*Journal of Software Engineering Research and Development, 2020, 8:1,* doi: 10.5753/jserd.2019.457

# Supporting a Hybrid Composition of Microservices. The EUCalipTool Platform

**Pedro Valderas [ PROS Research Center – Universitat Politècnica de València, Spain | pvalderas@pros.upv.es ]**
**Victoria Torres [ PROS Research Center – Universitat Politècnica de València, Spain | vtorres@pros.upv.es ]**
**Vicente Pelechano [ PROS Research Center – Universitat Politècnica de València, Spain | pele@pros.upv.es ]**

**Abstract**
To provide complex and elaborated functionalities, Microservices may cooperate with each other either by following a centralized (orchestration) or decentralized (choreography) approach. It seems that the decentralized nature of microservices makes the choreography approach more appropriate to achieve such cooperation, where lighter solutions based on events and message queues are used. However, orchestration through the usage of a process model facilitates the analysis of the composition when this is modified. To benefit from the goodness of these two approaches, this paper presents a hybrid solution based on the choreography of business process pieces that are obtained from a previously defined description of the complete microservice composition. To support this solution, the EUCalipTool platform is presented.

***Keywords:*** *microservice, composition, choreography, orchestration*

## 1 Introduction

Companies such as Amazon, Airbnb, Twitter, Netflix, Apple, Uber, and many others have shifted towards a microservices architecture intending to be more agile in doing their business. The technology and functionality independence acquired when applying this architecture allows companies to replace, scale, and upgrade their applications easily and very fast (Newman, 2015; Bucchiarone et al., 2018; Shadija et al., 2017). However, to provide their customers with valuable services, developer teams are forced to build microservice compositions due to the small granularity level in which these operate (Dragoni et al, 2017). The definition of such compositions is being made by many organizations programmatically ad-hoc. The major problem when creating compositions in this way is that their complexity grows, making more difficult their visualization, understanding, and maintenance. This complexity has forced many companies to build their solution to compose microservices. Among these solutions, we find Zeebe (the evolution of the Camunda project to orchestrate microservices), Netflix Conductor, ING Baker or Uber Cadence. Apart from Zeebe, the other solutions have been developed by non-software companies to deal with the growing number of microservices handled by each company to develop their business. In general, to achieve microservices compositions we can find two major different approaches, these are choreography and orchestration.

As a motivating example, let us consider a process designed to place orders in a webshop, which is supported by four microservices: *Customers*, *Payment*, *Inventory*, and *Shipment*. The sequence of steps to process an order is the following:

1. A customer places an order in the webshop.
2. The *Customers* microservice checks customer data and logs the request.
3. If the customer is accepted, the *Payment* microservice starts to collect the money. If it is required, payment details can be asked to the customer. In any case, the customer must be informed.
4. As soon as the payment is performed, the *Inventory* microservice starts to fetch the ordered items. If some problem occurs, the customer is informed and the order is canceled.
5. Finally, once the items are fetched correctly, the *Shipping* microservice creates an order of shipment and assigns a driver.

When following the choreography approach (Dragoni et al., 2017; Butzin et al., 2016), the logic of the composition is distributed through microservices, which communicate to each other through an event bus (usually supported by a message queue). Thus, once the client places an order in the webshop (see Fig. 1), an "Order created" event is issued in the queue. The *Customers* microservice, which is listening to this event, reacts to performing its assigned tasks, and a "Customer accepted" event is triggered when the customer data is ok. Then, the *Payment* microservice, which is listening to this event, performs its tasks and generates the event that makes the next microservice in the composition perform the next tasks. And so on.

Let us now suppose that our company wants to provide special treatment to its VIP customers, so they can proceed with the payment by the end of the process. To maintain these low-coupled microservices, this small change would imply the introduction of several changes in different microservices: the *Customers* microservice should generate a different event depending on the type of customer to allow the participation of either the *Payment* microservice (regular customers) or the *Inventory* microservice (VIP customers); in the same way, the *Shipment* microservice should generate a different event to proceed with the payment or on the contrary with the delivery of the order; and the *Payment* microservice should be also modified to allow delivering the order in case of VIP customers. Note how a single change requires the modification of several microservices. The major problem with this approach is that there is not a clear picture of how microservices participate in the process since the composition is hard-coded and distributed along with multiple microservices. Therefore, when engineering decisions need to be taken, it is difficult to analyze the composition's flow.

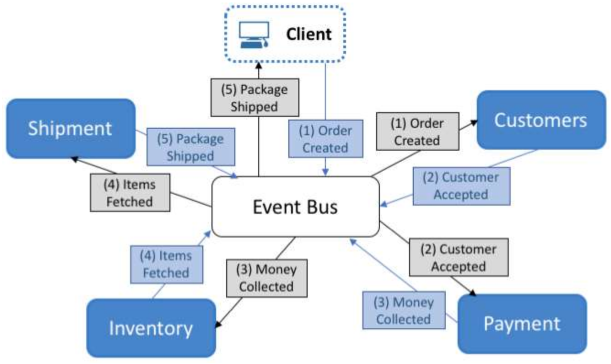


**Figure 1.** Microservice collaboration through Choreography.

On the other hand, when building compositions with the orchestration approach (Singhal et al., 2019; Hamidehkhan, 2019), the logic of the microservice composition is centralized in an orchestrator microservice. One of the possible solutions for this approach is to define compositions as BPMN models and endow the orchestrator microservice with a BPMN engine that is in charge of executing it. The BPMN representation of the motivating example presented above is shown in Fig. 2.

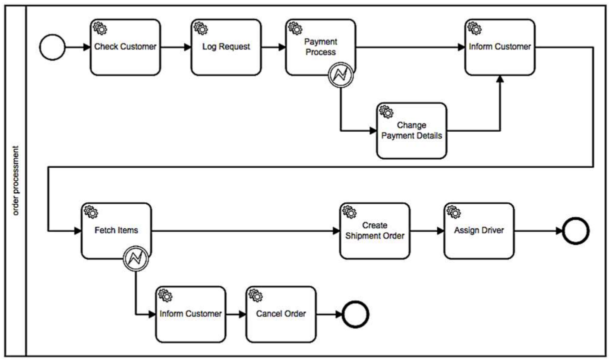


**Figure 2.** BPMN representation of the motivating example.

In this case (see Fig. 3), a client asks the *Orchestrator* microservice to process an order, and this microservice executes the BPMN model that describes the microservice composition that manages customer orders. According to the logic of this composition, the first step the orchestrator does is asking the *Customers* microservice to check the customer data, and then waits for a response from this microservice. Once the response from the *Customer* microservice is received, the *Orchestration* microservice asks the Payment microservice to collect the money and waits for a response. And so on.

With this approach, the logic of the microservice composition is centralized in the orchestrator microservice. If we want to change the composition to support VIP customers, we just need to update the BPMN model accordingly. However, all microservices depends on the orchestrator, reducing the degree of decoupling among them. Also, there are some misconceptions within the microservice community that can make the adoption of this solution difficult: (1) Many times, the task of process modeling is considered as an overhead for a software project; and (2) BPM tools are considered to be heavyweight and to take weeks to set up.

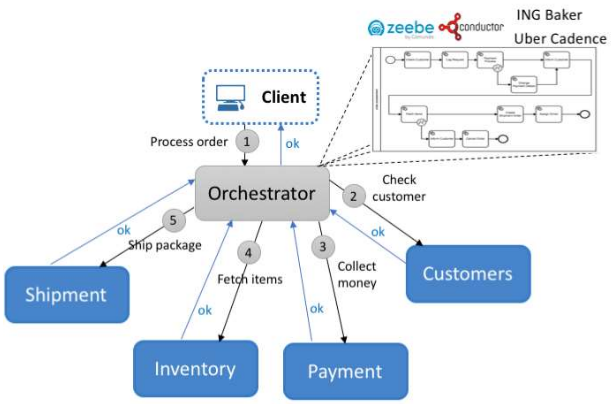


**Figure 3.** Orchestration to support microservice collaboration.

In this paper, we face the challenge of defining a hybrid solution to compose microservices that combine the benefits of both approaches. This solution is based on the following:

1. Developers describe the complete microservice composition by means of a centralized model. This allows having the big picture of the composition, which facilitates the following maintenance and analysis tasks.
2. The centralized model of the composition is split into different pieces whose execution responsibility is delegated to the different participating microservices. Each microservice is in charge of executing its piece and informing the other microservices about its execution. To do so, an event-based orchestration is proposed, which provides a degree of decoupling among microservices higher than the one provided by orchestration solutions.

