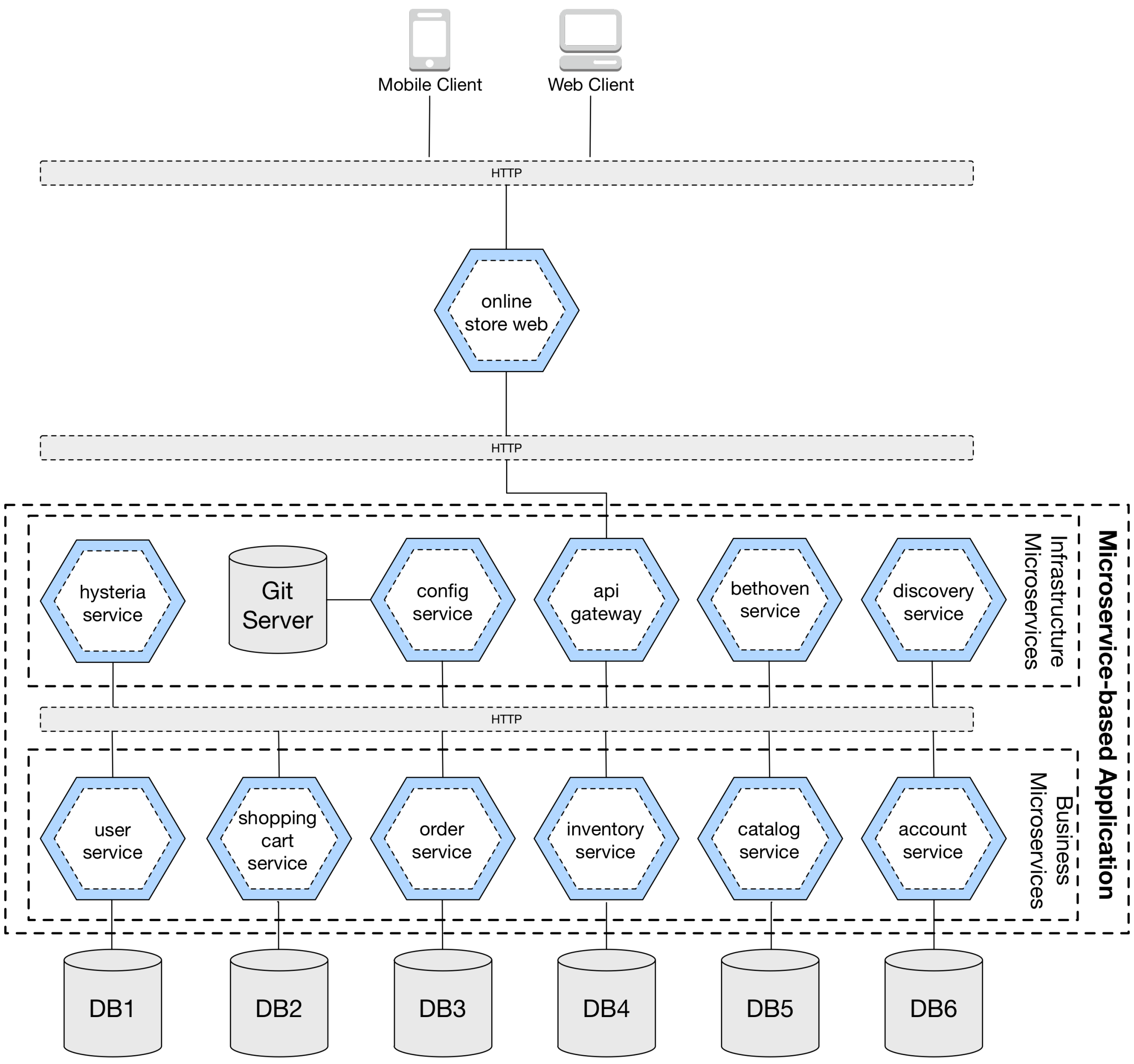
catalog-service : responsible for retrieving the active catalog of products for the online store;
inventory-service: responsible for managing the inventory and product catalogs ordered;
online-store-web: responsible for serving as the main user interface of the online store;
order-service: responsible for providing an API to facilitate the ordering of products;
shopping-cart-service: responsible for providing an API that manages the products that a user has chosen to add to their online shopping cart;
user-service: responsible for providing the authentication gateway for the application;
discovery-service: responsible for maintaining a registry of service information;
hystrix-service: responsible for providing a dashboard used to monitor circuit breakers;
api-gateway: responsible for securely exposing routes from microservices to consumers;
config-service: responsible for distributing external configurations for each microservice;
beethoven-service: responsible for managing and performing Partitur workflows.

The reference application architecture, graphically represented in Figure 28, is composed of sets of infrastructure and business microservices. In the infrastructure layer, the reference application is composed of the following microservices: *discovery-service*, *hystrix-service*, *api-gateway*, *config-service*, and *beethoven-service*. Each infrastructure microservice provides a support functionary for the reference application (e.g., service discovery, externalized configuration, or service composition). In the business layer, the reference application is formed of the following microservices: *account-service*, *catalog-service*, *inventory-service*, *order-service*, *shopping-cart-service*, and *user-service*. Each business microservice provides a business capability and accesses its own database. In addition to infrastructure and business microservices, there is one microservice named *online-store-web* responsible for providing the user interface as a web application for the online shopping store.

6.2.1 Checkout process

In the reference application, each microservice is responsible for running one single business capability. However, *shopping-cart-service* accumulates two responsibilities: managing the shopping cart and running the checkout process. The checkout process requires the collaboration of different microservices in order to be performed. To this end, the checkout business process, shown in Figure 29, is composed of the following activities: (i) collect the items that have been added from the shopping cart; (ii) checks the availability of the selected item; (iii) create a new order if the selected items are available; (iv) if the order is successfully created,

Figure 28 – Reference application's architecture

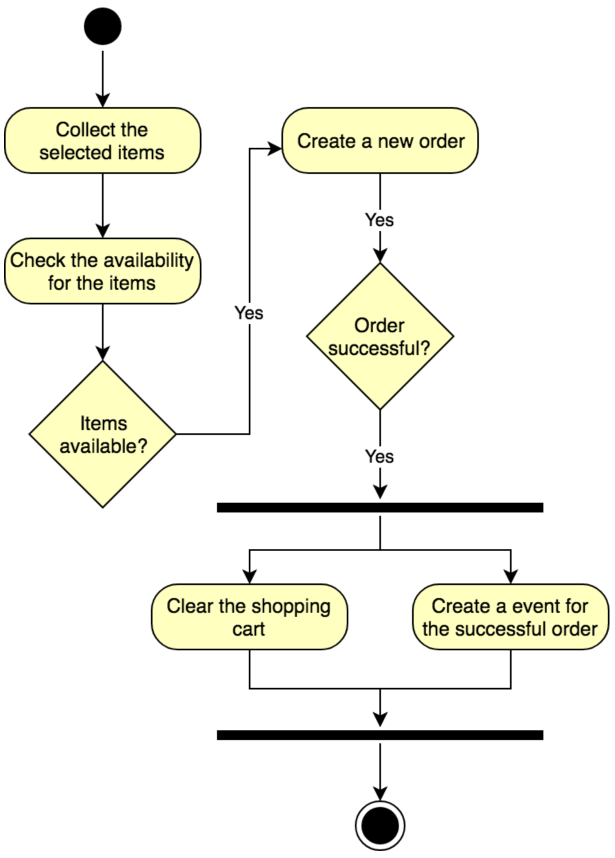


Source**:** Elaborated by the author.

then clear the shopping cart and create an event that represents the successful order. Since each activity is performed by a particular microservice (e.g., *inventory-service*, *catalog-service*, or *order-service*), thereby, there is a necessity to coordinate microservices in a given order to perform the activities of the checkout process.

In the original version of the reference application, there is a method named *checkout* that is responsible for implementing the checkout process. In the *checkout* method, illustrated in Code 19, each activity from the checkout process (see Figure 29) is imperatively implemented using the general purpose Java programming language. In order to simplify, some details of the *checkout* method have been omitted. The completed version of the *checkout* method is available on GitHub. In the *checkout* method, Lines 2-8 are responsible for collecting the items that have

Figure 29 – The checkout process


Collect the selected items
Check the availability for the items
Items available?
Yes
Create a new order
Yes
Order successful?
Yes
Clear the shopping cart
Create a event for the successful order

Source**:** Elaborated by the author.

been added by the user (first activity). Next, if the shopping cart is not empty, then the availability of the selected items should be verified (second activity). After that, Lines 25-33 are responsible for creating a new order (third activity). If the order is successfully created, then two activities must be performed: clear the shopping cart (forth activity) and create an event for the successful order (fifth activity). In the following subsection, the possible concerns induced by this approach are highlighted.