To support this solution, we present the EUCalipTool platform, which includes the following:

1. An authoring tool to define microservices compositions through a Domain Specific Modeling Language (DSML) that facilitates the modeling activity. This tool has been developed to alleviate the misconceptions of using a process model for composing microservices. Developers can design the whole composition using constructors that are easier to use than business modeling elements. This tool also supports the transformation of descriptions based on our DSML into executable BPMN specifications, and the split of it into pieces.
2. A microservice architecture that facilitates both, the deployment of each BPMN piece into the corresponding microservice, and the distributed execution of the microservice compositions through an event-based choreography. It also supports the maintenance and evolution of the microservice composition.

The remainder of the paper is structured as follows. Section 2 outlines the hybrid solution proposed in this work to achieve microservice compositions. Section 3 presents the architecture designed to support this solution. Section 4 presents the authoring tool proposed to model microservices compositions. Section 5 explains how a microservice composition is transformed into BPMN and split into pieces to be deployed in the proposed microservice architecture. Section 6 analyzes how the evolution of microservice compositions are supported. Section 7 introduces the related work. Finally, Section 8 concludes the paper and provides insights into directions for future work.

## 2 A Hybrid Approach to Compose Microservices

In this section, we present a hybrid approach to achieve microservice compositions. The stages proposed in this approach are the following:

1. Developers define a **centralized description** of the complete microservice composition.
2. The centralized description is split into **BPMN pieces** and these pieces are **distributed** among microservices.
3. The microservice composition is executed through an event-based **choreography of BPMN pieces.**

To illustrate the proposed approach, we make use of the motivating example. First, developers start defining a microservice composition in a centralized model. In the case of the motivation example, developers should create a composition as the one shown in Fig. 2. Note that this microservice composition is defined with BPMN. However, we propose a DSML to facilitate this modeling activity, which is presented in Section 4.

Once developers have described the complete microservice composition, the BPMN model is split into pieces whose execution responsibility is delegated to the different participating microservices. As Fig. 4 shows, the BPMN model of the motivating example is split into four pieces that must be executed by the different microservices.

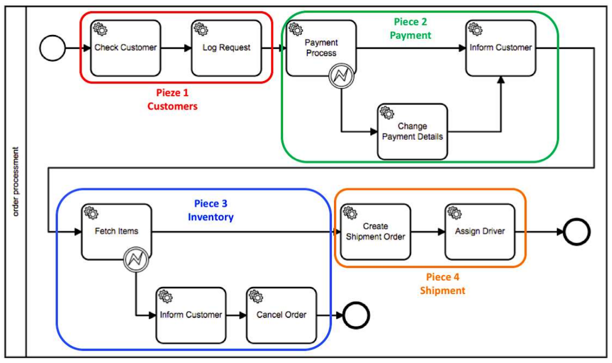


**Figure 4.** Microservice orchestration split into different fragments.

An event-based choreography of BPMN pieces is proposed to support the execution of a microservice composition. In this sense, each microservice is in charge of executing its piece and informing the others about it. Following with the motivating example, once the client places an order in the webshop (see Fig. 5), an "Order Process" event is issued in the message broker. The Customers microservice, which is listening to this event, reacts executing their associated BPMN piece, and the "Piece1_Completed" event is triggered whether the customer data is ok. Then, the Payment microservice, which is listening to this event, performs its BPMN piece and generates the event that makes the next microservice in the composition to execute the next piece. And so on.

Note that current business process management (BPM) tools provide little support to create a business process model and split it into pieces that can be deployed into different microservices. There is also little help to implement the communication mechanisms that are required to coordinate the execution of the different pieces to complete a process. In addition, note that we propose to have two versions of the composition. On the one hand, we have the model of the whole microservice composition. On the other hand, we have a split version that is distributed along with the microservices. Thus, when the microservice composition needs to be evolved due to changes in requirements, both versions must be updated, which implies additional efforts for developers.

Therefore, if we want that developers adopt our proposal we need to provide them with tools that facilitate the tasks of modeling and provide a high degree of automation to deploy composition pieces and configure the execution environment. To achieve this, we present the EUCalipTool platform. The next section introduces the supporting microservice architecture.

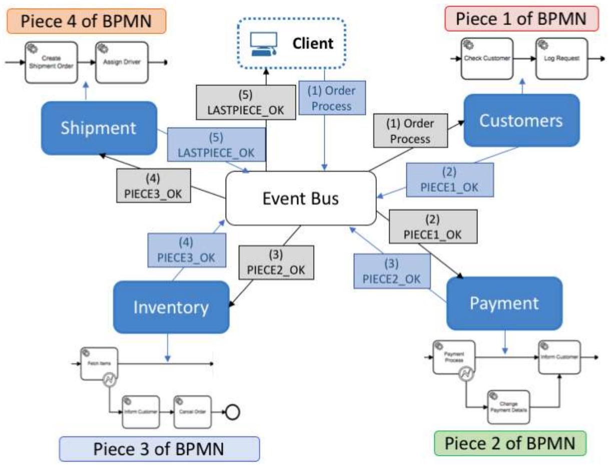


**Figure 5.** Event-based orchestration of BPMN pieces.

## 3 Supporting Microservice Architecture

In a microservice architecture, applications are structured as a collection of loosely coupled services, which implement the business capabilities of a system. Apart from those business microservices, it is usual to find in this type of architecture other microservices that are focused on supporting infrastructure issues. Examples of this type of microservices are the *Service Registry* that gives support to service discovery, containing the network locations of microservice instances; an

*API Gateway* that provides addressability capabilities; an *Authentication Server* that is in charge of controlling the access to the microservices; and a *Configuration Server* that manages microservice configuration on the cloud. In addition, it is also common the use of tools to monitor microservices' status and log their executions, as well as to deploy a message queue to manage asynchronous communication among microservices. Finally, microservices are usually complemented with a client-side load balancer and some library that implements the circuit breaker pattern to support fault tolerance.

Microservices architectures have already been used to build business process modeling and analysis tools (Alpers et al., 2015). In this work, we extend the typical microservice architecture with three main elements (see red-colored blocks in Fig. 6):

1. *EUCalipTool Composer*. It is a microservice endowed with an authoring tool to facilitate the creation of microservices compositions. This microservice also is in charge of transforming the compositions created through the authoring tool into a BPMN executable specification, splitting it into BPMN pieces, and sending them to the *EUCalipTool Server*. In addition, this microservice stores the whole description of the microservice composition created with the authoring tool.
2. *EUCalipTool Server*. It is a microservice that plays the role of gateway among business microservices and the *EUCalipTool Composer*. It is responsible for the following tasks:
   a. Receiving the split BPMN processes sent by the *EUCalipTool Composer*, registering them into a process repository, and distributing the pieces among the different microservices.
   b. Launching the execution of each process by triggering the first BPMN piece and delegating the responsibility of continuing the process to the corresponding microservice. To achieve this, a message queue is used.
   c. Providing the *EUCalipTool Composer* with the list of available microservices and their operations. To achieve this, microservices must be registered into this server using the *EUCalipTool Client*.
3. *EUCalipTool Client*. It is a client library that endows each microservice with: (1) a lightweight Activiti[1] BPMN engine and (2) a microservice composition authoring tool. The BPMN engine is included to support the execution of BPMN pieces. The authoring tool is included to support the evolution of these pieces by the developers of each microservice. This library is also in charge of automatically registering microservice's operations into the *EUCalipTool Server*.