Code 19 – The checkout method

```java
public CheckoutResult checkout() throws Exception {
    CheckoutResult checkoutResult = new CheckoutResult();
    ShoppingCart currentCart = null;
    try {
        currentCart = getShoppingCart();
    } catch (Exception e) {
        log.error("Could not retrieve shopping cart", e);
    }
    if (currentCart != null) {
        Inventory[] inventory =
                oAuth2RestTemplate.getForObject(
                    String.format(
                        "http://inventory-service/v1/inventory?productIds=%s",
                        currentCart.getLineItems()
                        .stream()
                        .map(LineItem::getProductId)
                        .collect(Collectors.joining(","))), Inventory[].class);
        if (inventory != null) {
            Map<String, Long> inventoryItems = Arrays.asList(inventory)
                    .stream()
                    .map(inv -> inv.getProduct().getProductId())
                    .collect(groupingBy(Function.identity(), counting()));
            if (checkAvailableInventory(
                    checkoutResult, currentCart, inventoryItems)) {
                Order orderResponse = oAuth2RestTemplate.postForObject(
                    "http://order-service/v1/orders",
                        currentCart.getLineItems().stream()
                            .map(prd ->
                                new demo.order.LineItem(prd.getProduct().getName(),
                                        prd.getProductId(), prd.getQuantity(),
                                        prd.getProduct().getUnitPrice(), TAX))
                                .collect(Collectors.toList()),
                        Order.class);

                if (orderResponse != null) {
                    checkoutResult.setResultMessage("Order created");
                    oAuth2RestTemplate.postForEntity(
                        String.format("http://order-service/v1/orders/%s/events",
                            orderResponse.getId()),
                            new OrderEvent(OrderEvent.OrderEventType.CREATED,
                                orderResponse.getId()), ResponseEntity.class);
                    checkoutResult.setOrder(orderResponse);
                }

                User user = oAuth2RestTemplate.getForObject(
                    "http://user-service/uaa/v1/me", User.class);
                addCartEvent(new CartEvent(
                    CartEvent.CartEventType.CHECKOUT, user.getId()), user);
            }
        }
    }
    return checkoutResult;
}
```

6.2.2 Issues with the original version

Gradually, the checkout process may evolve and it will be necessary to modify its initial activities. For instance, the federal, state, and local governments may impose new taxes for every sale at any online store. Therefore, in order to implement that new requirement, it will be necessary to modify the checkout process that is executed by the *shopping-cart-service*. Furthermore, new software requirements may arise from other sources (e.g., internal process, new marketing strategies, and so on). Consequently, after modifying the checkout process, it will be necessary a downtime of the online store application in order to update *shopping-cart-service*.

Although the reference application utilizes a set of best practices for building microservices-based applications, it infringes one of the main principles proposed by the microservices architectural style: componentization via services. In the microservices architectural style, componentization is performed at the level of services that must have only a single responsibility. However, to execute the checkout process, the reference application utilizes one of the SOA concepts: composed services. Composed services are services that access and depended on multiple services to provide business functionalities by executing workflows.

A software architectural style defines a set of principles that are used as constraints on a software architecture (PERRY; WOLF, 1992; GARLAN; SHAW, 1994). Since the microservices architecture is a software architectural style, its principles, characteristics, and constraints must be obeyed in order to develop microservices-based applications. Otherwise, the benefits proposed by this architectural style will not be experienced.

As a consequence of the adoption of the concept of composed services for building *shopping-cart-service*, the reference application centralizes the entire business process logic into a single microservice causing the following drawbacks: (i) difficulty in maintaining the business process since a composed microservice is depended on multiple microservices in order to execute business processes; (ii) high coupling between the microservice that executes the business process and the other microservices responsible for performing each activity of the business process; (iii) system downtime to apply changes in the particular microservice that is responsible for executing the business process; (iv) very elaborated business process for simple steps since it is required to handle possible exceptions that may occur during the execution of a business process.

6.2.3 Beethoven version

In order to compose the microservices responsible for executing the checkout process using the Beethoven platform, a new version of this process has been created. To this end, the following activities have been carried out: (i) creating new endpoints for each microservice responsible for performing an activity of the checkout business process; (ii) specifying the checkout process in the Partitur language; (iii) configuring the Beethoven platform in the reference architecture; (iv) performing HTTP requests to the Beethoven's API in order to manage the execution of the checkout process. Each activity is explained in detail as follows.

(i) Creating new endpoints — In order to create a new version of the checkout business process, new endpoints have been created for each microservice that performs a step in the checkout business process. For example, in order to retrieve the items that have been added to the shopping cart in the original version of the reference application, the following endpoint should be used "http://shopping-cart-service/v1/cart"; in contrast, to use the version orchestrated by Beethoven, the following endpoint has been created "http://shopping-cart-service/v1/cart/orchestrated". In this manner, two functional versions of the checkout business process have been maintained: one version that applies the concept of SOA named composed service and another version that utilizes the Beethoven platform in order to execute the checkout process.

(ii) Specifying the business process — In order to specify the checkout process in the Partitur language, the Code 20 has been created. In the first line, the workflow named `checkoutProcess` is declared. Next, thought lines 2-27, five tasks have been created using a contextual input in the header definition for a particular OAuth access token authorization. The fist task (lines 2-5) named `getShoppingCart` is responsible for retrieving shopping cart items. The second task (lines 6-11) named `checkAvailableInventory` is responsible for checking availability of selected items. The `checkAvailableInventory` task receives a contextual input (line 10) from the response of `getShoppingCart`. The third task (lines 12-17) named `createNewOrder` is responsible for creating a new order, receiving a contextual input from the response of `checkAvailableInventory`. The forth task (lines 18-23) named `createSuccessfulOrderEvent` is responsible for creating an event that represents a successful order. The forth task receives a contextual input (line 22) from the response of `createNewOrder`. The last task (lines 24-27) named `clearShoppingCart` is responsible for cleaning the shopping cart.

Code 20 – Partitur specification for the checkout process

```
 1 workflow checkoutProcess {
 2     task getShoppingCart {
 3         get("http://shopping-cart-service/v1/cart/orchestrated")
 4             .header("Authorization", "${access_token}")
 5     }
 6     task checkAvailableInventory {
 7         post("http://inventory-service/v1/inventory/checkavailable/orchestrated")
 8             .header("Content-Type", "application/json")
 9             .header("Authorization", "${access_token}")
10             .body("${getShoppingCart.response}")
11     }
12     task createNewOrder {
13         post("http://order-service/v1/orders/orchestrated")
14             .header("Content-Type", "application/json")
15             .header("Authorization", "${access_token}")
16             .body("${checkAvailableInventory.response}")
17     }
18     task createSuccessfulOrderEvent {
19         post("http://order-service/v1/orders/events/orchestrated")
20             .header("Content-Type", "application/json")
21             .header("Authorization", "${access_token}")
22             .body("${createNewOrder.response}")
23     }
24     task clearShoppingCart {
25         post("http://shopping-cart-service/v1/cart/clear/orchestrated")
26             .header("Authorization", "${access_token}")
27     }
28     handler h1 {
29         on WORKFLOW_SCHEDULED
30         when workflowNameEqualsTo("checkoutProcess")
31         then startTask("getShoppingCart")
32     }
33     handler h2 {
34         on TASK_COMPLETED
35         when taskNameEqualsTo("getShoppingCart")
36         then startTask("checkAvailableInventory")
37     }
38     handler h3 {
39         on TASK_COMPLETED
40         when taskNameEqualsTo("checkAvailableInventory")
41         then startTask("createNewOrder")
42     }
43     handler h4 {
44         on TASK_COMPLETED
45         when taskNameEqualsTo("createNewOrder")
46         then startTask("clearShoppingCart"),
47              startTask("createSuccessfulOrderEvent")
48     }
49 }
```

After defining a set of tasks, four event handlers have been specified thought lines 28-48. The first event handler (lines 28-32) named *h1* is composed of the following structure: when *WORKFLOW_SCHEDULED* occurs, if workflow name is equal to *checkoutProcess*, then start the *getShoppingCart* task. The second event handler (lines 33-37) named *h2* is composed of the following structure: when *TASK_COMPLETED* occurs, if task name is equal to *getShopping-Cart*, then start the *checkAvailableInventory* task. The third event handler (lines 38-42) named *h3* is composed of the following structure: when *TASK_COMPLETED* occurs, if task name is equal to *checkAvailableInventory*, then start the *createNewOrder* task. Finally, the fourth event handler (lines 43-48) named *h4* is composed of the following structure: when *TASK_COMPLETED* occurs, if task name is equal to *createNewOrder*, then start the *clearShoppingCart* and *create-SuccessfulOrderEvent* tasks.

(iii) Configuring the Beethoven platform — The configuration file used the YAML extension and is presented in Code 21. In lines 1-3, the service name is defined in order to allow the microservices of the reference application to access the Beethoven platform by its service name. For instance, considering only the microservices that are part of the reference application, it is possible to use the following URL to access the Beethoven endpoints: *http://beethoven-service/*. Then, in lines 4-15, the OAuth2 configuration is implemented as specified by the OAuth2 server that has the following endpoint: *http://auth-service/*. Finally, in lines 16-23, the discovery service configuration is presented. Specifically, the reference application relies on Eureka[5] for locating services.

Code 21 – Beethoven configuration

```
1  spring:
2    application:
3      name: beethoven-service
4  security:
5    oauth2:
6      resource:
7        userInfoUri: http://auth-service/uaa/user
8      client:
9        client-id: acme
10       access-token-uri: http://auth-service
11   enable-csrf: false
12   ignored: /api/**
13   user:
14     password: admin
15     name: admin
16 eureka:
17   instance:
18     prefer-ip-address: true
19   client:
20     registerWithEureka: true
21     fetchRegistry: true
22     serviceUrl:
23       defaultZone: http://localhost:8761/eureka/
```

(vi) Managing the execution of the business process — As mentioned earlier, an original version of the checkout process has been maintained and a new version of this process has been created to execute it using the Beethoven platform. The new method responsible for executing the checkout process is listed in Code 22. In lines 2-3, a contextual input is instantiated

[5] <https://github.com/Netflix/eureka>

with the OAuth access token that is used to authorize requests from the Beethoven platform. Next, in lines 5-8, a Beethoven command is instantiated with the name of the workflow that should be executed, a command operation to schedule the execution of the checkout process, and the contextual input that has been created in lines 2-3. After that, in lines 10-12, a POST request is performed to the endpoint that is responsible to receive commands to a particular workflow. In this request, the command that was instantiated in the previous lines of the method is passed as the request body.

Code 22 – Beethoven checkout method

```
 1  public CheckoutResult checkoutOrchestrated() {
 2      OAuth2AccessToken accessToken = oAuth2RestTemplate.getAccessToken();
 3      Set<ContextualInput> inputs = newHashSet(new ContextualInput(accessToken));
 4
 5      BeethovenOperation beethovenOperation = new BeethovenOperation();
 6      beethovenOperation.setWorkflowName("checkoutProcess");
 7      beethovenOperation.setOperation(SCHEDULE.getId());
 8      beethovenOperation.setInputs(inputs);
 9
10      restTemplate.postForEntity(
11              "http://beethoven-service/api/workflows/checkoutProcess/operations",
12              beethovenOperation, BeethovenOperation.class);
13
14      return new CheckoutResult();
15  }
```

6.2.4 Benefits

The benefits delivered by the Beethoven platform are related to the quality attributes that have been collected during the process of domain analysis in the definition of the reference architecture. As benefits, the Beethoven platform provides knowledge reuse, process effectiveness, process expressibility, and process expressibility. Each benefit is discussed below.

6.2.4.1 Knowledge reuse

The knowledge reuse is achieved through reusing existing microservices in order to implement new declarative business processes. In addition, by avoiding composed microservices, a microservice-based application does not violate the principle of the microservices architecture style. Another manner to achieve knowledge reuse is by reusing the existing declarative business processes or parts of them in order to create new declarative business processes.

6.2.4.2 Process flexibility

Process flexibility is achieved through the ability to apply modifications to declarative business processes during at both design and run-time. The Beethoven platform provides API for managing declarative business processes in terms of workflows, tasks, and event handlers. For instance, it is possible to alternate a particular event handler from a specific workflow during the execution of a workflow instance. Thus, by using the Beethoven platform for orchestrating microservices, there is no necessity to interrupt the execution of any microservice for applying modifications to business processes.

6.2.4.3 Process effectiveness

In order to provide process effectiveness, the Beethoven platform monitors the execution for each declarative business process in terms of elapsed time of the tasks executed successfully, causes failures of tasks performed unsuccessfully (e.g., timeout, HTTP request errors, or exceptions), states assumed during the execution of each workflow instance, and throughput of workflow executions per time. The architectural component responsible for providing process effectiveness is Report Event Processor (in Beethoven's reference architecture) or ReportActor (in Beethoven's concrete architecture).

6.2.4.4 Process expressibility

Process expressibility refers to the ability to express a business process using elements such as control flow, input data, execution and temporal information. The Beethoven platform provides process expressibility through the features offered by the Partitur language. For instance, it is possible to express control flow and temporal information using the Partitur event handler definition. In addition, the Partitur language adopts a data-driven execution model for specifying the input of data to a specific task from the result obtained after performing another task.

6.3 QUASI-EXPERIMENT

In order to analyze the performance difference between the original and orchestrated versions of the checkout process presented in the reference application, a controlled quasi-experiment has been conducted. Experiments may be human-oriented or technology-oriented. In technology-oriented experiments, typically different tools are applied to different objects, for example, two test case generation tools are applied to the same programs (WOHLIN *et al.*, 2012). The research protocol is composed of the following steps: experiment scoping, experiment planning, analysis and interpretation, presentation and package, and experiment report.

6.3.1 Research protocol

6.3.1.1 Objective definition

The objective of the presented experiment is to investigate the performance assessment of the execution of microservice composition by using the Beethoven platform, from the checkout business process of the reference application presented in Section 6.2. The experiment, as reported here, is part of a larger evaluation of the Beethoven platform focusing on the possible overhead and trade-off that are caused by the platform after orchestrating the original version of the reference application. Since the business processes of the reference application cannot be randomly assigned to subjects, the experiment is, in fact, a quasi-experiment. For the sake of simplicity, only two hypotheses are evaluated here. The data set for the larger study can be found on GitHub[6]. The experiment presented in this chapter uses a subset of the data.

6.3.1.2 Research questions

In order to achieve the aforementioned objective, the following research questions have been formulated:

RQ1: What is the performance difference between the execution of the checkout business process performed in the original and orchestrated versions of the reference application?

The purpose of this questions is to provide empirical evidence for addressing the objective of the presented experiment by investigating the performance difference between the checkout business process performed in the original and orchestrated versions of the reference application. To this end, this question has been divided into two research questions: RQ1.1 and RQ1.2.

6 <https://github.com/davimonteiro/performance-evaluation>

RQ1.1: What is the execution time of the checkout business process performed in the original version of the reference application?

The purpose of this questions is to provide performance data by executing the checkout business process in the original version of the reference application using different testing scenarios and collecting the execution time for each process that has been executed successfully. In addition, information about success and failure rate for each execution is collected.

RQ1.2: What is the execution time of the checkout business process performed in the orchestrated version of the reference application?

The purpose of this questions is to provide performance data by executing the checkout business process in the orchestrated version of the reference application using different testing scenarios and collecting the execution time for each process that has been executed successfully. In addition, information about success and failure rate for each execution is collected.

6.3.1.3 Hypotheses

In software engineering, software performance engineering represents the entire collection of software engineering activities and related analyses used during the software development cycle, which are directed to satisfy performance requirements (WOODSIDE; FRANKS; PETRIU, 2007). Software performance is evaluated from a user's perspective and is typically assessed in terms of quality attributes of the system such as throughput, scalability, reliability, and resource usage. In the presented experiment, the quality attribute used to measure the performance overhead is the time required to execute a specific instance of a business process. This leads to the formulation of the following hypotheses:

- H_{01}. The Beethoven platform does not cause performance overhead during the orchestration of the checkout business process.
- H_{11}. The Beethoven platform causes performance overhead during the orchestration of the checkout business process.

6.3.1.4 Variables

The independent variables are the number of simultaneous users performing the checkout business process and the number of items that a user has added to the shopping cart.

The dependent variable is the execution/elapsed time that is required to execute a specific instance of the checkout business process.

6.3.2 Research setup

The computational environment used in the present experiment is composed of a MacBook Pro (Retina, 13-inch, Late 2013) with 2.4 GHz Intel Core i5 (I5-4258U), 8GB of RAM (1600 MHz DDR3), and Intel Iris with 1536 MB. The machine used for the tests runs macOS High Sierra (version 10.13.5), Java (TM) SE Runtime Environment (build 1.8.0_45—b14), Java HotSpot(TM) 64-Bit Server VM (build 25.45-b02, mixed mode), and Apache Maven 3.5.0[7].

The Gatling testing tool[8] has been used for simulating users requesting an HTTP resource. Gatling is an open-source load and performance testing framework that provides an asynchronous architecture implemented in a non-blocking way for creating virtual users as messages instead of dedicated threads. Gatling is designed to be used as a load testing tool for analyzing and measuring the performance of web applications. By using Gatling, it possible to run thousands of concurrent virtual users.

In order to run Gatling tests, the scenario template in Code 23 has been specified. The scenario template is a simulation file that includes a definition of an HTTP request (line 8), a feed data of products from a CSV file (line 10), and a feed data of user credential data from a CSV file (line 12). In Gatling tests, feed data is a special type of data that is read from a source of data (e.g., JSON file, CSV file, or text file). In addition, the scenario template is composed of a definition of a scenario named *Performance simulation* (line 14). *Performance simulation* is composed of the following HTTP requests: (i) *Perform login* (lines 20-31); (ii) *Add items to the shopping cart* (lines 35-40); and (iii) *Perform the checkout process* (lines 44-47). Finally, in line 51, the scenario definition is executed using a specific number of simultaneous users.

[7] <https://maven.apache.org/>
[8] <https://gatling.io/>

Code 23 – Template for testing scenarios

```scala
import io.gatling.core.Predef._
import io.gatling.http.Predef._

import scala.concurrent.duration._

class PerformanceSimulation extends Simulation {

  val httpProtocol = http

  val products = csv("products.csv").queue

  val users = csv("user_credentials.csv").queue

  val scn = scenario("Performance simulation")
      .repeat(30) {
          pause(10 seconds)
              feed(users)
              feed(products)
              .exec(
                  http("Perform login")
                      .post("http://localhost:8181/uaa/oauth/token")
                      .header("Authorization", "Basic YWNtZTphY21lc2VjcmV0")
                      .header("Content-Type", "application/x-www-form-urlencoded")
                      .formParam("username", "${username}")
                      .formParam("password", "${password}")
                      .formParam("grant_type", "password")
                      .formParam("scope", "openid")
                      .formParam("client_id", "acme")
                      .formParam("client_secret", "acmesecret")
                      .check(jsonPath("$.access_token").exists.saveAs("token"))
                      .check(status is 200)
              )
              .pause(2 seconds)
              .exec(
                  http("Add items to the cart")
                      .post("http://localhost:8957/v1/events")
                      .header("Authorization", "Bearer ${token}")
                      .header("Content-Type", "application/json")
                      .body(StringBody("${product}")).asJSON
                      .check(status is 200)
              )
              .pause(2 seconds)
              .exec(
                  http("Perform the checkout process")
                      .post("http://localhost:8957/v1/checkout")
                      .header("Authorization", "Bearer ${token}")
                      .check(status is 200)
              )
      }

  setUp(scn.inject(atOnceUsers(1))).protocols(httpProtocol)

}
```

6.3.3 Research procedure

The aim of the experiment is to investigate whether the Beethoven platform causes a performance overhead on the execution of the checkout business process presented in the original version of the reference microservices-based application. To this end, the experiment design is two factor with three treatments. The factors in the experiment are the number of simultaneous users and the number of items in the shopping cart. For the experiment design, the following treatments have been used:

Treatments for the first factor (simultaneous user)

- 1 simultaneous user
- 10 simultaneous users
- 20 simultaneous users

Treatments for the second factor (items in the shopping cart)

- 1 item add to the shopping cart
- 10 item add to the shopping cart
- 20 item add to the shopping cart

In order to conduct the experiment design, six different execution scenarios have been prepared based on the template for testing scenarios presented in Code 23. An execution scenario is composed of the following activities: (i) performing the login process using user credentials from a CSV file and retrieving an access token from OAuth2 Server (user-service); (ii) adding a specific number of items using products from a CSV file to the shopping cart (shopping-cart-service); and (iii) performing the checkout process (shopping-cart-service).