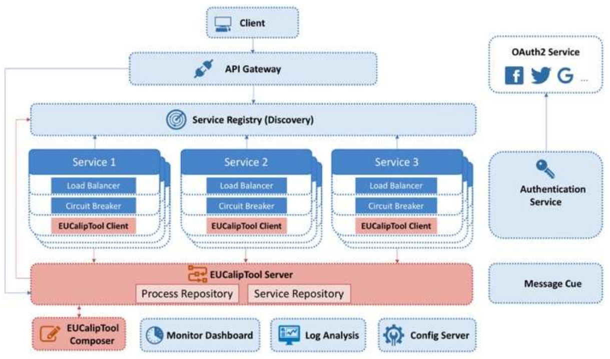


**Figure 6.** Microservice orchestration split into different fragments.

To satisfy the responsibilities associated with each architectural element, they must interact with each other. This interaction is done through the HTTP protocol. Thus, each architectural element is in charge of publishing the required HTTP end-points. For instance, the EUCalipTool Client library is in charge of publishing an HTTP end-point to allow the EUCalipTool Server to send the BPMN pieces to each microservice. In the same way, the EUCalipTool Server must publish an HTTP end-point to allows the EUCalipTool Client library to register the operations of each microservice.

## 3.1 Supporting Technology

One of the most important supporters of the microservice architecture is Netflix. This video streaming company has developed its software infrastructure by using microservices and has published all its supporting tools as open source. One of the main characteristics of these tools is their ease of use. These tools are based on the Spring Boot[2] framework and are distributed as Java libraries[3]. They propose the use of simple annotations and configuration files to develop and deploy the different components of the architecture. For instance, to build a *Service Registry* to support microservice discovery it is enough to create a Spring Boot Java class and annotate it with the annotation @EnableEurekaServer. Then, you just need to define some parameters in a configuration file and the "magic" is done. You have a functional Service Registry.

We want to follow the same strategy to facilitate the use of the EUCalipTool infrastructure in a real microservice architecture. Thus, we have created three Java packages that encapsulate the functionality of the three proposed architectural elements and they are complemented with the following three annotations:

- `@EUCalipToolComposer`
- `@EUCalipToolServer`
- `@EUCalipToolClient`

Thus, to create these microservices, developers just need to create a Spring Boot Java class, use these annotations and, in some cases, define some configuration parameters.

[1] https://www.activiti.org/
[2] https://projects.spring.io/spring-boot/
[3] https://netflix.github.io/

For instance, to create an *EUCalipTool Sever* microservice developers just need to import the corresponding Java libraries, and create a Java Class as follows:

```
@EUCalipToolServer
public class Server {
    public static void main(String[] args) {
      SpringApplication.run(Server.class,
args);
    }
}
```

The `SpringApplication` class is a Spring utility that creates a Java application with an embed Tomcat. When the above code is executed, we intercept the run method and search for our annotations by using reflection capabilities. When the `@EUCalipToolServer` is found, we deploy the functionality of this component into the embed Tomcat. We also create an HTTP Controller that publishes the required end-points to interact with the rest of the architectural elements. The configuration that is required for this component is the end-points of the components that need to interact with. In particular, the API Gateway, the Service Registry, and the Message Cue. This configuration is done through a YML file.

By using and configuring the other two annotations we achieve the following:

- `@EUCalipToolComposer`. It creates a Spring application with the EUCalipTool Composer deployed into the embed Tomcat. This annotation needs a config file that indicates the end-points of the EUCalipTool Server that (1) provides the list of microservices and their operations, and (2) allows sending a split composition. It also creates an HTTP Controller that publishes the end-points required to interact with the EUCalipTool Server.
- `@EUCalipToolClient`. It transforms a microservice into a EUCalipTool client. To do so, it includes a lightweight version of the Activiti engine to execute BPMN pieces. It also includes a web graphical editor deployed into the embed Tomcat. Also, it creates an HTTP Controller that publishes end-points to both receiving BPMN pieces and subscribing the microservices to choreography events. This annotation needs a config file that indicates the end-points of the EUCalipTool Server in order to register microservice's operations and send BPMN pieces when are modified.

## 4 Specifying Microservice Compositions

The EUCalipTool Composer includes a web-based authoring tool that proposes a Domain Specific Modeling Language (DSML) to facilitate the modeling of microservice composition. It is based on a previous work that focuses on helping end-users to compose services by using a visual interface from a mobile device (Valderas et al., 2017).

Next, we present the abstract syntax of the DSML (i.e. the conceptual elements) and the concrete syntax (i.e. the graphical components that define the web interface).

### 4.1 DSML Abstract Syntax

The abstract syntax of the DSML supported by the web graphical editor is based on the *Change patterns* (Weber et al., 2008) developed within the context of the process of process modeling. Change patterns are high-level abstractions aimed at achieving flexible and easy adaptations of a business process. These abstractions are defined in terms of high-level change operations (e.g., the creation of a parallel branch) which are based on the execution of a set of change primitives (e.g., add/delete activity). As opposed to change primitives, change pattern implementations typically guarantee model correctness after each transformation (Casati, 1998) by associating pre/post conditions with high-level change operations. Usually, process modeling environments supporting the correctness-by-construction principle (e.g., Dadam et al., 2009) just provide process modelers with those change patterns that transform a sound process model into another sound one. For this purpose, structural restrictions on process models (e.g., block structuredness) are imposed. In addition, correct usage of change patterns allows speeding up the creation of the composition. Some change patterns are (Weber et al., 2008): Insert Process Fragment, Embed Process Fragment in Loop, Embed Process Fragment in Conditional Branch, etc.

Inspired by the concept of fragment introduced by change patterns, the abstract syntax of the DSML proposed to compose microservices is shown in Fig. 7.

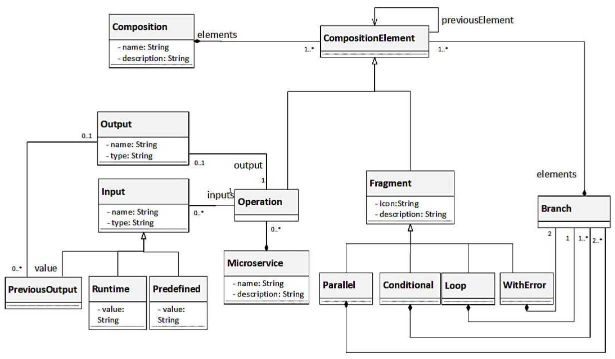


**Figure 7.** Domain Specific Language designed for EUCalipTool.

A microservice *Composition* is made up of *CompositionElement*s of two types which are *Operations* (of a *MicroService*) and *Fragments*. Each operation has some *Inputs* and one *Output*. Inputs are classified into three types depending on the source from which their value is obtained. This source can be the output of another operation; it can be obtained at runtime; or can be defined at design time. In the next subsection, this issue is explained with some examples. Regarding *Fragments*, there are four types: *Parallel*, which has two or more *Branches* of elements that must be executed in parallel; *Conditional*, which has one or more branches of elements that must be executed when a condition is satisfied; *Loop*, which has a branch of elements that must be executed while a condition is satisfied; and *WithError*, which has two branches of elements, a major one that is executed by default, and a compensation one the is executed if some errors occur with some of the major branch's operations. The *previousElement* relationship between *CompositionElement*s allows establishing the sequence order between operations and fragments.

To better understand the concepts of this metamodel, Fig. 8 illustrates them in a process that is composed of a sequence of four operations followed by a parallel fragment. In turn, this latter parallel fragment is made up of a conditional fragment and two operations that are executed in parallel to it.

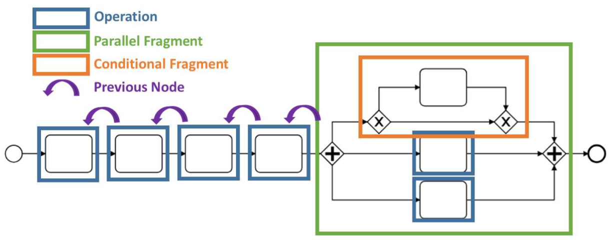


**Figure 8.** DSL Concepts applied in an example.

## 4.2 DSML Concrete Syntax

To create a composition of microservices we have defined a web interface based on the "adding element" metaphor where microservice developers just need to add a set of operations or fragments to a composition.