The definition of scenarios has been done following the complete factorial experiment, in which each scenario execution has been repeated 30 times, having the confidence interval in the results with 99% of confidence. In Table 11, each execution scenario is presented. The scenarios have been divided into two groups: the original version and the orchestrated version. The former represents the scenarios that have been executed on the original version of the checkout business process, while the latter represents the scenarios that have been executed on the orchestrated version of the checkout business process.

Table 11 – Execution scenarios

Scenario	Version	Simultaneous users	Number of items
1	Original	1	1
2	Original	10	10
3	Original	20	20
4	Orchestrated	1	1
5	Orchestrated	10	10
6	Orchestrated	20	20

Source**:** Elaborated by the author.

6.3.4 Research results

The results presented in this section are based on performance data from the execution of workflow instances that can be found on GitHub[9]. For each execution of a workflow instance, the following data have been stored: workflow name, workflow instance name, start time, end time, elapsed time, and successful execution. In the following subsection, the answers to the research questions are presented and the hypotheses are evaluated using statistical methods.

The execution time of the checkout business process performed in the original version of the reference application (RQ1.1)

The purpose of RQ1.1 is to collect performance data from the execution time of the checkout business process performed in the original version of the reference application. Thus, in order to answer the RQ1.1, the performance data from scenarios of G_1 have been used for collecting the elapsed time of each workflow instance. Based on the collected data, the following information about a workflow instance has been extracted: the minimum elapsed time, the maximum elapsed time, the average, the median, the success rate, and the failure rate.

As result, the scenario 1 produces the following results: (i) minimum elapsed time in milliseconds: 1575; (ii) maximum elapsed time in milliseconds: 3085; and (iii) average of elapsed time in milliseconds: 1753.9666. The scenario 2 produces the following results: (i) minimum elapsed time in milliseconds: 5293; (ii) maximum elapsed time in milliseconds: 61270; and (iii) average of elapsed time in milliseconds: 31455.5833. The scenario 3 produces the following results: (i) minimum elapsed time in milliseconds: 8231; (ii) maximum elapsed time in milliseconds: 90222; and (iii) average of elapsed time in milliseconds: 34714.4534. Table 12 summarizes the results from each scenario presented in G_1.

[9] <https://github.com/davimonteiro/performance-evaluation>

During the execution of workflow instances, only the information concerning instances that have been executed successfully have been registered, following the purpose of the research question RQ1.1 defined in Section 6.3.1.2. Therefore, the minimum, maximum, average, and median have been calculated based on successful execution of workflow instances. In order to provide information regarding the number of workflows that have been executed successfully, the success and failure rates have been calculated. As can be seen in Table 12, the first scenario has a success rate of 100% and a failure rate of 0% because the reference application can handle the requests performed during the execution of this scenario.

However, in scenarios 2 and 3, the success rates are, respectively, 96% and 55.50%. This occurred because the reference application cannot handle the requests generated during the execution of these scenarios. This information confirms that the reference application, within what has been scaled in terms of computational infrastructure, is beyond its normal operational capacity during the execution of scenarios 2 and 3. As result, the circuit breakers of each microservice that comprised the reference application have been opened in order to isolate failures and timeouts that may cascade to the entire microservices-based application.

Table 12 – Results of G_1

Scenario	Minimum (ms)	Maximum (ms)	Average (ms)	Median (ms)	Success rate	Error rate
S1	1575	3085	1753.966667	1683	100%	0%
S2	5293	61270	31455.58333	26815.5	96%	4%
S3	8231	90222	34714.4534	35140	55.50%	44.50%

Source**:** Elaborated by the author.

The execution time of the checkout business process performed in the orchestrated version of the reference application (RQ1.2)

The purpose of RQ1.2 is to collect performance data from the execution time of the checkout business process performed in the orchestrated version of the reference application. Therefore, in order to answer the RQ1.2, the performance data from the scenarios of G_2 have been used for collecting the elapsed time of each workflow instance. Based on the collected data, the following information about a workflow instance has been extracted: the minimum elapsed time, the maximum elapsed time, the average, the median, the success rate, and the failure rate.

As result, the scenario 4 produces the following results: (i) minimum elapsed time in milliseconds: 885; (ii) maximum elapsed time in milliseconds: 1784; and (iii) average of elapsed time in milliseconds: 1023.8. The scenario 5 produces the following results: (i) minimum elapsed time in milliseconds: 1794; (ii) maximum elapsed time in milliseconds: 32581; and (iii) average of elapsed time in milliseconds: 13265.3211. The scenario 6 produces the following results: (i) minimum elapsed time in milliseconds: 182; (ii) maximum elapsed time in milliseconds: 40619; and (iii) average of elapsed time in milliseconds: 16667.5943. Table 13 summarizes the results from each scenario presented in G_2. As mentioned in the answer to question RQ1.1, only the information regarding instances executed successfully have been registered. As can be seen in Table 12, the scenario 4 has a success rate of 100% and the scenarios 5 and 6 have a success rate of 72.67% and 46.83% respectively.

Table 13 – Results of G_2

Scenario	Minimum (ms)	Maximum (ms)	Average (ms)	Median (ms)	Success rate	Error rate
S4	885	1784	1023.8	939	100%	0%
S5	1794	32581	13265.3211	12047	72.67%	27.33%
S6	182	40619	16667.59431	17333	46.83%	53.17%

Source**:** Elaborated by the author.

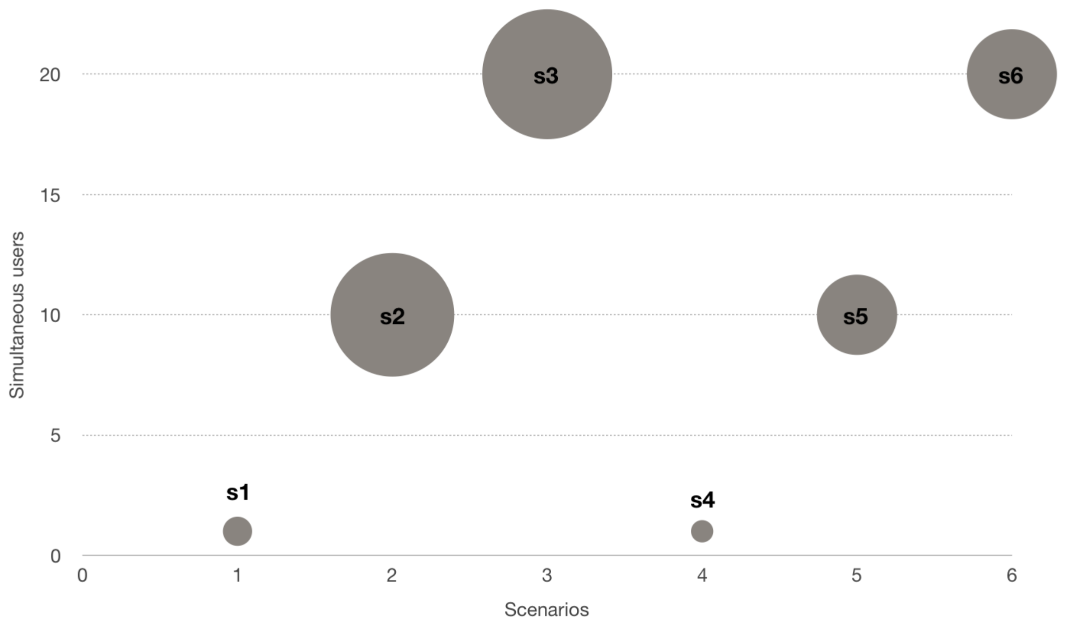
The performance difference between the execution of the checkout business process performed in the original and orchestrated versions of the reference application (RQ1)

The purpose of RQ1 is to provide empirical evidence for investigating the performance difference between the checkout business process performed in the original and orchestrated versions of the reference application. To this end, the performance data collected during the execution of the testing scenarios from G_1 and G_2 have been used to execute hypothesis testing and visualize the performance difference between the execution time of the checkout business process performed in the original and orchestrated versions.

The visualization of the performance difference is achieved by the bubble chat presented in Figure 30. Bubble charts expose correlations between three points of data in a series: x values, y values, and sizes. In the presented bubble chart, the x values are composed of the execution scenarios used during the experiment, the y values are composed of the number of simultaneous users using for executing each scenario, and the sizes are composed of the average of elapsed time consumed during the execution of a business process in milliseconds.

In Figure 30, the first three bubbles represent the scenarios 1, 2, and 3, while the last three bubbles represent the scenarios 4, 5, and 6. By comparing the two groups of scenarios, it is possible to notice that the bubbles of the second group have a smaller size when they are compared to the size of bubbles of the first group. In the following section, the size difference among those bubbles are evaluated statistically.

Figure 30 – Results in bubble chart



Source: Elaborated by the author.

6.3.4.1 Hypothesis testing

The execution scenarios have been divided into two groups (G_1 and G_2) as mentioned in Section 6.3.1.3. The former represents the scenarios that have been executed on the original version, while the latter represents the scenarios that have been executed on the orchestrated version. As result, each scenario produces a set of elapsed time that is required to execute one instance of the checkout business process. The data produced by the execution scenarios have been used to perform the hypothesis testing. In this context, the objective of this experiment is to investigate if $G_1 < G_2$, which means if the elapsed time of the execution of the checkout business process in the orchestrated version is greater than the elapsed time of the execution of the checkout business process in the original version.

The Mann-Whitney-Wilcoxon test is applied to evaluate the hypothesis that the Beethoven platform causes performance overhead during microservice composition. The results from Wilcoxon test are shown in Table 14, which presents the elapsed time averages, standard deviation, the p-value, effect size between the G_1 and G_2. From Table 14, considering a confidence level of 99%, $\alpha = 0.01$, it can be concluded that the difference in terms of the average of elapsed time between G_1 and G_2 is statistically significant since the effect size is large for all scenarios. In the comparison between the scenarios 1 and 4, the Beethoven platform causes a performance improvement of 41.16%. In the comparison between the scenarios 2 and 3, the Beethoven platform improves the performance of microservice composition in 57.82%. Finally, in in the comparison between the scenarios 4 and 6, the Beethoven platform improves the performance of microservice composition in 51.98%. Therefore, H_{11} is rejected considering the comparison among all execution scenarios. The actual reason for the difference has to be further evaluated.