To exemplify this interface, Fig. 9 shows some of the screens needed to define the *Payment piece* marked in green in Fig. 4. Fig. 9A shows the composition after adding the *checkCustomer* and *logRequest* operations of the microservice *Customers*. To add more elements, designers just need to click on the "+" symbol. The type of elements that can be added to a composition are single operations and fragments (note that there are two tabs in Fig. 9B). Fig. 9B shows a list of fragments that are ready to be used in the current composition. In this case, the designer is selecting a *With Error* fragment. As a result, a fragment of this type is included after the existing operations (see Fig. 9C). Here, the designer should specify two things, the major branch of operations to perform and the compensation branch of operations in case the major branch fails. In this case, the designer selects the *paymentProcess* operation offered by the *Payment* microservice to be included in the major branch (see Fig. 9D). This is offered as a single operation from the available catalog. This list shows the microservice operations that the *EUCalipTool Server* sends to the *EUCalipTool Composer*. These operations are automatically registered into the *EUCalipTool Server* by the *EUCalipTool Client* library that is installed in each microservice.

The selection of this single operation results in the screen shown in Fig. 9E. At this point, the designer still has to specify what to do when the major branch fails. This can be specified by selecting the tab labeled with the warning icon, and proceeding similarly to the definition of the major branch. In this case, the designer selects the operation *ChangePaymentDetails*. With this action, the second element of the composition is already completed (see Fig. 9F). At this point, the designer should continue by selecting the most appropriate operations or fragments until the composition is completely defined.

Once microservice composition's flow is described, developers must define the inputs that some microservice operations require to be properly executed. To facilitate this, we provide a graphical component (see Fig. 10) that allows: (1) linking an input with any compatible previous output, (2) indicating that the input value should be obtained at runtime; or (3) defining an input value at design time. For instance, let us consider that the operation *cancelOrde*r, which must be executed by the *Inventory* microservice in case of error, needs two inputs: the *customer* ID, which is a String, and the *order* number, which is an Integer. Let us consider also that all previous microservice operations generate a string value as output. Fig. 10B shows the options that are available for the customer input. In this case, it can be associated to any previous operation since their data types are compatible, and also can be defined as an input to be obtained *At runtime* or an input that is associated to a *Predefined Value* (defined at design time in this screen). Fig. 10 shows the options available for the order input. In this case, none of the previous operations is compatible so they are not available to be associated with this input.

If a developer selects the option *Predefined Value* for a microservice's input, an input component is shown in order to allow the developer to introduce the value associated with the microservice's input at design time. Regarding the option of defining an input to be obtained at runtime it implies that the values must be obtained when executing the microservice composition, and from a data source different from the own operations included in the composition. Currently, we are considering that the data source is the client that launches the microservice composition (see Fig. 5). Thus, any time a microservice needs to execute an operation that has some input to be obtained at runtime, the corresponding BPMN piece generates an event in order to ask the client for this data. In further work, we want to consider other data sources such as the results of other microservice compositions or some physical devices in the context of the Internet of Things.

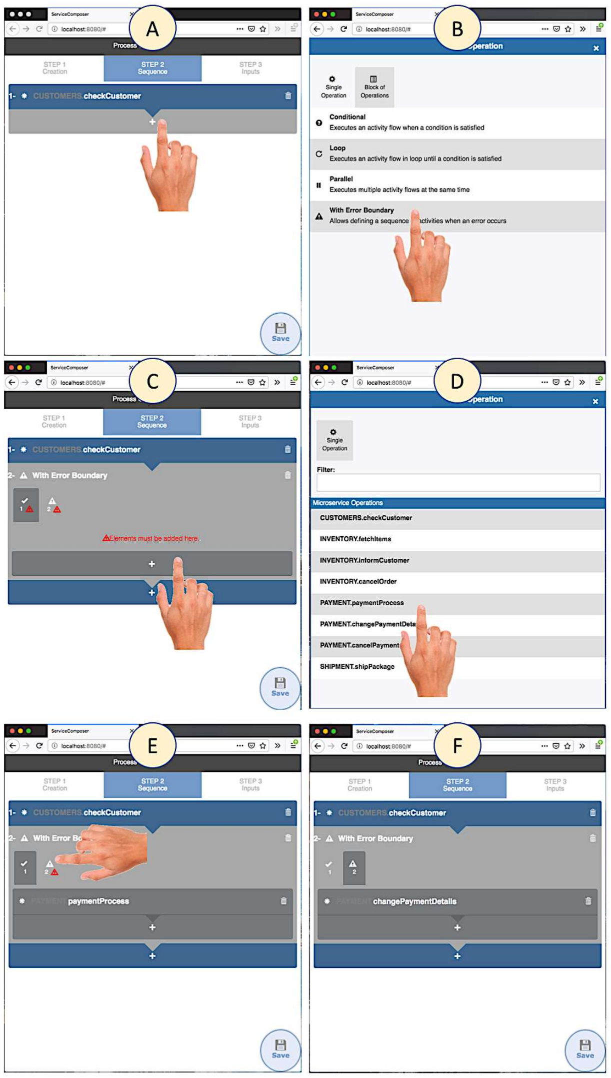
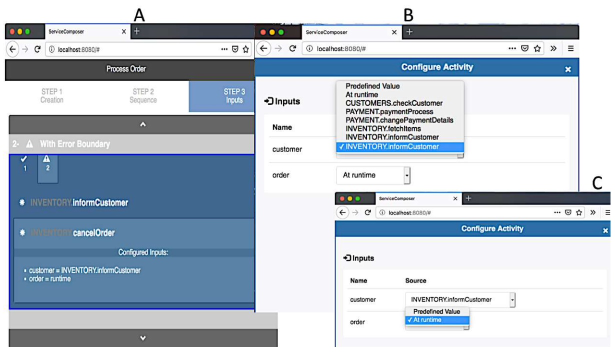


**Figure 9.** Example of the DSML Concrete Syntax to create microservice compositions.



Configured Inputs:
- customer = INVENTORY.informCustomer
- order = runtime

**Figure 10.** Configuration of microservice operation's inputs.

## 5 Supporting the execution of Split BPMN processes

Once a microservice composition is defined with the *EUCalipTool Composer* three main stages are followed to distribute the responsibility of the process execution:

(1) **Generation**. The composition is transformed into a set of BPMN pieces.

(2) **Distribution**. BPMN pieces are sent to the *EUCalipTool Server* which registers the process and deploys the pieces into the corresponding microservices.

(3) **Choreography**. Each microservice participates in the composition through an event-based orchestration.

### 5.1 Generation of BPMN pieces

The *EUCalipTool Composer* analyzes each process defined with the DSML and creates groups of actions according to the microservices that support them. Each of these groups will be transformed into a BPMN piece. For instance, let us consider the composition presented in the motivation example (cf. Fig. 11). In this case, the first two operations must be executed by the *customer* microservice and, therefore, they constitute the first piece. The second piece is defined by the third and fourth elements of the composition (a *With Error Boundary* block and a single operation), which both must be executed by the *payment* microservice. The third piece is defined from the operations that the *inventory* microservice must execute, i.e. fetch the items and the composition actions in case of error. Finally, the fourth piece is made up of the two last operations that must perform the s*hipment* microservice.