Table 14 – Elapsed time averages, standard deviation, p-values, and effect sizes

Scenarios	Elapsed time average		Standard deviation		p-value	$\hat{A}_{12}$
	G_1	G_2	G_1	G_2		
S1 and S4	1753.967	1023.8	270.8196	216.7577	9.27378e-10	0.9605556 (large)
S2 and S5	31455.58	13265.32	12924.34	6971.667	6.411714e-48	0.8772856 (large)
S3 and S6	34714.45	16667.59	15889.22	8770.868	1.751488e-51	0.8532696 (large)

Source: Elaborated by the author.

6.3.5 Threats to validity

A fundamental question concerning results of an experiment is their validity. Therefore, during the condition of the experiment, some concerns about the validity of the research protocol, setup, procedure have been recognized.

External—The reference MSA-based application represents an external threat to validity since the results of the experiment cannot be generalized based on only one application. However, since microservices is a new topic in the scientific and industry communities, there is no repository of open source projects containing a microservices-based application that execute distributed business processes.

Internal—The experiment setup is limited. Ideally, it should be used a cloud infrastructure and each microservice should run on a particular container (e.g., Docker). However, due time and budget constraints, the ideal scenario is not practical. In addition, the number of execution scenarios and the independent variables may present limitations to generalize the results. Another threat is that the experiment has been planned, conducted, and documented by the author.

7 CONCLUSION

This final chapter brings this dissertation to a conclusion. To this end, Section 7.1 summarizes the main contributions, shows how to the limitations of the related works have been addressed, presents the results from the evaluations (two example applications and one quasi-experiment), and exhibits how the specific objectives have been accomplished. After that, Section 7.3 presents a list of published papers as a result of the master's degree. Next, Section 7.3 presents the main limitation of the Beethoven platform, including the DSL orchestration, the concrete architecture, and the performed evaluations. Finally, Section 7.4 presents and outlines further work to improve the capabilities of Beethoven, Beethoven Spring Cloud, and Partitur. In addition, this section presents further research directions expanding the Beethoven platform to the domain of self-adaptive software systems engineering.

7.1 SUMMARY

This dissertation presents an event-driven platform, named Beethoven, for microservice orchestration that facilitates the creation of complex applications using microservice data flows. The Beethoven platform is composed of: (i) a reference architecture that is described systematically using the ProSA-RA methodology; and (ii) an orchestration DSL, named Partitur, based on declarative business processes. In order to instantiate the reference architecture, a concrete architecture, named Spring Cloud Beethoven, has been implemented based on the actor model and the Spring Cloud Netflix ecosystem.

Spring Cloud Beethoven provides integration for Spring Boot applications using auto-configuration and binding to Spring Cloud Netflix components. Therefore, in order to address dynamic microservice location, Spring Cloud Beethoven relies on Spring Cloud Eureka for service discovery and Spring Cloud Ribbon for the client-side load balancer. As consequence, there is no need to describe and register previously each microservice for performing orchestration. Thus, new microservices that are added to a microservices-based application can be used during the microservice composition.

In order to evaluate the proposed platform, two example applications have been developed and one controlled experiment has been conducted. The first example application is a microservices-based application that implements a CRM system for an investment bank. Besides, it is important to emphasize that the first example application has been developed by the author of this dissertation in order to demonstrate the feasibility of the Beethoven platform. To reduce the research bias during the evaluation of the proposed platform, a second example application has been developed by using a reference microservices-based application that has been adapted in order to use the Beethoven platform for performing microservice orchestration. As result, two versions of the reference application have been maintained: the original and orchestrated versions.

Finally, in order to analyze the performance differences between the original and orchestrated versions of the reference application, a controlled quasi-experiment has been conducted. After conducting the experiment, considering a confidence level of 99% and $\alpha = 0.01$, it can be concluded that the difference in terms of the performance between the two versions of the reference application is statistically significant. The alternative hypothesis H_{11} of the experiment (the Beethoven platform causes performance overhead during the orchestration of the checkout business process) has been rejected since the Beethoven platform provides a performance improvement for all the execution scenarios executed during the experiment.

In order to accomplish the specific objectives presented in Section 1.2.2, the following steps have been completed:

(i) In order to design and specify systematically an extensible and scalable event-driven lightweight reference architecture for microservice orchestration, a reference architecture has been specified using a methodology for defining software reference architectures named ProSA-RA;

(ii) In order to design and implement an orchestration DSL based on declarative business processes, an orchestration DSL, named Partitur (available on GitHub[1]) has been created;

(iii) In order to recognize the viability of the Beethoven platform, a concrete architecture, named Spring Cloud Beethoven (available on GitHub[2]), has been implemented based on the specification of the Beethoven's reference architecture;

(iv) In order to verify the feasibility of the Beethoven platform and its concrete architecture, an example application has been developed (available on GitHub[3]), demonstrating how the proposed platform can be used;

(v) In order to confirm the feasibility of the Beethoven platform and its concrete architecture, a second example application has been developed based on an existing reference application that has been adapted to use the Beethoven platform (available on GitHub[4]);

[1] <https://github.com/davimonteiro/partitur>
[2] <https://github.com/davimonteiro/beethoven>
[3] <https://github.com/davimonteiro/crm-msa-example>
[4] <https://github.com/davimonteiro/reference-msa-application>

(vi) In order to evaluate a possible overhead that the Beethoven platform and its concrete architecture may produce during microservice orchestration, a controlled quasi-experiment has been conducted. All the dataset and result from this experiment are available to the scientific community on GitHub[5].

7.2 PUBLICATIONS

In this Section, the list of published papers as a result of the master's degree and contribution to the research goals of this dissertation is presented as follow.

1. Davi Monteiro, Rômulo Gadelha, Paulo Henrique M. Maia, Lincoln S. Rocha, and Nabor C. Mendonça. Beethoven: An Event-Driven Lightweight Platform for Microservice Orchestration. In Proceedings of the ACM 12th European Conference on Software Architecture. ECSA 2018. (Qualis B1) (MONTEIRO *et al.*, 2018)
2. Davi Monteiro, Rômulo Gadelha, Thayse Alencar, Bruno Neves, Italo Yeltsin, Thiago Gomes, and Mariela Cortés. An Analysis of the Empirical Software Engineering over the last 10 Editions of Brazilian Software Engineering Symposium. In Proceedings of the 31st Brazilian Symposium on Software Engineering. SBES 2017. (Qualis B2) (MONTEIRO *et al.*, 2017)
3. Davi Monteiro, Rômulo Gadelha, Paulo Henrique M. Maia, and Evilásio Costa. Lotus@runtime: A Tool for Runtime Monitoring and Verification of Self-Adaptive Systems. In Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS 2017. (Qualis A2) (MONTEIRO *et al.*, 2017)
4. Davi Monteiro, Rômulo Gadelha, Paulo Henrique M. Maia, and Evilásio Costa. Lotus@Runtime: Uma Ferramenta para Monitoramento e Verificação em Tempo de Execução para Sistemas Autoadaptativos. In: Simpósio Brasileiro de Computação Ubíqua e Pervasiva. SBCUP 2016. (Qualis B4) (MONTEIRO *et al.*, 2016)

7.3 LIMITATION

As limitations, the Beethoven platform is based on a reactive behavior to events and commands that are exchanged among its event processors. In this manner, failures and timeouts that may occur during the execution of workflow instances must be handled by a microservice engineer using the orchestration DSL. For example, if a task execution fails, the engineer must

[5] <https://github.com/davimonteiro/performance-evaluation>

specify an event handler for that failure and define which command should be performed after the failure has occurred.

Another limitation of this dissertations is the orchestration DSL that has not been validated. For instance, there are research questions regarding the proposed DSL is enough for modeling different requirements in workflow fulfilling or whether the Partitur elements are suitable for dealing with different kind of workflows. Therefore, in order to ensure the completeness of the proposed reference architecture for workflow modeling, a validation regarding existing workflow patterns in literature has to be done.

In addition, there are some desired features for DSLs based on declarative business processes that need to be evaluated in Partitur such as the expressibility of a process modeling language and the level of comprehensibility. The former refers to the ability to express specific process elements (e.g., control-flow, data, execution and temporal information) (LU; SADIQ, 2007), while the latter refers to the ability of a process modelling language to define understandable process models that can be easily understandable among various stakeholders (e.g., application developers, business analysts, and business owners) (FAHLAND *et al.*, 2009).

7.4 FUTURE WORK

As future work, self-adaptive software systems engineering for microservice orchestration will be used to improve the reliability, flexibility, and resilience of the Beethoven platform. Specifically, further research directions intend to evolve from a reactive to a proactive platform by supporting proactive adaptation strategies during microservice orchestration. This allows dynamic microservices orchestration by replacing microservice operations at runtime and offering support to adaptation based on goals or policies. In order to address limitations of Partitur, empirical validations (e.g., surveys, case studies, and experiments) has to be done, collecting information to guide further research. In addition, the performance evaluation that has been conducted during the quasi-experiment has to be extended by using different execution scenarios such as reactive and proactive situations. At last, but not least, since the platform is an open source software, it is expected to receive feedback from both academic and industrial communities that can be used to improve the platform.

BIBLIOGRAPHY

AALST, W. M. P. van der; ROSA, M. L.; SANTORO, F. M. Business process management. **Business & Information Systems Engineering**, New York, NY, USA, v. 58, n. 1, p. 1–6, Feb 2016. ISSN 1867-0202.

AGHA, G. A.; MASON, I. A.; SMITH, S. F.; TALCOTT, C. L. A foundation for actor computation. **Journal of Functional Programming**, New York, NY, USA, v. 7, n. 1, p. 1–72, Jan 1997. ISSN 0956-7968.