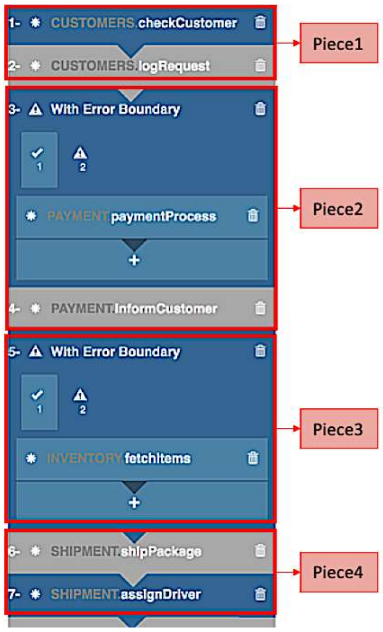


**Figure 11.** Identification of BPMN pieces.

For each BPMN piece, the *EUCalipTool Composer* generates a specification with the BPMN tasks to be performed as well as additional tasks to trigger the events that must manage the orchestration. For instance, let us consider the operations that must perform the microservice *Inventory* (the third piece of BPMN). This microservice must fetch the items of the order and, in case of error, inform the user and cancel the order. Fig. 12 shows the definition built with the *EUCalipTool Composer* and the generated BPMN process model. As we can see, two additional BPMN tasks are in charge of 1) triggering an Ok event in case there is no error, and 2) triggering a fail event if some problem occurs. These tasks are preconfigured to publish the event in a message queue.

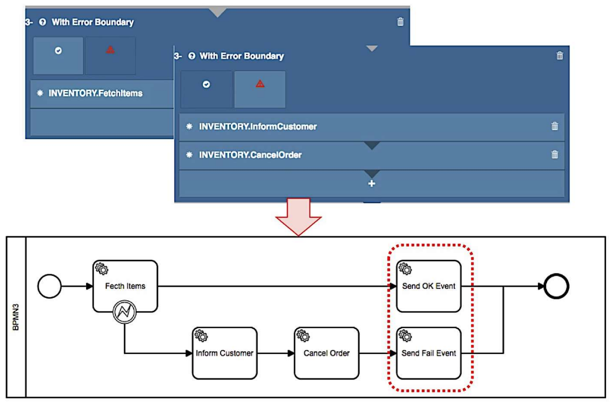


**Figure 12.** Generated piece of BPMN.

The EUCalipTool Composer internally manages each composition in JSON format. To transform JSON descriptions into BPMN (which is based on XML) it uses Java parsers of JSON and XML. The JSON description is parsed into a structure of Java objects that are maintained in memory. Next, this structure is analyzed in order to generate a BPMN specification by using the XML parser. In particular, we generate BPMN specifications that will be executed in the Activiti engine, i.e. the engine included in the microservice by the *EUCAlipTool Client* library.

## 5.2 Distribution of BPMN pieces

Once the set of BPMN pieces has been generated, the *EUCalipTool Composer* sends them to the *EUCalipTool Server*. To do so, the latter publishes an HTTP end-point that accepts this data through POST connections.

When the *EUCalipTool Server* receives a split composition, it performs the following actions (see Fig. 13):

(1) It registers the composition into its repository and creates an HTTP end-point to launch it.
(2) It deploys each piece of BPMN into the corresponding microservice.
(3) It defines an event to launch the first piece of BPMN and configures the first microservice to listening to it.
(4) For each event generated by a piece of BPMN, it configures the microservice that must execute the next piece to listen to this event.

Note that the *EUCalipTool Server* must interact with the microservices to deploy each piece of BPMN as well as to configure the microservice to listen to specific events. This can be done using a set of HTTP endpoints that each microservice has available when including the *EUCalipTool Client*.

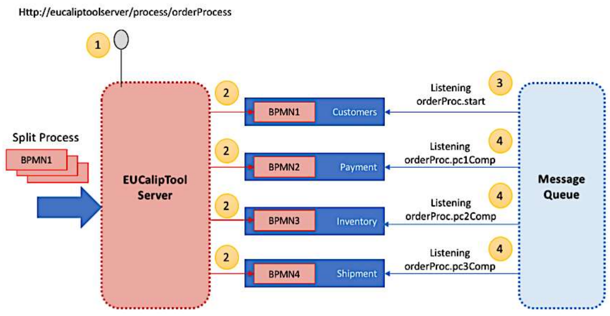

Http://eucaliptoolserver/process/orderProcess
1
Split Process
BPMN1
EUCalipTool Server
2
BPMN1 Customers
BPMN2 Payment
BPMN3 Inventory
BPMN4 Shipment
Listening orderProc.start
3
Listening orderProc.pc1Comp
4
Listening orderProc.pc2Comp
4
Listening orderProc.pc3Comp
4
Message Queue

**Figure 13.** Actions done by EUCalipTool Server.

## 5.3 Orchestration of BPMN pieces

The orchestration of the BPMN pieces deployed in microservices is done as follows (see Fig. 14):

(1) A client accesses the end-point published by the *EUCalipTool Server*.
(2) The *EUCalipTool Server* launches the start event for this process.
(3) The microservice that is listening to this event executes the first piece of BPMN. This execution finishes by triggering an event that indicates that the execution of the first BPMN piece is completed.
(4) The microservice that is listening to the event that indicates the execution of the first BPMN piece launches its BPMN piece (the second one) and when executed, it generates another event that indicates that the execution of the second BPMN piece is completed.
(5) The microservice that is waiting for the event that indicates the execution of the second BPMN piece does the same actions as the previous one: launches its corresponding BPMN piece and generates an event that indicates its execution.
(6) And so on until the process is completed.

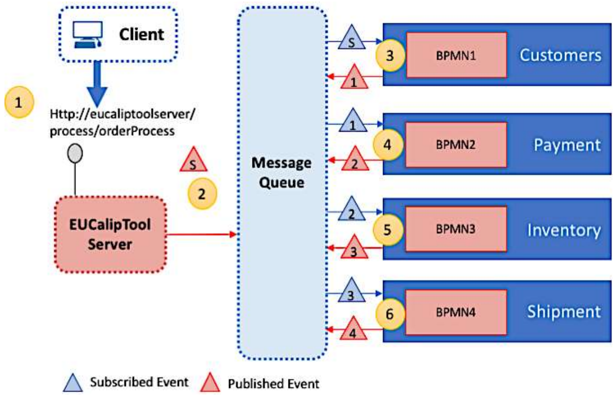


**Figure 14.** Event-based orchestration of a split BPMN process.

## 6 Supporting the evolution of microservice compositions

Following the proposed hybrid approach, we have two descriptions of a microservice composition. On the one hand, we have the whole picture of the composition that is stored by the *EUCalipTool Composer*. This centralized description helps developers to analyze the whole composition to take engineering decisions. On the other hand, we have the split version of the composition that is distributed through the different microservices. This split description provides a high degree of decoupling among microservices when the composition is executed through an event-based choreography.

One of the most important challenges to be faced within this context is the evolution of the microservice composition and the synchronization of both descriptions. Our main goal is to propose a solution that provides developers with a high degree of flexibility to perform changes. So these can be done either at the centralized composition, i.e., at the whole composition, or at the microservice level, i.e., at the pieces deployed in each microservice.

To achieve this, as introduced in Section 3, the following mechanisms are provided by the proposed three architectural elements:

- The *EUCalipTool Client* library includes a web editor like the one shown in Section 6 where developers can independently evolve their composition pieces.
- The *EUCalipTool Server* publishes an HTTP end-point to receive modified composition pieces from microservices to send them to the *EUCalipTool Composer*.
- The *EUCalipTool Composer* publishes an HTTP endpoint to receive modified composition pieces from the *EUCalipTool Server* to update the whole version of the composition.

Thus, the evolution of a microservice composition can be done in two ways:

1 Developers update the whole description of the composition from the *EUCalipTool Composer* microservice (see Fig. 15A). In this case:
  1.1 The *EUCalipTool Composer* microservice generates the corresponding BPMN pieces and sends those pieces that have been changed to the *EUCalipTool Server*.
  1.2 The *EUCalipTool Server* microservice distributes the pieces among the corresponding business microservices.
  1.3 Microservices that receive a new version of a piece, replace the old version by the new one.

2 Developers change a composition piece from a business microservice (see Fig. 15B). In this case:
  2.1 The microservice sends the new version of the piece to the *EUCalipTool Server*.
  2.2 The EUCalipTool Server sends the received piece to the *EUCalipTool Composer*.
  2.3 The *EUCalipTool Composer* updates the whole description with the changes that introduce the modified piece.

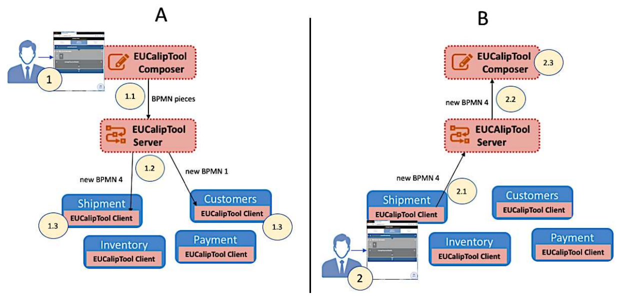


**Figure 15.** Evolution of a microservice composition.

To update the whole composition when an updated BPMN piece is received, the *EUCalipTool Composer* applies the transformation inverse to the one used to generate the BPMN pieces and obtains a JSON representation of the piece. This JSON representation is based on the DSML presented above and the *EUCalipTool Composer* just needs to replace the elements of the whole description that correspond with the updated piece. Note that updating the whole description of the microservice composition is easy since pieces are composed of operations and fragments that are added to a container. There are no connections with previous or further elements that need to be managed like can happen with a BPMN model. In order to better understand this aspect Fig. 16 illustrates how the composition of the motivating example is updated with a new piece 2.

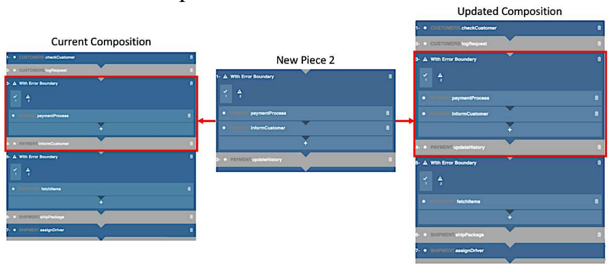


**Figure 16.** Example of composition update by replacing a piece.

## 7 Evaluation

This section presents the experiment that we have conducted to show the efficiency of our proposal in the development and evolution of microservice compositions. This experiment aimed to compare the efficiency measurement obtained by a development based on EUCalipTool with the measurement obtained by an ad-hoc implementation of an event-based choreography. This ad-hoc implementation was done by using the technology provided by Spring and Netflix. To support the exchange of messages among microservices, a RabbitMQ message broker was used in both cases.

To do the experiment, we followed the guidelines presented by Kitchenham et al. (1995) and Wohlin et al. (2012). According to these guidelines, we have divided the experiment into three main phases: scoping, planning, operation and analysis, and interpretation

### 7.1 Scope

The scope of an experiment is set by defining its goal. To do so, we have used the template proposed by Basili et al.

(1988). The goal of our experiment is characterized as follows:

<u>Analyze:</u> Our approach based on EUCalipTool
<u>For the purpose of</u>: evaluating the impact of our approach compared to ad-hoc development
<u>With respect to</u>: efficiency
<u>From the point of view of</u>: microservice developers
<u>In the context of</u>: researchers in software engineering composing microservices

## 7.2 Experimental Design

In the planification activity, we must formalize the hypotheses, determine the dependent and independent variables, describe the context of the experiment and the instrumentation used, and consider the threats of validity we can expect.

**Hypothesis.** The hypotheses defined for the experiment were the following:

- Null hypothesis 1, H10. The efficiency of the EUCalipTool approach for developing and evolving microservice compositions is the same as an ad-hoc development.
- Alternative hypothesis 1, H11. The efficiency of the EUCalipTool approach for developing and evolving microservice compositions is greater than an ad-hoc development.

**Identification of variables.** We identified two types of variables:

- <u>Dependent variables</u>: Variables that correspond to the outcomes of the experiment. In this work, the efficiency in composing microservices was the target of the study, which was measured in terms of the following software quality factors: *development time* and *evolving time*.
- <u>Independent variables</u>: Variables that affect the dependent variables. The development method was identified as a factor that affects the dependent variable. This variable had two alternatives: (1) EUCalipTool approach and (2) an ad-hoc implementation.

**Context.** The context of the experiment was the following:

- <u>Experimental subjects</u>. Ten subjects participated in the experiment, all of the researchers in software engineering. Their ages ranged between 28 and 45 years old. The subjects had an extensive background in Java programming and modeling tools; however, they did not have experience in the use of EUCalipTool. Only 3 of them have experience in using the Spring Framework and message queues, and 4 of them have previously worked with BPMN.
- <u>Objects of study.</u> The experiment was conducted using a case study similar to the motivating example used throughout the paper, i.e. the microservice composition to manage a purchase order in a webshop (see Section 1).

**Instrumentation**. The instruments that were used to carry out the experiment were:

- <u>A demographic questionnaire</u>: a set of questions to know the level of the users' experience in Java/Spring programming, modeling tools, and BPMN.
- <u>Work description</u>: the description of the work that the subjects should carry out in the experiment by using EUCalipTool and the ad-hoc solution. This work description explained two activities: (1) the development of the microservice composition to support purchase orders, and (2) the modification of this composition to support new requirements.
- <u>A form</u>: a form was defined to capture the start and completion times of the proposed work. For each task that was proposed in the experiment, participants had to annotate the starting and completion times by using the clock of the computer. If some interruptions occur while performing the work, subjects wrote down the times every time they started and stopped carrying out the activity; thus, the total time was derived using these start and completion times. Finally, additional space was left after the completion time of the work for additional comments about the subjects about the performed activity.

**Threats of Validity.** Our experiment was threatened by the random heterogeneity of subjects. This threat appears when some users within a user group have more experience than others. This threat was minimized with a demographic questionnaire that allowed us to evaluate the knowledge and experience of each participant beforehand. This questionnaire revealed that all the users had experience in Java programming and modeling techniques. Some of them had experience in the use of technologies related to the implementation of choreographies, while others did not. This problem could affect the evaluation of the development with an ad-hoc solution since this type of development requires these technologies. Some participants had experience in BPMN which could affect the evaluation of the development based on EUCalipTool since it is based on some abstractions of BPMN. To minimize this threat, all subjects participated in training sessions about both choreography implementation technologies and EUCalipTool.

In addition, to minimize the effect of the order in which the subjects applied the approaches, the order was assigned randomly to each subject. However, in order to have a balanced design, the same number of subjects was assigned to start with each approach. To do so, the ten participants were aleatorily divided into two groups, and each group was initially assigned to a development type. Then, each group changed of development type to do again the same tasks. In this way, we minimized the threat of learning from previous experience.

Finally, our experiment was threatened by the reliability of measures threat: objective measures, that can be repeated with the same outcome, are more reliable than subjective measures. In this experiment, the precision of the measures may have been affected since the activity completion time was measured manually by users using the computer clock. To reduce this threat, we observed subjects while they were performing different tasks to guarantee their exclusive dedication in the activities and supervise the times that they wrote down.

## 7.3 Execution

We followed a within-subjects design where all subjects were exposed to every treatment/approach (EUCalipTool solution and ad-hoc solution). The main advantage of this design was that it allowed statistical inference to be made with fewer subjects, making the evaluation much more streamlined and less resource-heavy (Wohlin et al., 2012).

To perform the experiment, we arranged a workshop of three days with two sessions per day (see Table 1).