ALFERES, J. J.; BANTI, F.; BROGI, A. An event-condition-action logic programming language. In: FISHER, M.; HOEK, W. van der; KONEV, B.; LISITSA, A. (Ed.). **Logics in Artificial Intelligence**. Berlin, Germany: Springer, 2006.

ALONSO, G.; CASATI, F.; KUNO, H.; MACHIRAJU, V. **Web Services: Concepts, Architectures and Applications**. Berlin, Germany: Springer, 2004.

ANGELOV, S.; GREFEN, P.; GREEFHORST, D. A classification of software reference architectures: Analyzing their success and effectiveness. In: **European Conference on Software Architecture**. Cambridge, UK: [s.n.].

ANGELOV, S.; GREFEN, P.; GREEFHORST, D. A framework for analysis and design of software reference architectures. **Information and Software Technology**, Newton, MA, USA, v. 54, n. 4, p. 417–431, Apr 2012. ISSN 0950-5849.

AUSTIN, R. D.; DEVIN, L. Research commentary—weighing the benefits and costs of flexibility in making software: Toward a contingency theory of the determinants of development process design. **Information Systems Research**, Linthicum, MD, USA, v. 20, n. 3, p. 462–477, Sep 2009. ISSN 1526-5536.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Migrating to cloud-native architectures using microservices: An experience report. In: CELESTI, A.; LEITNER, P. (Ed.). **Advances in Service-Oriented and Cloud Computing**. Berlin, Germany: Springer, 2016.

BANâTRE, J.-P.; FRADET, P.; RADENAC, Y. Generalised multisets for chemical programming. **Mathematical Structures in Computer Science**, Cambridge University Press, New York, NY, USA, v. 16, n. 4, p. 557–580, Aug 2006. ISSN 0960-1295.

BANâTRE, J.-P.; FRADET, P.; RADENAC, Y.

BARKER, A.; WALTON, C. D.; ROBERTSON, D. Choreographing web services. **IEEE Transactions on Services Computing**, Piscataway, NJ, USA, v. 2, n. 2, p. 152–166, Apr 2009. ISSN 1939-1374.

BARROS, A.; DUMAS, M.; OAKS, P. Standards for web service choreography and orchestration: Status and perspectives. In: SPRINGER. **Business process management workshops**. [S.l.], 2005. v. 3812, p. 61–74.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. 3rd. ed. Boston, MA, USA: Addison-Wesley, 2012.

BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. **IEEE Cloud Computing**, Piscataway, NJ, USA, v. 1, n. 3, p. 81–84, Sep 2014. ISSN 2325-6095.

CERNY, T.; DONAHOO, M. J.; PECHANEC, J. Disambiguation and comparison of soa, microservices and self-contained systems. In: **Proceedings of the International Conference on Research in Adaptive and Convergent Systems**. New York, NY, USA: ACM, 2017. (RACS '17), p. 228–235. ISBN 978-1-4503-5027-3. Disponível em: <http://doi.acm.org/10.1145/3129676.3129682>.

CERNY, T.; DONAHOO, M. J.; TRNKA, M. Contextual understanding of microservice architecture: Current and future directions. **Applied Computing Review**, New York, NY, USA, v. 17, n. 4, p. 29–45, Jan 2018. ISSN 1559-6915.

CHANNABASAVAIAH, K.; HOLLEY, K.; TUGGLE, E. Migrating to a service-oriented architecture. **IBM DeveloperWorks**, v. 16, p. 727–728, 2003.

CONWAY, M. E. How do committees invent. **Datamation**, v. 14, n. 4, p. 28–31, Feb 1968.

DUGGAN, D. **Enterprise Software Architecture and Design: Entities, Services, and Resources**. Hoboken, NJ, USA: John Wiley & Sons, 2012. v. 10.

ESCOBAR, D.; CáRDENAS, D.; AMARILLO, R.; CASTRO, E.; GARCéS, K.; PARRA, C.; CASALLAS, R. Towards the understanding and evolution of monolithic applications as microservices. In: **2016 XLII Latin American Computing Conference (CLEI)**. [S.l.: s.n.], 2016. p. 1–11.

EUGSTER, P. T.; FELBER, P. A.; GUERRAOUI, R.; KERMARREC, A.-M. The many faces of publish/subscribe. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 35, n. 2, p. 114–131, Jun 2003. ISSN 0360-0300.

EVANS, E. **Domain-Driven Design: Tacking Complexity In the Heart of Software**. Boston, MA, USA: Addison-Wesley, 2003.

FAHLAND, D.; LÜBKE, D.; MENDLING, J.; REIJERS, H.; WEBER, B.; WEIDLICH, M.; ZUGAL, S. Declarative versus imperative process modeling languages: The issue of understandability. In: **Enterprise, Business-Process and Information Systems Modeling**. Berlin, Germany: Springer, 2009.

FAZIO, M.; CELESTI, A.; RANJAN, R.; LIU, C.; CHEN, L.; VILLARI, M. Open issues in scheduling microservices in the cloud. **IEEE Cloud Computing**, v. 3, n. 5, p. 81–88, Sep 2016. ISSN 2325-6095.

FERNáNDEZ, H.; TEDESCHI, C.; PRIOL, T. A chemistry-inspired workflow management system for decentralizing workflow execution. **IEEE Transactions on Services Computing**, v. 9, n. 2, p. 213–226, Mar 2016. ISSN 1939-1374.

FIELDING, R. T.; TAYLOR, R. N. Principled design of the modern web architecture. **ACM Transactions on Internet Technology**, ACM, New York, NY, USA, v. 2, n. 2, p. 115–150, May 2002. ISSN 1533-5399.

FINK, L.; NEUMANN, S. Exploring the perceived business value of the flexibility enabled by information technology infrastructure. **Information & Management**, Cambridge, MA, USA, v. 46, n. 2, p. 90–99, Mar 2009. ISSN 0378-7206.

FOWLER, M. **FluentInterface**. [S.l.], 2005. Disponível em: <https://martinfowler.com/bliki/FluentInterface.html>.

FRANCESCO, P. D. Architecting microservices. In: **2017 IEEE International Conference on Software Architecture Workshops (ICSAW)**. [S.l.: s.n.], 2017. p. 224–229.

FRANCESCO, P. D.; MALAVOLTA, I.; LAGO, P. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: **2017 IEEE International Conference on Software Architecture (ICSA)**. [S.l.: s.n.], 2017. p. 21–30.

FUGGETTA, A.; PICCO, G. P.; VIGNA, G. Understanding code mobility. **IEEE Transactions on Software Engineering**, IEEE, Piscataway, NJ, USA, v. 24, n. 5, p. 342–361, 1998. ISSN 0098-5589.

GALSTER, M.; AVGERIOU, P. Empirically-grounded reference architectures: A proposal. In: **Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS**. New York, NY, USA: ACM, 2011. (QoSA-ISARCS '11), p. 153–158. ISBN 978-1-4503-0724-6. Disponível em: <http://doi.acm.org/10.1145/2000259.2000285>.

GARLAN, D.; SHAW, M. **An Introduction to Software Architecture**. Pittsburgh, PA, USA, 1994.

GASEVIC, D.; GROSSMANN, G.; HALLE, S. Dynamic and declarative business processes. In: **2009 13th Enterprise Distributed Object Computing Conference Workshops**. [S.l.: s.n.], 2009. p. 1–4. ISSN 2325-6583.

GOEDERTIER, S.; VANTHIENEN, J.; CARON, F. Declarative business process modelling: principles and modelling languages. **Enterprise Information Systems**, Taylor & Francis, Bristol, PA, USA, v. 9, n. 2, p. 161–185, 2015. ISSN 1751-7575.

HALLER, P.; ODERSKY, M. Scala actors: Unifying thread-based and event-based programming. **Theoretical Computer Science**, Cambridge, MA, USA, v. 410, n. 2-3, p. 202–220, Feb 2009. ISSN 0304-3975.

HAYWOOD, D. **In Defence of the Monolith, Part 1**. 2017. Disponível em: <https://www.infoq.com/articles/monolith-defense-part-1>.

HAYWOOD, D. **In Defence of the Monolith, Part 2**. 2017. Disponível em: <https://www.infoq.com/articles/monolith-defense-part-2>.

HEWITT, C.; BISHOP, P.; STEIGER, R. A universal modular actor formalism for artificial intelligence. In: **Proceedings of the 3rd International Joint Conference on Artificial Intelligence**. San Francisco, CA, USA: [s.n.], 1973. (IJCAI'73), p. 235–245. Disponível em: <http://dl.acm.org/citation.cfm?id=1624775.1624804>.

HUHNS, M. N.; SINGH, M. P. Service-oriented computing: key concepts and principles. **IEEE Internet Computing**, Piscataway, NJ, USA, v. 9, n. 1, p. 75–81, Jan 2005. ISSN 1089-7801.

INSTITUTE, P. M. **A Guide to the Project Management Body of Knowledge: PMBOK Guide**. 5. ed. Philadelphia, Pennsylvania, USA: Project Management Institute, 2013. (PMBOK® Guide Series).

JARADAT, W.; DEARLE, A.; BARKER, A. A dataflow language for decentralised orchestration of web service workflows. In: **2013 IEEE Ninth World Congress on Services**. [S.l.: s.n.], 2013. p. 13–20. ISSN 2378-3818.

JOSUTTIS, N. **Soa in Practice: The Art of Distributed System Design**. Sebastopol, CA, EUA: O'Reilly Media, 2007.

KARMANI, R. K.; SHALI, A.; AGHA, G. Actor frameworks for the jvm platform: a comparative analysis. In: ACM. **Proceedings of the 7th International Conference on Principles and Practice of Programming in Java**. [S.l.], 2009. p. 11–20.

KAUR, A.; MANN, K. S. Component based software engineering. **International Journal of Computer Applications**, Foundation of Computer Science, New York, NY, USA, v. 2, n. 1, p. 105–108, May 2010. ISSN 0975-8887.