**Table 1.** Sessions of the experiment

| | **Session 1** | **Session 2** |
|---|---|---|
| **Day 1** | Duration: 4h<br>All participants: Training in choreography implementation | Duration: 4h<br>All participants: Training in EUCalipTool |
| **Day 2** | Duration: 5h<br>Group A: Development of a microservice composition with an ad-hoc solution<br>Group B: Development of a microservice composition with EUCalipTool | Duration: 3h<br>Group A: Evolution of a microservice composition with an ad-hoc solution<br>Group B: Evolution of a microservice composition with EUCalipTool |
| **Day 3** | Duration: 5h<br>Group A: Development of a microservice composition with EUCalipTool<br>Group B: Development of a microservice composition with an ad-hoc solution | Duration: 3h<br>Group A: Evolution of a microservice composition with EUCalipTool<br>Group B: Evolution of a microservice composition with an ad-hoc solution |

During the first day, we had two sessions of 4 hours in which participants were proposed to fill in a demographic questionnaire to capture participants' background and were trained in choreography technologies and EUCalipTool. In particular:

- Regarding choreography technologies, we provided the subjects with the necessary tutorials and tools to learn the basics of the Spring and Netflix technologies needed to develop the case study. We also made an introduction to message queues and RabbitMQ. The subjects also participated in the implementation of some guided examples to gain experience with the technologies.
- Regarding EUCalipTool, we provided the subjects with a tutorial where the web authoring tool included in the EUCalipTool Composer was explained. The subjects also worked with some examples to gain experience with the DSML of this tool. We also explained the proposed architecture and how the proposed EUCalipTool architectural elements interact among them and need to be configured.

During the second and third days, participants were divided aleatorily into two groups, A and B, and two sessions of five and three hours respectively were proposed for each day. We did the same experiment in both days. In one day, group A used an ad-hoc solution to develop and evolve a microservice composition while group B used EUCalipTool. The second day groups changed the development methods.

The tasks designed for the experiment were initiated with a short presentation in which general information and instructions were given. Afterward, the work description and the form were given to the subjects and they started to develop and evolve the microservice composition following the development method (EUCalipTool and ad-hoc) that was indicated for each group. The microservice composition that participants had to develop was described in a textual way. After performing this work, participants filled in a form to capture the development times. Once the subjects developed the composition, they started to modify it to evaluate the evolution. For these activities, they also filled in the form to capture the time taken to evolve the composition.

To properly perform this work, we previously developed the microservice architecture required to support the case study. To do so, we used Netflix's technology. The *EUCalipTool Composer* and the *EUCalipTool Server* microservices were also created, and every business microservice was defined as a *EUCalipTool client*.

In a more detailed way, the activities carried out with each development approach were the following:

- Ad-hoc development: From the case study description, they started the implementation of the microservice composition for the management of purchase orders. Generally, they identified the operations that each microservice should perform, and defined for them both, a starting event and an end event. Once this data was clear, they updated each microservice with the classes required to connect to RabbitMQ and listen at the starting event to launch the operations corresponding to each microservice. To execute these operations, they implemented some classes that call the corresponding methods. These classes also were in charge of launching the ending event. Once they modified each microservice and achieved the compilation of the code, they spent some time testing the composition and detecting code errors. Finally, we provided a set of requirement changes for the composition to evaluate the evolution. In particular, we proposed them to support VIP customers in such a way it was introduced in Section 1. In this activity, the participants changed the code of the involved microservices to support the new requirements. Then, the participants tested the new composition and corrected the errors.
- EUCalipTool-based development. Following this approach, the participants first designed the microservice composition with the EUCalipTool Composer according to the case study description. Then, they asked the EUCalipTool Composer to deploy the composition. Afterward, they spent some time testing the composition and detecting errors in the composition design. Finally, we asked participants to support the same new requirements as explained in the previous activity. In this case, the participants changed the composition done with the EUCalipTool Composer and deployed it again. Then,

the participants tested the new composition and corrected the errors.

## 7.4 Analysis of results

In this subsection, we analyze and compare the usefulness of both approaches based on the time used for the development and evolution of a microservice composition. The results have been studied based on time mean comparison and the standard deviation. Table 2 presents the descriptive statistics for each of the studied quality factors.

**Table 2.** Descriptive statistics for each quality factor.

| Quality factor | Dev. method | Mean (hours) | Num. of Subjects | Std.dev. (hours) |
|---|---|---|---|---|
| Develop. time | Ad-hoc | 4.38 | 10 | 0.52 |
| | EUCalipTool | 1.15 | 10 | 0.44 |
| Evolution time | Ad-hoc | 1.55 | 10 | 0.69 |
| | EUCalipTool | 0.29 | 10 | 0.05 |

Next, we provide further analysis of the results for each measured software quality factor:

- Development time. The development time following the ad-hoc approach differed according to the subject implementation experience, ranging from 3.25 hours (the most experienced subject) to 5. Following the EUCalipTool approach, the development activity ranged from 75 min to 2.10 hours. The difference between the two approaches was high since developing the microservice composition in an ad-hoc way was more complex and difficult for the participants since they had to implement all the composition logic manually as well as all the code required to connect with RabbitMQ to participate in the event-based choreography. The EUCalipTool approach allowed participants to focus on the required requirements instead of solving technological problems. Note that by following this approach, none of the participants had to implement anything to manage the invocation of operations neither the events required to participate in the choreography. Regarding the standard deviation, it was low for both development approaches (see Table 1) indicating that development times tended to be close for each development approach.
- Evolution time. Concerning the ad-hoc development, this activity took subjects from 1.10 to 2.3 hours since they had to identify the microservices that must be updated, and modify the corresponding code. Changing the EUCalipTool description of the microservices composition took less than 30 min. for all the subjects (very low standard deviation obtained). This is because evolving the microservice composition to fit the new requirements was as easy as modifying the whole description with the web authoring tool. In this case, participants focused again only on requirements. They did not need to identify microservices and hardcoded changes.

With the EUCalipTool approach, the subjects took, on average, 1.44 hours to develop the case study, whereas with an ad-hoc implementation the subjects took 5.93 hours. Therefore, the process for automating and evolving microservice compositions is more efficient using the EUCalipTool approach than using an ad-hoc solution.

In order to verify whether we can accept the null hypothesis, we performed a statistical study called paired T-test using the IBM SPSS Statistics V201 at a confidence level of 95% ($\alpha = 0.05$). This test is a statistical procedure that is used to make a paired comparison of two sample means, i.e., to see if the means of these two samples differ from one another. For our study, this test examines the difference in mean times for every subject with the different approaches to test whether the means of an ad-hoc development and the EUCalipTool approach are equal. When the critical level (the significance) is higher than 0.05, we can accept the null hypothesis because the means are not statistically significantly different. For our experiment, the significance of the paired T-test for the total time means is 0.000 (calculated using the IBM SPSS Statistics), which means that we can reject the null hypothesis H10 (the efficiency of the EUCalipTool approach for developing and evolving microservice compositions is the same as an ad-hoc development). Based on this test, we have given strong evidence that the kind of development influences the usefulness. Specifically, the efficiency using the EUCalipTool approach is significantly better than using an ad-hoc solution, i.e., the mean values for all the measures are lower when using the EUCalipTool approach; thus, the alternative hypothesis H11 is fulfilled: The efficiency of the EUCalipTool approach for developing and evolving microservice compositions is greater than an ad-hoc development.

## 7.5 Conclusions

The above-presented experiment evaluated our approach to develop and evolve microservice compositions concerning ad-hoc solutions based on choreographies. We have validated that our approach is more efficient than ad-hoc solutions and have confirmed the expected benefits suggested in the introduction. On the one hand, having the big picture of the composition has facilitated its analysis to support its evolution when requirements changed. On the other hand, the visual editor of EUCalipTool, as well as the supporting infrastructure to manage event-based communication, have significantly facilitated the definition and execution of choreographed microservice compositions. Note that we have evaluated ad-hoc solutions based on choreographies since the decentralized nature of microservices seems to make choreographies more appropriate to define microservices compositions (Dragoni et al., 2017; Butzin et al., 2016). A similar experiment focusing on orchestration will be considered as further work.

[1] Statistical analyses using spss, http://www.ats.ucla.edu/stat/spss/whatstat/whatstat.htm#1sampt

## 8 Related work

Rajasekar et al. (2012) presented the integrated Rule Oriented Data System (iRODS) to orchestrate microservices within data-intensive distributed systems. A microservice choreography is defined as a set of textual event-condition-action (ECA) rules. Each rule defines the data management actions that a microservice must execute. These actions generate events within the system that trigger the rules associated with other microservices. The authors also proposed the use of recovery microservices to maintain transactional properties. The main drawback of this work is that the logic of the process is distributed along with the different rules that each microservice implements, making the maintenance and evolution difficult to perform.

Yahia et al. (2016) introduce Medley, an event-driven lightweight platform for microservice orchestration. They propose a textual domain-specific language (DSL) for describing orchestrations using high-level constructs and domain-specific semantics. These descriptions are compiled into low-level code run on top of an event-driven process-based and lightweight platform. The main drawback of this approach is that developers need to explicitly manage service orchestration issues at the modeling level. Our solution allows developers to focus only on modeling business requirements. Also, a choreography solution is proposed to obtain a major level of independence among microservices.

Kouchaksaraei et al. (2018) present Pishahang, a framework for jointly managing and orchestrating cloud-based microservices. This framework introduces tools to easily integrate SONATA (Dräxler et al., 2017), an orchestration framework, with Terraform (2019), a multi-cloud tool. However, tools for modeling business processes and support them within a decoupled microservice infrastructure are not provided.

Indrasiri & Siriwardena (2018) introduce Ballerina, an emerging technology that is built as a programming language and aims to make it easy to write programs that integrate and orchestrate microservices. However, although they propose an environment to design microservice integrations with sequence diagrams, most of the communication issues among microservices need to be managed at programming level. Our solution automatically generates the implementation artifacts required to support microservice communication from business process models.

Petrasch (2017) presents an approach based on UML to design microservices and communication among them. However, complex business processes involving multiple microservices cannot be modeled.

Guidi et al. (2017) present the need for specific programming languages aimed towards microservices composition. Authors claim that these languages should include concepts such as communication, interfaces, and dependencies. They instantiate their proposal in terms of the Jolie (2019) programming language. Similar work to this is the one presented by Safina et al. (2016), which extends the Jolie programming language to support data-driven workflows. This means that the flow of microservice compositions is controlled at the time of message passing according to the nature of the message structure and type. Our work differs from these two approaches in the fact that we provide a solution based on business process modeling instead of programming languages to create ad-hoc solutions.

Finally, it is worth noting that in this paper we present an extended version of the work proposed in (Valderas et al., 2019). In this current work, we introduce the evolution of microservice compositions from both, a top-down perspective (i.e. from the EUCalipTool composer to the microservices), and a bottom-up strategy (i.e. from the microservices to the EUCalipTool Composer). We have improved the DSML defining how inputs and outputs of microservices can be linked. We also present the development infrastructure implemented to support developers in the composition of microservices by using our approach. In addition, our approach has been evaluated through a complete experiment that compares it with ad-hoc solutions to compose microservices.

## 9 Conclusion and further work

In this work, we have presented a hybrid solution that combines the choreography and orchestration approaches to deal with microservice compositions with the use of EUCalipTool. The main reason to follow such a hybrid solution is that we want to take advantage of the goodness of each approach. This is, we want to maintain the flexibility and decoupling nature offered in choreographies but also want to keep the composition global vision and management offered by an orchestration approach. For this purpose, the EUCalipTool platform has been presented and integrated in a typical microservice architecture to provide: 1) tool support to the specification of microservices compositions, 2) mechanisms to automate the distributed deployment of microservice compositions and its execution through an event-based choreography, and 3) support the evolution of compositions following a top-down strategy (i.e. from the global vision of the composition) or a bottom-up strategy (i.e. from a piece of a specific business microservice).

In addition to the evaluation based on the motivating example, it would be very interesting to evaluate also the performance of the designed architecture in a real scenario. Furthermore, since our objective is to improve how compositions are made, as future work we plan to enrich EUCalipTool with goal-oriented capabilities. This way, instead of specifying compositions, users would just need to state their goals. Then, based on them, EUCalipTool would propose an initial composition intended to satisfy the user stated goals.

## Acknowledgments

This work has been developed with the financial support of the Spanish State Research Agency under the project TIN2017-84094-R and co-financed with ERDF.

## References

Alpers, S., Becker, C., Oberweis, A., Schuster, T. (2015). Microservice Based Tool Support for Business Process Modeling. EDOC Workshops: 71-78

Basili, V.R., Rombach, H.D. (1988). The TAME project: towards improvement-oriented software environments. IEEE Trans. Softw. Eng. 14(6), 758–773

Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S. T., and Mazzara, M. (2018). From Monolithic to Microservices: An Experience Report from the Banking Domain. IEEE Software, vol. 35, no. 3, pp. 50-55

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

Casati, F.: Models, Semantics, and Formal Methods for the design of Workflows and their Exceptions. (1998). PhD thesis, Milano

Dadam, P., Reichert, M. (2009). The ADEPT project: a decade of research and development for robust and flexible process support. Comp Scie - R&D 23: 81-97

Dragoni, N, Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. Present and Ulterior Software Engineering: 195-216

Dräxler, S., Karl, H., Peuster, M., Kouchaksaraei, H. R., Bredel, M., Lessmann, J., ... & Xilouris, G. (2017). SONATA: Service programming and orchestration for virtualized software networks. In 2017 IEEE International Conference on Communications Workshops (ICC Workshops) (pp. 973-978). IEEE.

Guidi, C., Lanese, I., Mazzara, M., & Montesi, F. (2017). Microservices: a language-based approach. In Present and Ulterior Software Engineering (pp. 217-225). Springer, Cham.

Hamidehkhan, P. (2019). Analysis and evaluation of composition languages and orchestration engines for microservices (Master's thesis).

Indrasiri, K., & Siriwardena, P. (2018). Integrating Microservices. In Microservices for the Enterprise (pp. 167-217). Apress, Berkeley, CA.

Jolie. (2019). A service oriented language. URL: https://www.jolie-lang.org/ Last time accesed: November 2019.

Kitchenham, B., Pickard, L. and Pfleeger, S. L. (1995). Case studies for method and tool evaluation, Software, IEEE, vol. 12, no. 4, pp. 52–62, 1995.

Newman, S. (2015). Building Microservices, USA:O'Reilly Media Inc., February 2015.

Petrasch, R. (2017). Model-based engineering for microservice architectures using enterprise integration patterns for inter-service communication. In 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE) (pp. 1-4). IEEE.

Rajasekar, A., Wan, M., Moore, R., & Schroeder, W. (2012). Micro-Services: A Service-Oriented Paradigm for. Data Intensive Distributed Computing. In: Challenges and Solutions for Large-scale Information Management (pp. 74-93). IGI Global.

Safina, L., Mazzara, M., Montesi, F., & Rivera, V. (2016). Data-driven workflows for microservices: Genericity in jolie. In 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA) (pp. 430-437). IEEE.

Shadija, D., Rezai, M., Hill, R. (2017). Towards an understanding of microservices. ICAC 2017: 1-6

Singhal, N., Sakthivel, U., & Raj, P. (2019). Selection Mechanism of Micro-Services Orchestration Vs. Choreography. International Journal of Web & Semantic Technology (IJWesT), 10(1), 25.

Terraform. (2019). URL: https://www.terraform.io/ Last time accesed: November 2019.

Valderas, P., Torres, T., Mansanet, M., Pelechano, V. (2017). A mobile-based solution for supporting end-users in the composition of services. Multimedia Tools Appl. 76 (15): 16315-16345

Valderas, P, Torres, V, and Pelechano, V. (2019). Hybrid Composition of Microservices with EUCalipTool. Proceedings of the XXII Iberoamerican Conference on Software Engineering, CIbSE 2019, La Habana, Cuba, April 22-26, 2019: 2-15.

Weber, B., Reichert, M., Rinderle, S. (2008). Change Patterns and Change Support Features - Enhancing Flexibility in Process-Aware Information Systems. Data and Knowledge Engineering 66: 438-466

Wohlin, C., Runeson, P. , Höst, M., Ohlsson, M. C., Regnell, B. and Wesslén, A. (2012). Experimentation in Software Engineering, Springer.

Yahia, E. B. H., Réveillère, L., Bromberg, Y. D., Chevalier, R., & Cadot, A. (2016). Medley: An event-driven lightweight platform for service composition. In International Conference on Web Engineering (pp. 3-20). Springer, Cham.