KECSKEMETI, G.; MAROSI, A. C.; KERTESZ, A. The entice approach to decompose monolithic services into microservices. In: **2016 International Conference on High Performance Computing Simulation (HPCS)**. [S.l.: s.n.], 2016. p. 591–596.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.-M.; IRWIN, J. Aspect-oriented programming. In: SPRINGER. **European conference on object-oriented programming**. Berlin, Germany, 1997. p. 220–242.

KIM, W. Y.; AGHA, G. Efficient support of location transparency in concurrent object-oriented programming languages. In: **Proceedings of the IEEE/ACM SC95 Conference**. [S.l.: s.n.], 1995. p. 39–39.

KOSTER, J. D.; CUTSEM, T. V.; MEUTER, W. D. 43 years of actors: A taxonomy of actor models and their key properties. In: **Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control**. New York, NY, USA: [s.n.], 2016. (AGERE 2016), p. 31–40. ISBN 978-1-4503-4639-9. Disponível em: <http://doi.acm.org/10.1145/3001886.3001890>.

KŘIKAVA, F.; COLLET, P.; FRANCE, R. B. Actor-based runtime model of adaptable feedback control loops. In: ACM. **Proceedings of the 7th Workshop on Models@ run. time**. [S.l.], 2012. p. 39–44.

KYRIACOU, C.; EVRIPIDOU, P.; TRANCOSO, P. Data-driven multithreading using conventional microprocessors. **Transactions on Parallel and Distributed Systems**, Piscataway, NJ, USA, v. 17, n. 10, p. 1176–1188, Oct 2006. ISSN 1045-9219.

LASKEY, K. B.; LASKEY, K. Service oriented architecture. **WIREs Computational Statistics**, New York, NY, USA, v. 1, n. 1, p. 101–105, Jul 2009. ISSN 1939-0068.

LEWIS, J.; FOWLER, M. **Microservices**. 2014. Disponível em: <http://www.martinfowler.com/articles/microservices.html>.

LU, R.; SADIQ, S. A survey of comparative business process modeling approaches. In: **Proceedings of the 10th International Conference on Business Information Systems**. Berlin, Germany: [s.n.], 2007. (BIS'07), p. 82–94. ISBN 978-3-540-72034-8.

MARTíNEZ-FERNáNDEZ, S.; AYALA, C. P.; FRANCH, X.; MARQUES, H. M. Benefits and drawbacks of software reference architectures. **Information and Software Technology**, Newton, MA, USA, v. 88, n. C, p. 37–52, Aug 2017. ISSN 0950-5849.

MAZLAMI, G.; CITO, J.; LEITNER, P. Extraction of microservices from monolithic software architectures. In: **2017 IEEE International Conference on Web Services (ICWS)**. [S.l.: s.n.], 2017. p. 524–531.

MILANOVIC, N.; MALEK, M. Current solutions for web service composition. **IEEE Internet Computing**, IEEE, Piscataway, NJ, USA, v. 8, n. 6, p. 51–59, Nov 2004. ISSN 1089-7801.

MONTEIRO, D.; GADELHA, R.; ALENCAR, T.; NEVES, B.; YELTSIN, I.; GOMES, T.; CORTÉS, M. An analysis of the empirical software engineering over the last 10 editions of brazilian software engineering symposium. In: **Proceedings of the 31st Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2017. (SBES'17), p. 44–53. ISBN 978-1-4503-5326-7.

MONTEIRO, D.; GADELHA, R.; MAIA, P. H. M.; ROCHA, L. S.; MENDONçA, N. C. Beethoven: An event-driven lightweight platform for microservice orchestration. In: **Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings**. New York, NY, USA: ACM, 2018. (ECSA '18).

MONTEIRO, D.; LIMA, R. G. D. M.; MAIA, P. H. M.; COSTA, E. Lotus@runtime: Uma ferramenta para monitoramento e verificação em tempo de execução para sistemas autoadaptativos. in: Simpósio brasileiro de computação ubíqua e pervasiva. In: **VIII Simpósio Brasileiro de Computação Ubíqua e Pervasiva (SBCUP)**. [S.l.: s.n.], 2016.

MONTEIRO, D.; LIMA, R. G. D. M.; MAIA, P. H. M.; COSTA, E. Lotus@runtime: A tool for runtime monitoring and verification of self-adaptive systems. In: **2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**. [S.l.: s.n.], 2017. p. 24–30.

MONTESI, F.; GUIDI, C.; ZAVATTARO, G. Service-oriented programming with jolie. In: ____. **Web Services Foundations**. New York, NY, USA: [s.n.], 2014. p. 81–107. Disponível em: <https://doi.org/10.1007/978-1-4614-7518-7_4>.

MONTESI, F.; WEBER, J. Circuit breakers, discovery, and API gateways in microservices. **Computing Research Repository**, abs/1609.05830, 2016. Disponível em: <http://arxiv.org/abs/1609.05830>.

MUNAFÒ, M. R.; NOSEK, B. A.; BISHOP, D. V.; BUTTON, K. S.; CHAMBERS, C. D.; SERT, N. P. du; SIMONSOHN, U.; WAGENMAKERS, E.-J.; WARE, J. J.; IOANNIDIS, J. P. A manifesto for reproducible science. **Nature Human Behaviour**, Nature Publishing Group, London, UK, v. 1, p. 0021, 2017.

MURGUZUR, A.; INTXAUSTI, K.; URBIETA, A.; TRUJILLO, S.; SAGARDUI, G. Process flexibility in service orchestration: A systematic literature review. **International Journal of Cooperative Information Systems**, Jersey City, NJ, USA, v. 23, n. 3, p. 1430001.1–1430001.31, Jun 2014. ISSN 1793-6365.

NADAREISHVILI, I.; MITRA, R.; MCLARTY, M.; AMUNDSEN, M. **Microservice Architecture: Aligning Principles, Practices, and Culture**. Sebastopol, CA, EUA: O'Reilly Media, Inc., 2016.

NAKAGAWA, E. Y. **Uma contribuição ao projeto arquitetural de ambientes de engenharia de software**. Tese (Doutorado) — Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2006.

NANDA, M. G.; CHANDRA, S.; SARKAR, V. Decentralizing execution of composite web services. **ACM SIGPLAN Notices**, ACM, New York, NY, USA, v. 39, n. 10, p. 170–187, Oct 2004. ISSN 0362-1340.

NEUMAN, S. **Building Microservices: Designing Fine-Grained Systems**. Sebastopol, CA, EUA: O'Reilly Media, 2015.

NEWMAN, S. **Building Microservices**. Sebastopol, CA, EUA: O'Reilly Media, 2015.

NURCAN, S. A survey on the flexibility requirements related to business processes and modeling artifacts. In: IEEE. **Hawaii International Conference on System Sciences, Proceedings of the 41st Annual**. [S.l.], 2008. p. 378–378.

NYGARD, M. **Release it!: design and deploy production-ready software**. Raleigh, NC, USA: Pragmatic Bookshelf, 2007.

OBERHAUSER, R. Microflows: Lightweight automated planning and enactment of workflows comprising semantically-annotated microservices. In: SPRINGER. **Proceedings of the Sixth International Symposium on Business Modeling and Software Design (BMSD 2016)**. Berlin, Germany, 2016. p. 134–143.

PAPAZOGLOU, M. P. Service-oriented computing: concepts, characteristics and directions. In: **Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003. WISE 2003.** [S.l.: s.n.], 2003. p. 3–12.

PAPAZOGLOU, M. P.; HEUVEL, W.-J. van den. Service oriented architectures: approaches, technologies and research issues. **The International Journal on Very Large Data Bases**, New York, NY, USA, v. 16, n. 3, p. 389–415, Jul 2007. ISSN 1066-8888.

PAUTASSO, C.; ZIMMERMANN, O.; AMUNDSEN, M.; LEWIS, J.; JOSUTTIS, N. Microservices in practice, part 1: Reality check and service design. **IEEE Software**, Piscataway, NJ, USA, v. 34, n. 1, p. 91–98, Jan 2017. ISSN 0740-7459.

PAUTASSO, C.; ZIMMERMANN, O.; AMUNDSEN, M.; LEWIS, J.; JOSUTTIS, N. Microservices in practice, part 2: Service integration and sustainability. **IEEE Software**, Piscataway, NJ, USA, v. 34, n. 2, p. 97–104, Mar 2017. ISSN 0740-7459.

PELTZ, C. Web services orchestration and choreography. **Computer**, IEEE, Piscataway, NJ, USA, v. 36, n. 10, p. 46–52, 2003.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **ACM SIGSOFT Software Engineering Notes**, New York, NY, USA, v. 17, n. 4, p. 40–52, Oct 1992. ISSN 0163-5948.

RAO, A. S.; GEORGEFF, M. P. *et al.* Bdi agents: From theory to practice. In: **ICMAS**. [S.l.: s.n.], 1995. v. 95, p. 312–319.

RICHARDS, M. **Microservices vs. Service-Oriented Architecture**. Sebastopol, CA, EUA: O'Reilly Media, 2015.

RICHARDS, M. **Software architecture patterns**. Sebastopol, CA, EUA: O'Reilly Media, 2015.

SAFINA, L.; MAZZARA, M.; MONTESI, F.; RIVERA, V. Data-driven workflows for microservices: Genericity in jolie. In: **2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)**. [S.l.: s.n.], 2016. p. 430–437. ISSN 1550-445X.

SALAH, T.; ZEMERLY, M. J.; YEUN, C. Y.; AL-QUTAYRI, M.; AL-HAMMADI, Y. The evolution of distributed systems towards microservices architecture. In: **2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)**. [S.l.: s.n.], 2016. p. 318–325.

SANDERS, D. T.; HAMILTON JR., J. A.; MACDONALD, R. A. Supporting a service-oriented architecture. In: **Proceedings of the 2008 Spring Simulation Multiconference**. San Diego, CA, USA: Society for Computer Simulation International, 2008. (SpringSim '08), p. 325–334. ISBN 1-56555-319-5. Disponível em: <http://dl.acm.org/citation.cfm?id=1400549.1400595>.

SEBESTA, R. W. **Concepts of Programming Languages**. 10th. ed. London, England: Pearson, 2012.

SHADIJA, D.; REZAI, M.; HILL, R. Towards an understanding of microservices. In: IEEE. **Automation and Computing (ICAC), 2017 23rd International Conference on**. [S.l.], 2017. p. 1–6.

SHAW, M. Writing good software engineering research papers. In: IEEE. **Proceedings of the 25th International Conference on Software Engineering (ICSE)**. [S.l.], 2003. p. 726–736.

SHAW, M.; GARLAN, D. **Software Architecture: Perspectives on an Emerging Discipline**. NJ, USA: Prentice-Hall, 1996.

SILVA, N. C.; CARVALHO, R. M. de; OLIVEIRA, C. A. L.; LIMA, R. M. F. Reflex: An efficient web service orchestrator for declarative business processes. In: SPRINGER. **International Conference on Service-Oriented Computing**. [S.l.], 2013. p. 222–236.

SOMMERVILLE, I. **Software Engineering**. 9th. ed. Boston, MA, USA: Addison-Wesley, 2010. ISBN 0137035152, 9780137035151.

THöNES, J. Microservices. **IEEE Software**, Piscataway, NJ, USA, v. 32, n. 1, p. 116–116, Jan 2015. ISSN 0740-7459.

VASILECAS, O.; KALIBATIENE, D.; LAVBIC, D. Rule- and context-based dynamic business process modelling and simulation. **Journal of Systems and Software**, New York, NY, USA, v. 122, n. C, p. 1–15, Dec 2016. ISSN 0164-1212.

VILLAMIZAR, M.; OCHOA, L.; CASTRO, H.; SALAMANCA, L.; VERANO, M.; CASALLAS, R.; GIL, S.; VALENCIA, C.; ZAMBRANO, A.; LANG, M. *et al.* Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In: IEEE. **Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on**. [S.l.], 2016. p. 179–182.

WANG, C.; PAZAT, J. L. A chemistry-inspired middleware for self-adaptive service orchestration and choreography. In: **2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing**. [S.l.: s.n.], 2013. p. 426–433.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. Berlin, Germany: Springer, 2012.

WOLFF, E. **Microservices: Flexible Software Architecture**. Boston, MA, USA: Addison-Wesley, 2016.

WOODSIDE, M.; FRANKS, G.; PETRIU, D. C. The future of software performance engineering. In: **2007 Future of Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. (FOSE '07), p. 171–187. ISBN 0-7695-2829-5. Disponível em: <http://dx.doi.org/10.1109/FOSE.2007.32>.

XIAO, Z.; WIJEGUNARATNE, I.; QIANG, X. Reflections on soa and microservices. In: **2016 4th International Conference on Enterprise Systems (ES)**. [S.l.: s.n.], 2016. p. 60–67.

YAHIA, E. B. H.; RÉVEILLÈRE, L.; BROMBERG, Y.-D.; CHEVALIER, R.; CADOT, A. Medley: An event-driven lightweight platform for service composition. In: ____. **Web Engineering: 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings**. Berlin, Germany: Springer, 2016.

APPENDICES

APPENDIX A – Partitur DSL specification

Code 24 – Partitur DSL specification

```
grammar io.beethoven.partitur.Partitur with org.eclipse.xtext.common.Terminals

generate partitur "http://www.beethoven.io/partitur/Partitur"


PartiturWorkflow:
    'workflow' name=ID '{'
        (tasks+=PartiturTask)*
        (handlers+=PartiturHandler)*
     '}'
;

PartiturTask:
    'task' name=ID '{'
        (
            partiturHttpRequest = HttpGet |
            partiturHttpRequest = HttpPost |
            partiturHttpRequest = HttpPut |
            partiturHttpRequest = HttpDelete
        )
     '}'
;

PartiturHandler:
    'handler' name=ID '{'
        'on' (event = EventType)
        'when' conditions+=PartiturCondition (',' conditions+=PartiturCondition)*
        'then' commands+=PartiturCommand (',' commands+=PartiturCommand)*
     '}'
;

PartiturCondition:
    (conditionFunction=PartiturConditionFunction)'('arg=STRING')'
;

enum PartiturConditionFunction:
    taskNameEqualsTo = 'taskNameEqualsTo' |
    taskResponseEqualsTo = 'taskResponseEqualsTo' |
    workflowNameEqualsTo = 'workflowNameEqualsTo'
;

PartiturCommand:
    (commandFunction=PartiturCommandFunction)'('arg=STRING')'
;

enum PartiturCommandFunction:
    startTask = 'startTask' |
    startWorkflow = 'startWorkflow' |
    stopWorkflow = 'stopWorkflow' |
    cancelWorkflow = 'cancelWorkflow'
;

enum EventType:
    // Task events
    TASK_STARTED |
    TASK_COMPLETED |
    TASK_TIMEDOUT |
    TASK_FAILED |

    // Workflow events
    WORKFLOW_SCHEDULED |
    WORKFLOW_STARTED |
    WORKFLOW_COMPLETED
;

HttpGet:
    'get' '(' url = STRING ')'
        (uriVariables = UriVariables)?
        (headers += HttpHeader)*
        (params += QueryParam)*
;

HttpPost:
    'post' '(' url = STRING ')'
        (uriVariables = UriVariables)?
        (headers += HttpHeader)*
        (body = HttpBody)?
;

HttpPut:
    'put' '(' url = STRING ')'
        (uriVariables = UriVariables)?
        (headers += HttpHeader)*
        (body = HttpBody)?
;

HttpDelete:
    'delete' '(' url = STRING ')'
        (uriVariables = UriVariables)?
        (headers += HttpHeader)*
;

HttpHeader:
    '.header(' name = STRING ',' value = STRING  ')'
;

UriVariables:
    '.uriVariables(' values += STRING (',' values += STRING)* ')'
;

QueryParam:
    '.queryParams(' name = STRING ',' value = STRING ')'
;

HttpBody:
    '.body(' value = STRING ')'
;
```

APPENDIX B – Beethoven's endpoints

Send command operations

- **HTTP request**: `POST api/workflows/{workflowName}/operations`
- **Description**: Responsible for sending command operations (e.g. start, stop, or cancel) to a particular workflow.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier.
- **Request body (required)**: A command operation in JSON.

Create a new workflow

- **HTTP request**: `POST api/workflows`
- **Description**: Responsible for creating a new workflow.
- **Request body (required)**: A workflow definition in JSON.

Create a new task

- **HTTP request**: `POST api/workflows/{workflowName}/tasks`
- **Description**: Responsible for creating a new task for a particular workflow definition.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier.
- **Request body (required)**: A t definition in JSON.

Create a new event handler

- **HTTP request**: `POST api/workflows/{workflowName}/handlers`
- **Description**: Responsible for creating a new event handler for a particular workflow definition.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier.
- **Request body (required)**: An event handler definition in JSON.

Find all workflows

- **HTTP request**: `GET api/workflows/workflowName`
- **Description**: Responsible for retrieving a list of workflow definitions.

Find a particular workflow

- **HTTP request**: `GET api/workflows`
- **Description**: Responsible for retrieving a particular workflow definition.

Find all tasks

- **HTTP request**: `GET api/workflows/{workflowName}/tasks`
- **Description**: Responsible for retrieving a list of tasks from a particular workflow.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier in which the task is contained.

Find a particular task

- **HTTP request**: `GET api/workflows/{workflowName}/tasks/{taskName}`
- **Description**: Responsible for retrieving a particular task form a given task identifier.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier in which the task is contained.
- **Path variable (required)**: `{taskName}` represents the task identifier in which should be retrieved.

Find all event handlers

- **HTTP request**: `GET api/workflows/{workflowName}/handlers`
- **Description**: Responsible for retrieving a list of tasks from a particular workflow.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier in which the task is contained.

Find a particular event handler

- **HTTP request**: `GET api/workflows/{workflowName}/handlers/{handlerName}`
- **Description**: Responsible for retrieving a particular event handler form a given handler identifier.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier in which the event handler is contained.
- **Path variable (required)**: `{handlerName}` represents the handler identifier in which

should be retrieved.

Update a particular workflow

- **HTTP request**: `PUT api/workflows/{workflowName}`
- **Description**: Responsible for updating a particular workflow definition.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier that must be updated.
- **Request body (required)**: A workflow definition in JSON.

Update a particular task

- **HTTP request**: `PUT api/workflows/{workflowName}/tasks/{taskName}`
- **Description**: Responsible for updating a particular task definition.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier in which the task is contained.
- **Path variable (required)**: `{taskName}` represents the task identifier that must be updated.
- **Request body (required)**: A task definition in JSON.

Update a particular event handler

- **HTTP request**: `PUT api/workflows/{workflowName}/handlers/{handlerName}`
- **Description**: Responsible for updating a particular handler definition.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier in which the event handler is contained.
- **Path variable (required)**: `{handlerName}` represents the event handler identifier that must be updated.
- **Request body (required)**: A event handler definition in JSON.

Delete a particular workflow

- **HTTP request**: `DELETE api/workflows/{workflowName}`
- **Description**: Responsible for deleting a particular workflow definition.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier that must be excluded.

Delete a particular task

- **HTTP request**: `DELETE api/workflows/{workflowName}/tasks/{taskName}`
- **Description**: Responsible for deleting a particular task definition.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier in which the task is contained.
- **Path variable (required)**: `{taskName}` represents the task identifier that must be excluded.

Delete a particular event handler

- **HTTP request**: `DELETE api/workflows/{workflowName}/handlers/{handlerName}`
- **Description**: Responsible for deleting a particular event handler definition.
- **Path variable (required)**: `{workflowName}` represents the workflow identifier in which the event handler is contained.
- **Path variable (required)**: `{handlerName}` represents the event handler identifier that must be excluded.